NVIDIA TensorRT

Developer Guide | NVIDIA Docs

PG-08540-001_v10.2.0 | July 2024

Table of Contents

Revision History

This is the revision history of the NVIDIA TensorRT 10.2.0 Developer Guide.

Chapter 6 Updates

Date	Summary of Change
June 12, 2024	▶ Added a new section called Fine Grained Refit Build. ▶ Added a new section called Stripping Weights with Fine Grained Refit Build.

List of Figures

List of Tables

Chapter 1. Introduction

NVIDIA® TensorRT™ is an SDK that facilitates high-performance machine learning inference. It is designed to work in a complementary fashion with training frameworks such as TensorFlow, PyTorch, and MXNet. It focuses specifically on running an already-trained network quickly and efficiently on NVIDIA hardware.

Refer to the NVIDIA TensorRT Installation Guide for instructions on how to install TensorRT.

The NVIDIA TensorRT Quick Start Guide is for users who want to try out TensorRT SDK; specifically, you will learn how to construct an application to run inference on a TensorRT engine quickly.

1.1. Structure of This Guide

Chapter 1 provides information about how TensorRT is packaged and supported, and how it fits into the developer ecosystem.

Chapter 2 provides a broad overview of TensorRT capabilities.

Chapters three and four contain introductions to the C++ and Python APIs respectively.

Subsequent chapters provide more detail about advanced features.

The appendix contains a layer reference and answers to FAQs.

1.2. Samples

The NVIDIA TensorRT Sample Support Guide illustrates many of the topics discussed in this guide. Additional samples focusing on embedded applications can be found here.

1.3. Complementary GPU Features

Multi-Instance GPU, or MIG, is a feature of NVIDIA GPUs with NVIDIA Ampere Architecture or later architectures that enable user-directed partitioning of a single GPU into multiple smaller GPUs. The physical partitions provide dedicated compute and memory slices with QoS and independent execution of parallel workloads on fractions of the GPU. For TensorRT applications with low GPU utilization, MIG can produce

higher throughput at small or no impact on latency. The optimal partitioning scheme is application-specific.

1.4. Complementary Software

The NVIDIA Triton™ Inference Server is a higher-level library providing optimized inference across CPUs and GPUs. It provides capabilities for starting and managing multiple models, and REST and gRPC endpoints for serving inference.

NVIDIA DALI® provides high-performance primitives for preprocessing image, audio, and video data. TensorRT inference can be integrated as a custom operator in a DALI pipeline. A working example of TensorRT inference integrated as a part of DALI can be found here.

TensorFlow-TensorRT (TF-TRT) is an integration of TensorRT directly into TensorFlow. It selects subgraphs of TensorFlow graphs to be accelerated by TensorRT, while leaving the rest of the graph to be executed natively by TensorFlow. The result is still a TensorFlow graph that you can execute as usual. For TF-TRT examples, refer to Examples for TensorRT in TensorFlow.

Torch-TensorRT (Torch-TRT) is a PyTorch-TensorRT compiler that converts PyTorch modules into TensorRT engines. Internally, the PyTorch modules are first converted into TorchScript/FX modules based on the Intermediate Representation (IR) selected. The compiler selects subgraphs of the PyTorch graphs to be accelerated by TensorRT, while leaving the rest of the graph to be executed natively by Torch. The result is still a PyTorch module that you can execute as usual. For examples, refer to Examples for Torch-TRT.

The TensorFlow-Quantization Toolkit provides utilities for training and deploying Tensorflow 2-based Keras models at reduced precision. This toolkit is used to quantize different layers in the graph exclusively based on operator names, class, and pattern matching. The quantized graph can then be converted into ONNX and then into TensorRT engines. For examples, refer to the model zoo.

The PyTorch Quantization Toolkit provides facilities for training PyTorch models at reduced precision, which can then be exported for optimization in TensorRT.

In addition, the PyTorch Automatic Sparsity (ASP) tool provides facilities for training models with structured sparsity, which can then be exported and allows TensorRT to use the faster sparse tactics on NVIDIA Ampere Architecture GPUs.

TensorRT is integrated with NVIDIA's profiling tools, NVIDIA Nsight™ Systems and NVIDIA Deep Learning Profiler (DLProf).

A restricted subset of TensorRT is certified for use in NVIDIA DRIVE® products. Some APIs are marked for use only in NVIDIA DRIVE and are not supported for general use.

1.5. ONNX

TensorRT's primary means of importing a trained model from a framework is through the ONNX interchange format. TensorRT ships with an ONNX parser library to assist in

importing models. Where possible, the parser is backward compatible up to opset 9; the ONNX Model Opset Version Converter can assist in resolving incompatibilities.

The GitHub version may support later opsets than the version shipped with TensorRT. Refer to the ONNX-TensorRT operator support matrix for the latest information on the supported opset and operators. For TensorRT deployment, we recommend exporting to the latest available ONNX opset.

The ONNX operator support list for TensorRT can be found here.

PyTorch natively supports ONNX export. For TensorFlow, the recommended method is tf2onnx.

A good first step after exporting a model to ONNX is to run constant folding using Polygraphy. This can often solve TensorRT conversion issues in the ONNX parser and generally simplify the workflow. For details, refer to this example. In some cases, it may be necessary to modify the ONNX model further, for example, to replace subgraphs with plugins or reimplement unsupported operations in terms of other operations. To make this process easier, you can use ONNX-GraphSurgeon.

1.6. Code Analysis Tools

For guidance using the valgrind and clang sanitizer tools with TensorRT, refer to the Troubleshooting chapter.

1.7. API Versioning

TensorRT version number (MAJOR.MINOR.PATCH) follows Semantic Versioning 2.0.0 for its public APIs and library ABIs. Version numbers change as follows:

1. MAJOR version when making incompatible API or ABI changes
2. MINOR version when adding functionality in a backward compatible manner
3. PATCH version when making backward compatible bug fixes

Note that semantic versioning does not extend to serialized objects. To reuse plan files, and timing caches, version numbers must match across major, minor, patch, and build versions (with some exceptions for the safety runtime as detailed in the NVIDIA DRIVE OS 6.0 Developer Guide). Calibration caches can typically be reused within a major version but compatibility is not guaranteed.

1.8. Deprecation Policy

Deprecation is used to inform developers that some APIs and tools are no longer recommended for use. Beginning with version 8.0, TensorRT has the following deprecation policy:

- Deprecation notices are communicated in the NVIDIA TensorRT Release Notes.

- When using the C++ API:
  - API functions are marked with the `TRT_DEPRECATED_API` macro.
  - Enums are marked with the `TRT_DEPRECATED_ENUM` macro.
  - All other locations are marked with the `TRT_DEPRECATED` macro.
  - Classes, functions, and objects will have a statement documenting when they were deprecated.
- When using the Python API, deprecated methods and classes will issue deprecation warnings at runtime, if they are used.
- TensorRT provides a 12-month migration period after the deprecation.
- APIs and tools continue to work during the migration period.
- After the migration period ends, APIs and tools are removed in a manner consistent with semantic versioning.

For any APIs and tools specifically deprecated in TensorRT 7.x, the 12-month migration period starts from the TensorRT 8.0 GA release date.

1.9. Hardware Support Lifetime

TensorRT 8.5.3 was the last release supporting NVIDIA Kepler (SM 3.x) and NVIDIA Maxwell (SM 5.x) devices. These devices are no longer supported in TensorRT 8.6. NVIDIA Pascal (SM 6.x) devices are deprecated in TensorRT 8.6.

1.10. Support

Support, resources, and information about TensorRT can be found online at https://developer.nvidia.com/tensorrt. This includes blogs, samples, and more.

In addition, you can access the NVIDIA DevTalk TensorRT forum at https://devtalk.nvidia.com/default/board/304/tensorrt/ for all things related to TensorRT. This forum offers the possibility of finding answers, making connections, and getting involved in discussions with customers, developers, and TensorRT engineers.

1.11. Reporting Bugs

NVIDIA appreciates all types of feedback. If you encounter any problems, follow the instructions in the Reporting TensorRT Issues section to report the issues.

Chapter 2. TensorRT's Capabilities

This chapter provides an overview of what you can do with TensorRT. It is intended to be useful to all TensorRT users.

2.1. C++ and Python APIs

TensorRT's API has language bindings for both C++ and Python, with nearly identical capabilities. The Python API facilitates interoperability with Python data processing toolkits and libraries like NumPy and SciPy. The C++ API can be more efficient, and may better meet some compliance requirements, for example in automotive applications.

Note: The Python API is not available for all platforms. For more information, refer to the NVIDIA TensorRT Support Matrix.

2.2. The Programming Model

TensorRT operates in two phases. In the first phase, usually performed offline, you provide TensorRT with a model definition, and TensorRT optimizes it for a target GPU. In the second phase, you use the optimized model to run inference.

2.2.1. The Build Phase

The highest-level interface for the build phase of TensorRT is the *`Builder`* (C++, Python). The builder is responsible for optimizing a model, and producing an *`Engine`*.

In order to build an engine, you must:

- Create a network definition.
- Specify a configuration for the builder.
- Call the builder to create the engine.

The *`NetworkDefinition`* interface (C++, Python) is used to define the model. The most common path to transfer a model to TensorRT is to export it from a framework in ONNX format, and use TensorRT's ONNX parser to populate the network definition. However,

you can also construct the definition step by step using TensorRT's *Layer* (C++, Python) and *Tensor* (C++, Python) interfaces.

Whichever way you choose, you must also define which tensors are the inputs and outputs of the network. Tensors that are not marked as outputs are considered to be transient values that can be optimized away by the builder. Input and output tensors must be named, so that at runtime, TensorRT knows how to bind the input and output buffers to the model.

The *BuilderConfig* interface (C++, Python) is used to specify how TensorRT should optimize the model. Among the configuration options available, you can control TensorRT's ability to reduce the precision of calculations, control the tradeoff between memory and runtime execution speed, and constrain the choice of CUDA® kernels. Since the builder can take minutes or more to run, you can also control how the builder searches for kernels, and cached search results for use in subsequent runs.

After you have a network definition and a builder configuration, you can call the builder to create the engine. The builder eliminates dead computations, folds constants, and reorders and combines operations to run more efficiently on the GPU. It can optionally reduce the precision of floating-point computations, either by simply running them in 16-bit floating point, or by quantizing floating point values so that calculations can be performed using 8-bit integers. It also times multiple implementations of each layer with varying data formats, then computes an optimal schedule to execute the model, minimizing the combined cost of kernel executions and format transforms.

The builder creates the engine in a serialized form called a *plan*, which can be deserialized immediately, or saved to disk for later use.

Note:

- By default, engines created by TensorRT are specific to both the TensorRT version with which they were created and the GPU on which they were created. Refer to the Version Compatibility and Hardware Compatibility sections for how to configure an engine for forward compatibility.
- TensorRT's network definition does not deep-copy parameter arrays (such as the weights for a convolution). Therefore, you must not release the memory for those arrays until the build phase is complete. When importing a network using the ONNX parser, the parser owns the weights, so it must not be destroyed until the build phase is complete.
- The builder times algorithms to determine the fastest. Running the builder in parallel with other GPU work may perturb the timings, resulting in poor optimization.

2.2.2. The Runtime Phase

The highest-level interface for the execution phase of TensorRT is the *Runtime* (C++, Python).

When using the runtime, you will typically carry out the following steps:

- Deserialize a plan to create an engine.
- Create an execution context from the engine.

Then, repeatedly:

- Populate input buffers for inference.
- Call `enqueueV3()` on the execution context to run inference.

The *`Engine`* interface (C++, Python) represents an optimized model. You can query an engine for information about the input and output tensors of the network - the expected dimensions, data type, data format, and so on.

The *`ExecutionContext`* interface (C++, Python), created from the engine is the main interface for invoking inference. The execution context contains all of the state associated with a particular invocation - thus you can have multiple contexts associated with a single engine, and run them in parallel.

When invoking inference, you must set up the input and output buffers in the appropriate locations. Depending on the nature of the data, this may be in either CPU or GPU memory. If not obvious based on your model, you can query the engine to determine in which memory space to provide the buffer.

After the buffers are set up, inference can be enqueued (`enqueueV3`). The required kernels are enqueued on a CUDA stream, and control is returned to the application as soon as possible. Some networks require multiple control transfers between CPU and GPU, so control may not return immediately. To wait for completion of asynchronous execution, synchronize on the stream using `cudaStreamSynchronize`.

2.3. Plugins

TensorRT has a *`Plugin`* interface to allow applications to provide implementations of operations that TensorRT does not support natively. Plugins that are created and registered with TensorRT's `PluginRegistry` can be found by the ONNX parser while translating the network.

TensorRT ships with a library of plugins, and source for many of these and some additional plugins can be found here.

You can also write your own plugin library and serialize it with the engine.

If cuDNN or cuBLAS is needed, install the library as TensorRT no longer ships with them. To obtain `cudnnContext*` or `cublasContext*`, the corresponding `TacticSource` flag must be set using `nvinfer1::IBuilderConfig::setTacticSource()`.

Refer to the Extending TensorRT with Custom Layers chapter for more details.

2.4. Types and Precision

2.4.1. Supported Types

TensorRT supports FP32, FP16, BF16, FP8, INT4, INT8, INT32, INT64, UINT8, and BOOL data types. Refer to the TensorRT Operator documentation for layer I/O data type specification.

- FP32, FP16, BF16: unquantized floating point types
- INT8: low-precision integer type
  - Implicit quantization
    - Interpreted as a quantized integer. A tensor with INT8 type must have an associated scale factor (either through calibration or `setDynamicRange` API).
  - Explicit quantization
    - Interpreted as a signed integer. Conversion to/from INT8 type requires an explicit Q/DQ layer.
- INT4: low-precision integer type for weight compression
  - INT4 is used for weight-only-quantization. Requires dequantization before compute is performed.
  - Conversion to and from INT4 type requires an explicit Q/DQ layer.
  - INT4 weights are expected to be serialized by packing two elements per-byte. Refer to the Quantized Weights section for additional information.
- FP8: low-precision floating-point type
  - 8-bit floating point type with 1-bit for sign, 4-bits for exponent, 3-bits for mantissa
  - Conversion to/from FP8 type requires an explicit Q/DQ layer.
- UINT8: unsigned integer I/O type
  - Data type only usable as a network I/O type.
  - Network level inputs in UINT8 must be converted from UINT8 to either FP32 or FP16 using a `CastLayer` before the data is used in other operations.
  - Network-level outputs in UINT8 must be produced by a `CastLayer` that has been explicitly inserted into the network (will only support conversions from FP32/FP16 to UINT8).
  - UINT8 quantization is not supported.
  - The `ConstantLayer` does not support UINT8 as an output type.
- BOOL
  - A boolean type used with supported layers.

2.4.2. Strong Typing vs Weak Typing

When providing a network to TensorRT, you specify whether it is strongly or weakly typed, with the default being weakly typed.

For strongly typed networks, TensorRT's optimizer will statically infer intermediate tensor types based on the network input types and the operator specifications, which match type inference semantics in frameworks. The optimizer will then adhere strictly to those types. For more information, refer to Strongly Typed Networks.

For weakly typed networks, TensorRT's optimizer may substitute different precisions for tensors if it increases performance. In this mode, TensorRT defaults to FP32 for all floating-point operations, but there are two ways to configure different levels of precision:

- To control precision at the model level, `BuilderFlag` options (C++, Python) can indicate to TensorRT that it may select lower-precision implementations when searching for the fastest (and because lower precision is generally faster, if allowed to, it typically will).

  For example, by setting a single flag you can easily instruct TensorRT to use FP16 calculations for your entire model. For regularized models whose input dynamic range is approximately one, this typically produces significant speedups with negligible change in accuracy.
- For finer-grained control, where a layer must run at higher precision because part of the network is numerically sensitive or requires high dynamic range, arithmetic precision can be specified for that layer.

Refer to Reduced Precision in Weakly Typed Networks for more details.

2.5. Quantization

TensorRT supports quantized floating point, where floating-point values are linearly compressed and rounded to low precision quantized types (INT8, FP8, INT4). This significantly increases arithmetic throughput while reducing storage requirements and memory bandwidth. When quantizing a floating-point tensor, TensorRT must know its dynamic range - that is, what range of values is important to represent - values outside this range are clamped when quantizing.

Dynamic range information can be calculated by the builder (this is called *calibration*) based on representative input data (this is currently supported only for INT8). Or you can perform quantization-aware training in a framework and import the model to TensorRT with the necessary dynamic range information.

Refer to the Working with Quantized Types chapter for more details.

2.6. Tensors and Data Formats

When defining a network, TensorRT assumes that tensors are represented by multidimensional C-style arrays. Each layer has a specific interpretation of its inputs: for example, a 2D convolution will assume that the last three dimensions of its input are in CHW format - there is no option to use, for example a WHC format. Refer to NVIDIA TensorRT Operator's Reference for how each layer interprets its inputs.

Note that tensors are limited to at most 2^31-1 elements.

While optimizing the network, TensorRT performs transformations internally (including to HWC, but also more complex formats) to use the fastest possible CUDA kernels. In general, formats are chosen to optimize performance, and applications have no control over the choices. However, the underlying data formats are exposed at I/O boundaries (network input and output, and passing data to and from plugins) to allow applications to minimize unnecessary format transformations.

Refer to the I/O Formats section for more details.

2.7. Dynamic Shapes

By default, TensorRT optimizes the model based on the input shapes (batch size, image size, and so on) at which it was defined. However, the builder can be configured to allow the input dimensions to be adjusted at runtime. In order to enable this, you specify one or more instances of `OptimizationProfile` (C++, Python) in the builder configuration, containing for each input a minimum and maximum shape, along with an optimization point within that range.

TensorRT creates an optimized engine for each profile, choosing CUDA kernels that work for all shapes within the [minimum, maximum] range and are fastest for the optimization point - typically different kernels for each profile. You can then select among profiles at runtime.

Refer to the Working with Dynamic Shapes chapter for more details.

2.8. DLA

TensorRT supports NVIDIA's Deep Learning Accelerator (DLA), a dedicated inference processor present on many NVIDIA SoCs that supports a subset of TensorRT's layers. TensorRT allows you to execute part of the network on the DLA and the rest on GPU; for layers that can be executed on either device, you can select the target device in the builder configuration on a per-layer basis.

Refer to the Working with DLA chapter for more details.

2.9. Updating Weights

When building an engine, you can specify that it may later have its weights updated. This can be useful if you are frequently updating the weights of the model without changing the structure, such as in reinforcement learning or when retraining a model while retaining the same structure. Weight updates are performed using the *`Refitter`* (C++, Python) interface.

Refer to the Refitting an Engine section for more details.

2.10. Streaming Weights

TensorRT can be configured to stream the network's weights from host memory to device memory during network execution instead of placing them in device memory at engine load time. This enables models with weights larger than free GPU memory to run, but potentially with significantly increased latency. Weight streaming is an opt-in feature at both build time (`BuilderFlag::kWEIGHT_STREAMING`) and runtime (`ICudaEngine::setWeightStreamingBudgetV2`).

Note: Weight streaming is only supported with strongly typed networks. For more information, refer to Weight Streaming.

2.11. `trtexec` Tool

Included in the `samples` directory is a command-line wrapper tool called `trtexec`. `trtexec` is a tool to use TensorRT without having to develop your own application. The `trtexec` tool has three main purposes:

- *benchmarking networks* on random or user-provided input data.
- *generating serialized engines* from models.
- *generating a serialized timing cache* from the builder.

Refer to the trtexec section for more details.

2.12. Polygraphy

Polygraphy is a toolkit designed to assist in running and debugging deep learning models in TensorRT and other frameworks. It includes a Python API and a command-line interface (CLI) built using this API.

Among other things, with Polygraphy you can:

- Run inference among multiple backends, like TensorRT and ONNX-Runtime, and compare results (for example API,CLI).
- Convert models to various formats, for example, TensorRT engines with post-training quantization (for example API,CLI).
- View information about various types of models (for example CLI)
- Modify ONNX models on the command line:
  - Extract subgraphs (for example CLI).
  - Simplify and sanitize (for example CLI).
- Isolate faulty tactics in TensorRT (for example CLI).

For more details, refer to the Polygraphy repository.

Chapter 3. The C++ API

This chapter illustrates basic usage of the C++ API, assuming you are starting with an ONNX model. sampleOnnxMNIST illustrates this use case in more detail.

The C++ API can be accessed through the header `NvInfer.h`, and is in the `nvinfer1` namespace. For example, a simple application might begin with:

```
#include "NvInfer.h"

using namespace nvinfer1;
```

Interface classes in the TensorRT C++ API begin with the prefix `I`, for example `ILogger`, `IBuilder`, and so on.

A CUDA context is automatically created the first time TensorRT makes a call to CUDA, if none exists before that point. It is generally preferable to create and configure the CUDA context yourself before the first call to TensorRT.

In order to illustrate object lifetimes, code in this chapter does not use smart pointers; however, their use is recommended with TensorRT interfaces.

3.1. The Build Phase

To create a builder, you first must instantiate the `ILogger` interface. This example captures all warning messages but ignores informational messages:

```
class Logger : public ILogger
{
    void log(Severity severity, const char* msg) noexcept override
    {
        // suppress info-level messages
        if (severity <= Severity::kWARNING)
            std::cout << msg << std::endl;
    }
} logger;
```

You can then create an instance of the builder:

```
IBuilder* builder = createInferBuilder(logger);
```

3.1.1. Creating a Network Definition

After the builder has been created, the first step in optimizing a model is to create a network definition. The network creation options are specified using a combination of flags OR-d together.

The `kEXPLICIT_BATCH` flag is required in order to import models using the ONNX parser. For more information, refer to Explicit Versus Implicit Batch.

You can also specify that the network should be considered strongly typed using the `NetworkDefinitionCreationFlag::kSTRONGLY_TYPED` flag. For more information, refer to Strongly Typed Networks.

Finally, create a network:

```
INetworkDefinition* network = builder->createNetworkV2(flag);
```

3.1.2. Importing a Model Using the ONNX Parser

Now, the network definition must be populated from the ONNX representation. The ONNX parser API is in the file `NvOnnxParser.h`, and the parser is in the `nvonnxparser` C++ namespace.

```
#include "NvOnnxParser.h"

using namespace nvonnxparser;
```

You can create an ONNX parser to populate the network as follows:

```
IParser* parser = createParser(*network, logger);
```

Then, read the model file and process any errors.

```
parser->parseFromFile(modelFile,
    static_cast<int32_t>(ILogger::Severity::kWARNING));
for (int32_t i = 0; i < parser->getNbErrors(); ++i)
{
std::cout << parser->getError(i)->desc() << std::endl;
}
```

An important aspect of a TensorRT network definition is that it contains pointers to model weights, which are copied into the optimized engine by the builder. Since the network was created using the parser, the parser owns the memory occupied by the weights, and so the parser object should not be deleted until after the builder has run.

3.1.3. Building an Engine

The next step is to create a build configuration specifying how TensorRT should optimize the model.

```
IBuilderConfig* config = builder->createBuilderConfig();
```

This interface has many properties that you can set in order to control how TensorRT optimizes the network. One important property is the maximum workspace size. Layer implementations often require a temporary workspace, and this parameter limits the maximum size that any layer in the network can use. If insufficient workspace is provided, it is possible that TensorRT will not be able to find an implementation for a layer. By default the workspace is set to the total global memory size of the given device; restrict it when necessary, for example, when multiple engines are to be built on a single device.

```
config->setMemoryPoolLimit(MemoryPoolType::kWORKSPACE, 1U << 20);
```

Another significant consideration is the maximum shared memory allocation for the CUDA backend implementation. This allocation becomes pivotal in scenarios where TensorRT needs to coexist with other applications, such as when the GPU is concurrently utilized by both TensorRT and DirectX.

```
config->setMemoryPoolLimit(MemoryPoolType::kTACTIC_SHARED_MEMORY, 48 << 10);
```

Once the configuration has been specified, the engine can be built.

```
IHostMemory* serializedModel = builder->buildSerializedNetwork(*network, *config);
```

Since the serialized engine contains the necessary copies of the weights, the parser, network definition, builder configuration and builder are no longer necessary and may be safely deleted:

```
delete parser;
delete network;
delete config;
delete builder;
```

The engine can then be saved to disk, and the buffer into which it was serialized can be deleted.

```
delete serializedModel
```

Note: Serialized engines are not portable across platforms. Engines are specific to the exact GPU model that they were built on (in addition to the platform).

Since building engines is intended as an offline process, it can take significant time. Refer to the Optimizing Builder Performance section for how to make the builder run faster.

3.2. Deserializing a Plan

Assuming you have previously serialized an optimized model and want to perform inference, you must create an instance of the `Runtime` interface. Like the builder, the runtime requires an instance of the logger:

```
IRuntime* runtime = createInferRuntime(logger);
```

After you have read the model into a buffer, you can deserialize it to obtain an engine:

```
ICudaEngine* engine =
  runtime->deserializeCudaEngine(modelData, modelSize);
```

3.3. Performing Inference

The engine holds the optimized model, but to perform inference you must manage additional state for intermediate activations. This is done using the `ExecutionContext` interface:

```
IExecutionContext *context = engine->createExecutionContext();
```

An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. (A current exception to this is when using

dynamic shapes, when each optimization profile can only have one execution context, unless the preview feature, `kPROFILE_SHARING_0806`, is specified.)

To perform inference, you must pass TensorRT buffers for input and output, which TensorRT requires you to specify with calls to `setTensorAddress`, which takes the name of the tensor and the address of the buffer. You can query the engine using the names you provided for input and output tensors to find the right positions in the array:

```
context->setTensorAddress(INPUT_NAME, inputBuffer);
context->setTensorAddress(OUTPUT_NAME, outputBuffer);
```

If the engine was built with dynamic shapes, you must also specify the input shapes:

```
context->setInputShape(INPUT_NAME, inputDims);
```

You can then call TensorRT's method `enqueueV3` to start inference using a CUDA stream:

```
context->enqueueV3(stream);
```

A network will be executed asynchronously or not depending on the structure and features of the network. A non-exhaustive list of features that can cause synchronous behavior are data dependent shapes, DLA usage, loops, and plugins that are synchronous, for example. It is common to enqueue data transfers with `cudaMemcpyAsync()` before and after the kernels to move data from the GPU if it is not already there.

To determine when the kernels (and possibly `cudaMemcpyAsync()`) are complete, use standard CUDA synchronization mechanisms such as events or waiting on the stream.

Chapter 4. The Python API

This chapter illustrates basic usage of the Python API, assuming you are starting with an ONNX model. The onnx_resnet50.py sample illustrates this use case in more detail.

The Python API can be accessed through the `tensorrt` module:

```
import tensorrt as trt
```

4.1. The Build Phase

To create a builder, you must first create a logger. The Python bindings include a simple logger implementation that logs all messages preceding a certain severity to `stdout`.

```
logger = trt.Logger(trt.Logger.WARNING)
```

Alternatively, it is possible to define your own implementation of the logger by deriving from the `ILogger` class:

```
class MyLogger(trt.ILogger):
    def __init__(self):
       trt.ILogger.__init__(self)

    def log(self, severity, msg):
        pass # Your custom logging implementation here

logger = MyLogger()
```

You can then create a builder:

```
builder = trt.Builder(logger)
```

Since building engines is intended as an offline process, it can take significant time. Refer to the Optimizing Builder Performance section for how to make the builder run faster.

4.1.1. Creating a Network Definition in Python

After the builder has been created, the first step in optimizing a model is to create a network definition. The network definition options are specified using a combination of flags OR-d together.

The `EXPLICIT_BATCH` flag is required in order to import models using the ONNX parser. For more information, refer to Explicit Versus Implicit Batch.

You can also specify that the network should be considered strongly typed using the `NetworkDefinitionCreationFlag.STRONGLY_TYPED` flag. For more information, refer to Strongly Typed Networks.

Finally, create a network:

```
network = builder.create_network(flag)
```

4.1.2. Importing a Model Using the ONNX Parser

Now, the network definition must be populated from the ONNX representation. You can create an ONNX parser to populate the network as follows:

```
parser = trt.OnnxParser(network, logger)
```

Then, read the model file and process any errors:

```
success = parser.parse_from_file(model_path)
for idx in range(parser.num_errors):
    print(parser.get_error(idx))

if not success:
    pass # Error handling code here
```

4.1.3. Building an Engine

The next step is to create a build configuration specifying how TensorRT should optimize the model:

```
config = builder.create_builder_config()
```

This interface has many properties that you can set in order to control how TensorRT optimizes the network. One important property is the maximum workspace size. Layer implementations often require a temporary workspace, and this parameter limits the maximum size that any layer in the network can use. If insufficient workspace is provided, it is possible that TensorRT will not be able to find an implementation for a layer. By default, the workspace is set to the total global memory size of the given device; restrict it when necessary, for example, when multiple engines are to be built on a single device.

```
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 20) # 1 MiB
```

After the configuration has been specified, the engine can be built and serialized with:

```
serialized_engine = builder.build_serialized_network(network, config)
```

It may be useful to save the engine to a file for future use. You can do that like so:

```
with open("sample.engine", "wb") as f:
    f.write(serialized_engine)
```

Note: Serialized engines are not portable across platforms. Engines are specific to the exact GPU model that they were built on (in addition to the platform).

4.2. Deserializing a Plan

To perform inference, deserialize the engine using the `Runtime` interface. Like the builder, the runtime requires an instance of the logger.

```
runtime = trt.Runtime(logger)
```

You can then deserialize the engine from a memory buffer:

```
engine = runtime.deserialize_cuda_engine(serialized_engine)
```

If you want, first load the engine from a file:

```
with open("sample.engine", "rb") as f:
    serialized_engine = f.read()
```

4.3. Performing Inference

The engine holds the optimized model, but to perform inference requires additional state for intermediate activations. This is done using the `IExecutionContext` interface:

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

An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. (A current exception to this is when using dynamic shapes, when each optimization profile can only have one execution context, unless the preview feature, `PROFILE_SHARING_0806`, is specified.)

To perform inference, you must specify buffers for inputs and outputs:

```
context.set_tensor_address(name, ptr)
```

Several Python packages allow you to allocate memory on the GPU, including, but not limited to, the official CUDA Python bindings, PyTorch, cuPy, and Numba.

After populating the input buffer, you can call TensorRT's `execute_async_v3` method to start inference using a CUDA stream. A network will be executed asynchronously or not depending on the structure and features of the network. A non-exhaustive list of features that can cause synchronous behavior are data dependent shapes, DLA usage, loops, and plugins that are synchronous, for example.

First, create the CUDA stream. If you already have a CUDA stream, you can use a pointer to the existing stream. For example, for PyTorch CUDA streams, that is, `torch.cuda.Stream()`, you can access the pointer using the `cuda_stream` property; for Polygraphy CUDA streams, use the `ptr` attribute; or you can create a stream using CUDA Python binding directly by calling `cudaStreamCreate()`.

Next, start inference:

```
context.execute_async_v3(buffers, stream_ptr)
```

It is common to enqueue asynchronous transfers (`cudaMemcpyAsync()`) before and after the kernels to move data from the GPU if it is not already there.

To determine when inference (and asynchronous transfers) are complete, use the standard CUDA synchronization mechanisms such as events or waiting on the stream. For example, with PyTorch CUDA streams or Polygraphy CUDA streams, issue `stream.synchronize()`. With streams created with CUDA Python binding, issue `cudaStreamSynchronize(stream)`.

Chapter 5. How TensorRT Works

This chapter provides more detail on how TensorRT works.

5.1. Object Lifetimes

TensorRT's API is class-based, with some classes acting as factories for other classes. For objects owned by the user, the lifetime of a factory object must span the lifetime of objects it creates. For example, the `NetworkDefinition` and `BuilderConfig` classes are created from the `Builder` class, and objects of those classes should be destroyed before the builder factory object.

An important exception to this rule is creating an engine from a builder. After you have created an engine, you may destroy the builder, network, parser, and build config and continue using the engine.

5.2. Error Handling and Logging

When creating TensorRT top-level interfaces (builder, runtime or refitter), you must provide an implementation of the *`Logger`* (C++, Python) interface. The logger is used for diagnostics and informational messages; its verbosity level is configurable. Since the logger may be used to pass back information at any point in the lifetime of TensorRT, its lifetime must span any use of that interface in your application. The implementation must also be thread-safe, since TensorRT may use worker threads internally.

An API call to an object will use the logger associated with the corresponding top-level interface. For example, in a call to `ExecutionContext::enqueueV3()`, the execution context was created from an engine, which was created from a runtime, so TensorRT will use the logger associated with that runtime.

The primary method of error handling is the *`ErrorRecorder`* (C++, Python) interface. You can implement this interface, and attach it to an API object to receive errors associated with that object. The recorder for an object will also be passed to any others it creates - for example, if you attach an error recorder to an engine, and create an execution context from that engine, it will use the same recorder. If you then attach a new error recorder to the execution context, it will receive only errors coming from that context. If an error is generated but no error recorder is found, it will be emitted through the associated logger.

Note that CUDA errors are generally asynchronous - so when performing multiple inferences or other streams of CUDA work asynchronously in a single CUDA context, an asynchronous GPU error may be observed in a different execution context than the one that generated it.

5.3. Memory

TensorRT uses considerable amounts of device memory, (that is, memory directly accessible by the GPU, as opposed to the host memory attached to the CPU). Since device memory is often a constrained resource, it is important to understand how TensorRT uses it.

5.3.1. The Build Phase

During build, TensorRT allocates device memory for timing layer implementations. Some implementations can consume a large amount of temporary memory, especially with large tensors. You can control the maximum amount of temporary memory through the memory pool limits of the builder config. The workspace size defaults to the full size of device global memory but can be restricted when necessary. If the builder finds applicable kernels that could not be run because of insufficient workspace, it will emit a logging message indicating this.

Even with relatively little workspace however, timing requires creating buffers for input, output, and weights. TensorRT is robust against the operating system (OS) returning out-of-memory for such allocations. On some platforms the OS may successfully provide memory, which then the out-of-memory killer process observes that the system is low on memory, and kills TensorRT. If this happens free up as much system memory as possible before retrying.

During the build phase, there will typically be at least two copies of the weights in host memory: those from the original network, and those included as part of the engine as it is built. In addition, when TensorRT combines weights (for example convolution with batch normalization) additional temporary weight tensors will be created.

5.3.2. The Runtime Phase

At runtime, TensorRT uses relatively little host memory, but can use considerable amounts of device memory.

An engine, on deserialization, allocates device memory to store the model weights. Since the serialized engine is almost all weights, its size is a good approximation to the amount of device memory the weights require.

An `ExecutionContext` uses two kinds of device memory:

- Persistent memory required by some layer implementations - for example, some convolution implementations use edge masks, and this state cannot be shared between contexts as weights are, because its size depends on the layer input shape, which may vary across contexts. This memory is allocated on creation of the execution context, and lasts for its lifetime.

- Scratch memory, used to hold intermediate results while processing the network. This memory is used for intermediate activation tensors. It is also used for temporary storage required by layer implementations, the bound for which is controlled by `IBuilderConfig::setMemoryPoolLimit()`.

You may optionally create an execution context without scratch memory using `ICudaEngine::createExecutionContextWithoutDeviceMemory()` and provide that memory yourself for the duration of network execution. This allows you to share it between multiple contexts that are not running concurrently, or for other uses while inference is not running. The amount of scratch memory required is returned by `ICudaEngine::getDeviceMemorySizeV2()`.

Information about the amount of persistent memory and scratch memory used by the execution context is emitted by the builder when building the network, at severity `kINFO`. Examining the log, the messages look similar to the following:

```
[08/12/2021-17:39:11] [I] [TRT] Total Host Persistent Memory: 106528
[08/12/2021-17:39:11] [I] [TRT] Total Device Persistent Memory: 29785600
[08/12/2021-17:39:11] [I] [TRT] Total Scratch Memory: 9970688
```

By default, TensorRT allocates device memory directly from CUDA. However, you can attach an implementation of TensorRT's *`IGpuAllocator`* (C++, Python) interface to the builder or runtime and manage device memory yourself. This is useful if your application wants to control all GPU memory and suballocate to TensorRT instead of having TensorRT allocate directly from CUDA.

NVIDIA cuDNN and NVIDIA cuBLAS can occupy large amounts of device memory. TensorRT allows you to control whether these libraries are used for inference by using the `TacticSources` (C++, Python) attribute in the builder configuration. Some plugin implementations require these libraries, so that when they are excluded, the network may not be compiled successfully. The `cudnnContext` and `cublasContext` handles are passed to the plugins using `IPluginV2Ext::attachToContext()` if the appropriate tactic sources are set.

The CUDA infrastructure and TensorRT's device code also consume device memory. The amount of memory varies by platform, device, and TensorRT version. You can use `cudaGetMemInfo` to determine the total amount of device memory in use.

TensorRT measures the amount of memory in use before and after critical operations in builder and runtime. These memory usage statistics are printed to TensorRT's information logger. For example:

```
[MemUsageChange] Init CUDA: CPU +535, GPU +0, now: CPU 547, GPU 1293 (MiB)
```

It indicates the memory use change by CUDA initialization. `CPU +535, GPU +0` is the increased amount of memory after running CUDA initialization. The content after `now:` is the CPU/GPU memory usage snapshot after CUDA initialization.

Note: In a multi-tenant situation, the reported memory use by `cudaGetMemInfo` and TensorRT is prone to race conditions where a new allocation/free done by a different process or a different thread. Since CUDA is not in control of memory on unified-memory devices, the results returned by `cudaGetMemInfo` may not be accurate on these platforms.

5.3.3. CUDA Lazy Loading

CUDA lazy loading is a CUDA feature that can significantly reduce the peak GPU and host memory usage of TensorRT and speed up TensorRT initialization with negligible (< 1%) performance impact. The saving of memory usage and initialization time depends on the model, software stack, GPU platform, etc. It is enabled by setting the environment variable `CUDA_MODULE_LOADING=LAZY`. Refer to the NVIDIA CUDA documentation for more information.

5.3.4. L2 Persistent Cache Management

NVIDIA Ampere and later architectures support L2 cache persistence, a feature which allows prioritization of L2 cache lines for retention when a line is chosen for eviction. TensorRT can use this to retain activations in cache, reducing DRAM traffic, and power consumption.

Cache allocation is per-execution context, enabled using the context's `setPersistentCacheLimit` method. The total persistent cache among all contexts (and other components using this feature) should not exceed `cudaDeviceProp::persistingL2CacheMaxSize`. Refer to the NVIDIA CUDA Best Practices Guide for more information.

5.4. Threading

In general, TensorRT objects are not thread safe; accesses to an object from different threads must be serialized by the client.

The expected runtime concurrency model is that different threads will operate on different execution contexts. The context contains the state of the network (activation values, and so on) during execution, so using a context concurrently in different threads results in undefined behavior.

To support this model, the following operations are thread safe:

- Nonmodifying operations on a runtime or engine.
- Deserializing an engine from a TensorRT runtime.
- Creating an execution context from an engine.
- Registering and deregistering plugins.

There are no thread-safety issues with using multiple builders in different threads; however, the builder uses timing to determine the fastest kernel for the parameters provided, and using multiple builders with the same GPU will perturb the timing and TensorRT's ability to construct optimal engines. There are no such issues using multiple threads to build with different GPUs.

5.5. Determinism

The TensorRT builder uses timing to find the fastest kernel to implement a given operator. Timing kernels is subject to noise - other work running on the GPU, fluctuations in GPU clock speed, and so on. Timing noise means that on successive runs of the builder, the same implementation may not be selected.

In general, different implementations will use a different order of floating point operations, resulting in small differences in the output. The impact of these differences on the final result is usually very small. However, when TensorRT is configured to optimize by tuning over multiple precisions, the difference between an FP16 and an FP32 kernel can be more significant, particularly if the network has not been well regularized or is otherwise sensitive to numerical drift.

Other configuration options that can result in a different kernel selection are different input sizes (for example, batch size) or a different optimization point for an input profile (refer to the Working with Dynamic Shapes section).

The *`AlgorithmSelector`* (C++, Python) interface allows you to force the builder to pick a particular implementation for a given layer. You can use this to ensure that the same kernels are picked by the builder from run to run. For more information, refer to the Algorithm Selection and Reproducible Builds section.

After an engine has been built, except for `IFillLayer` and `IScatterLayer`, it is deterministic: providing the same input in the same runtime environment will produce the same output.

5.5.1. IFillLayer Determinism

When `IFillLayer` is added to a network using either the `RANDOM_UNIFORM` or `RANDOM_NORMAL` operations, the determinism guarantee above is no longer valid. On each invocation, these operations generate tensors based on the RNG state, and then update the RNG state. This state is stored on a per-execution context basis.

5.5.2. IScatterLayer Determinism

If `IScatterLayer` is added to a network, and the input tensor `indices` have duplicate entries, the determinism guarantee above is not valid for both `ScatterMode::kELEMENT` and `ScatterMode::kND` modes. Additionally, one of the values from the input `updates` tensor will be picked arbitrarily.

5.6. Runtime Options

TensorRT provides multiple runtime libraries to meet a variety of use cases. C++ applications that run TensorRT engines should link against one of the following:

- The *default* runtime is the main library (`libnvinfer.so/.dll`).

- The *lean* runtime library (`libnvinfer_lean.so/.dll`) is much smaller than the default library, and contains only the code necessary to run a version-compatible engine. It has some restrictions; primarily, it cannot refit or serialize engines.
- The *dispatch runtime* (`libnvinfer_dispatch.so/.dll`) is a small shim library that can load a lean runtime, and redirect calls to it. The dispatch runtime is capable of loading older versions of the lean runtime, and together with appropriate configuration of the builder, can be used to provide compatibility between a newer version of TensorRT and an older plan file. Using the dispatch runtime is almost the same as manually loading the lean runtime, but it checks that APIs are implemented by the lean runtime loaded, and performs some parameter mapping to support API changes where possible.

The lean runtime contains fewer operator implementations than the default runtime. Since TensorRT chooses operator implementations at build time, you need to specify that the engine should be built for the lean runtime by enabling version compatibility. It may be slightly slower than an engine built for the default runtime.

The lean runtime contains all the functionality of the dispatch runtime, and the default runtime contains all the functionality of the lean runtime.

TensorRT provides Python packages corresponding to each of the above libraries:

`tensorrt`
: A Python package. It is the Python interface for the *default* runtime.

`tensorrt_lean`
: A Python package. It is the Python interface for the *lean* runtime.

`tensorrt_dispatch`
: A Python package. It is the Python interface for the *dispatch* runtime.

Python applications that run TensorRT engines should import one of the above packages to load the appropriate library for their use case.

5.7. Compatibility

By default, serialized engines are only guaranteed to work correctly when used with the same OS, CPU architectures, GPU models, and TensorRT versions used to serialize the engines. Refer to the Version Compatibility and Hardware Compatibility sections for how to relax the constraints on TensorRT versions and GPU models.

Chapter 6. Advanced Topics

6.1. Version Compatibility

By default, TensorRT engines are compatible only with the version of TensorRT with which they are built. With appropriate build-time configuration, engines can be built that are compatible with later TensorRT versions. TensorRT engines built with TensorRT 8 will also be compatible with TensorRT 9 and TensorRT 10 runtimes, but not vice versa.

Version compatibility is supported from version 8.6; that is, the plan must be built with a version at least 8.6 or higher, and the runtime must be 8.6 or higher.

When using version compatibility, the API supported at runtime for an engine is the intersection of the API supported in the version with which it was built, and the API of the version used to run it. TensorRT removes APIs only on major version boundaries so this is not a concern within a major version. However, users wishing to use TensorRT 8 or TensorRT 9 engines with TensorRT 10 must migrate away from removed APIs, and are advised to avoid the deprecated APIs.

The recommended approach to creating a version-compatible engine is to build as follows:

C++

```
builderConfig.setFlag(BuilderFlag::kVERSION_COMPATIBLE);
IHostMemory* plan = builder->buildSerializedNetwork(network, config);
```

Python

```
builder_config.set_flag(tensorrt.BuilderFlag.VERSION_COMPATIBLE)
plan = builder.build_serialized_network(network, config)
```

If the network was created with TensorRT 8 or 9, it must have been created with `NetworkDefinitionCreationFlag::kEXPLICIT_BATCH`. TensorRT 10 makes `explicit batch` the default and impossible to turn off.

The request for a version compatible engine causes a copy of the lean runtime to be added to the plan. When you subsequently deserialize the plan, TensorRT recognizes that it contains a copy of the runtime. It loads the runtime, and uses it to deserialize and execute the rest of the plan. Because this results in code being loaded and run from the plan in the context of the owning process, you should only deserialize trusted plans this way. To indicate to TensorRT that you trust the plan, call:

C++

```
runtime->setEngineHostCodeAllowed(true);
```

Python

```
runtime.engine_host_code_allowed = True
```

The flag for trusted plans is also required if you are packaging plugins in the plan (refer to Plugin Shared Libraries).

6.1.1. Manually Loading the Runtime

The previous approach (Version Compatibility) packages a copy of the runtime with every plan, which can be prohibitive if your application uses a large number of models. An alternative approach is to manage the runtime loading yourself. For this approach, build version compatible plans as explained in the previous section, but also set an additional flag to exclude the lean runtime.

C++

```
builderConfig.setFlag(BuilderFlag::kVERSION_COMPATIBLE);
builderConfig.setFlag(BuilderFlag::kEXCLUDE_LEAN_RUNTIME);
IHostMemory* plan = builder->buildSerializedNetwork(network, config);
```

Python

```
builder_config.set_flag(tensorrt.BuilderFlag.VERSION_COMPATIBLE)
builder_config.set_flag(tensorrt.BuilderFlag.EXCLUDE_LEAN_RUNTIME)
plan = builder.build_serialized_network(network, config)
```

To run this plan, you must have access to the lean runtime for the version with which it was built. Suppose you have built the plan with TensorRT 8.6 and your application is linked against TensorRT 10, you can load the plan as follows.

C++

```
IRuntime* v10Runtime = createInferRuntime(logger);
IRuntime* v8ShimRuntime = v10Runtime->loadRuntime(v8RuntimePath);
engine = v8ShimRuntime->deserializeCudaEngine(v8plan);
```

Python

```
v10_runtime = tensorrt.Runtime(logger)
v8_shim_runtime = v10_runtime.load_runtime(v8_runtime_path)
engine = v8_shim_runtime.deserialize_cuda_engine(v8_plan)
```

The runtime will translate TensorRT 10 API calls for the TensorRT 8.6 runtime, checking to ensure that the call is supported and performing any necessary parameter remapping.

6.1.2. Loading from Storage

On most OSs, TensorRT can load the shared runtime library directly from memory. However, on Linux kernels prior to 3.17, a temporary directory is required. Use the `IRuntime::setTempfileControlFlags` and `IRuntime::setTemporaryDirectory` APIs to control TensorRT's use of these mechanisms.

6.1.3. Using Version Compatibility with the ONNX Parser

When building a version-compatible engine from a TensorRT network definition generated using TensorRT's ONNX parser, you must specify that the parser must use the native `InstanceNormalization` implementation instead of the plugin one.

To do this, use the `IParser::setFlag()` API.

C++

```
auto *parser = nvonnxparser::createParser(network, logger);
parser->setFlag(nvonnxparser::OnnxParserFlag::kNATIVE_INSTANCENORM);
```

Python

```
parser = trt.OnnxParser(network, logger)
parser.set_flag(trt.OnnxParserFlag.NATIVE_INSTANCENORM)
```

In addition, the parser may require the use of plugins in order to fully implement all ONNX operators used in the network. In particular, if the network is used to build a version-compatible engine, some plugins may need to be included with the engine (either serialized with the engine, or provided externally and loaded explicitly).

To query the list of plugin libraries needed to implement a particular parsed network, use the `IParser::getUsedVCPluginLibraries` API:

C++

```
auto *parser = nvonnxparser::createParser(network, logger);
parser->setFlag(nvonnxparser::OnnxParserFlag::kNATIVE_INSTANCENORM);
parser->parseFromFile(filename, static_cast<int>(ILogger::Severity::kINFO));
int64_t nbPluginLibs;
char const* const* pluginLibs = parser->getUsedVCPluginLibraries(nbPluginLibs);
```

Python

```
parser = trt.OnnxParser(network, logger)
parser.set_flag(trt.OnnxParserFlag.NATIVE_INSTANCENORM)

status = parser.parse_from_file(filename)
plugin_libs = parser.get_used_vc_plugin_libraries()
```

Refer to Plugin Shared Libraries, for how to use the resulting library list to serialize the plugins or package them externally.

6.2. Hardware Compatibility

By default, TensorRT engines are only compatible with the type of device where they were built. With build-time configuration, engines can be built that are compatible with other types of devices. Currently, hardware compatibility is supported only for Ampere and later device architectures and is not supported on NVIDIA DRIVE OS or JetPack.

For example, to build an engine compatible with all Ampere and newer architectures, configure the `IBuilderConfig` as follows:

```
config->setHardwareCompatibilityLevel(nvinfer1::HardwareCompatibilityLevel::kAMPERE_PLUS);
```

When building in hardware compatibility mode, TensorRT excludes tactics that are not hardware compatible, such as those that use architecture-specific instructions or require more shared memory than is available on some devices. Thus, a hardware-compatible engine may have lower throughput and/or higher latency than its non-hardware-compatible counterpart. The degree of this performance impact depends on the network architecture and input sizes.

6.3. Compatibility Checks

TensorRT records in a plan the major, minor, patch and build versions of the library used to create the plan. If these do not match the version of the runtime used to deserialize

the plan, it will fail to deserialize. When using version compatibility, the check will be performed by the lean runtime deserializing the plan data. By default, that lean runtime is included in the plan, and the match is guaranteed to succeed.

TensorRT also records the compute capability (major and minor versions) in the plan, and checks it against the GPU on which the plan is being loaded. If they do not match, the plan will fail to deserialize. This ensures that kernels selected during the build phase are present and can run. When using hardware compatibility, the check is relaxed; with `HardwareCompatibilityLevel::kAMPERE_PLUS`, the check will ensure that the compute capability is greater than or equal to 8.0 rather than checking for an exact match.

TensorRT additionally checks the following properties and will issue a warning if they do not match, except when using hardware compatibility:

- Global memory bus width
- L2 cache size
- Maximum shared memory per block and per multiprocessor
- Texture alignment requirement
- Number of multiprocessors
- Whether the GPU device is integrated or discrete

If GPU clock speeds differ between engine serialization and runtime systems, the chosen tactics from the serialization system may not be optimal for the runtime system and may incur some performance degradation.

If it is not possible to build a TensorRT engine for each individual type of GPU, you can select several GPUs to build engines with and run the engine on different GPUs with the same architecture. For example, among the NVIDIA RTX 40xx GPUs, you can build an engine with RTX 4090 and an engine with RTX 4070. At runtime, you can use the RTX 4090 engine on an RTX 4080 GPU, and the 4070 engine on all smaller GPUs. In most cases, the engine will run without functional issues and with only a small performance drop when compared to running the engine built with the same GPU.

However, if the engine requires a large amount of device memory, and if the device memory available during deserialization is smaller than when the engine was built, deserialization may fail. In this case, it is recommended to build the engine on a smaller GPU or to build the engine on the larger device with limited compute resources (refer to the Limiting Compute Resources section).

The safety runtime is able to deserialize engines generated in an environment where the major, minor, patch, and build version of TensorRT does not match exactly in some cases. Refer to the NVIDIA DRIVE OS 6.0 Developer Guide for more information.

6.4. Refitting an Engine

TensorRT can *refit* an engine with new weights without having to rebuild it, however, the option to do so must be specified when building:

```
...
config->setFlag(BuilderFlag::kREFIT)
```

```
builder->buildSerializedNetwork(network, config);
```

Later, you can create a `Refitter` object:

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

Then, update the weights. For example, to update a set of weights named “Conv Layer Kernel Weights”:

```
Weights newWeights = ...;
refitter->setNamedWeights("Conv Layer Kernel Weight",
                    newWeights);
```

The new weights should have the same count as the original weights used to build the engine. `setNamedWeights` returns false if something went wrong, such as a wrong weights name or a change in the weights count.

You can use `INetworkDefinition::setWeightsName()` to name weights at build time - the ONNX parser uses this API to associate the weights with the names used in the ONNX model. Otherwise, TensorRT will name the weights internally based on the related layer names and weights roles.

You can also pass GPU weights to refitter via:

```
Weights newBiasWeights = ...;
refitter->setNamedWeights("Conv Layer Bias Weight", newBiasWeights, TensorLocation::kDEVICE);
```

Because of the way the engine is optimized, if you change some weights, you might have to supply some other weights too. The interface can tell you what additional weights must be supplied.

This typically requires two calls to `IRefitter::getMissingWeights`, first to get the number of weights objects that must be supplied, and second to get their layers and roles.

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

You can supply the missing weights, in any order:

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

The set of missing weights returned is complete, in the sense that supplying only the missing weights does not generate a need for any more weights.

Once all the weights have been provided, you can update the engine:

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

If the refit returns `false`, check the log for a diagnostic; perhaps the issue is about weights that are still missing. There is also an async version, `refitCudaEngineAsync`, that can accept a stream parameter.

You can update the weights memory directly and then call `refitCudaEngine/refitCudaEngineAsync` in another iteration. If weights pointers need to be changed, call `setNamedWeights` to override the previous setting. Call `unsetNamedWeights` to unset

previously set weights so that they will not be used in later refitting and it becomes safe to release these weights.

After refitting is done, you can then delete the refitter:

```
delete refitter;
```

The updated engine behaves as if it had been built from a network updated with the new weights. And the previously created execution context can continue to be used after refitting the engine.

To view all refittable weights in an engine, use `refitter->getAllWeights(...)`; similarly to how `getMissingWeights` were used above.

6.4.1. Weight-Stripping

When refit is enabled, all the constant weights in the network can be updated after building the engine. However, this introduces both a cost to refit the engine with new weights, and a potential runtime impact because the inability to constant-fold weights may prevent the builder from performing some optimizations.

When the weights with which the engine will be refitted are unknown at build time, this cost is unavoidable. However, in some scenarios the weights are known. For example, you may be using TensorRT as one of multiple back ends to execute an ONNX model, and wish to avoid an additional copy of weights in the TensorRT plan.

The weight-stripping build configuration enables this scenario, when enabled, TensorRT enables refit only for constant weights that do not impact the builder's ability to optimize and produce an engine with the same runtime performance as a non-refittable engine. Those weights are then omitted from the serialized engine, resulting in a small plan file that can be refitted at runtime using the weights from the ONNX model.

The `trtexec` tool provides the `--stripWeights` flags that can be used to build the weight-stripped engine. Refer to the trtexec section for more details.

The following steps show how to refit the weights for weight-stripped engines. When working with ONNX models, the ONNX parser library can perform the refit automatically. Refer to Refitting a Weight-Stripped Engine Directly from ONNX for more information.

1. Set the corresponding builder flag to enable the weight-stripped build. Here, the `kSTRIP_PLAN` flag works with either `kREFIT` or `kREFIT_IDENTICAL`. It defaults to the latter. The `REFIT_IDENTICAL` flag instructs the TensorRT builder to optimize under the assumption that the engine will be refitted with weights identical to those provided at build time. The `kSTRIP_PLAN` flag minimizes plan size by stripping out the refittable weights.

   C++

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

   Python

   ```
   config.flags |= 1 << int(trt.BuilderFlag.STRIP_PLAN)
   config.flags |= 1 << int(trt.BuilderFlag.REFIT_IDENTICAL)
   ```

```
builder.build_serialized_network(network, config)
```

2. After the engine is built, save the engine plan file and distribute it in the installer.
3. On the client side, when you launch the network for the first time, update all the weights in the engine. Here, use the `getAllWeights` API since all the weights in the engine plan were removed.

   C++

   ```
   int32_t const n = refitter->getAllWeights(0, nullptr);
   ```

   Python

   ```
   all_weights = refitter.get_all()
   ```

4. Update the weights one by one.

   C++

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

   Python

   ```
   for name in wts_list:
       refitter.set_named_weights(name, weights[name])
   ```

5. Save the full engine plan file.

   C++

   ```
   auto serializationConfig = SampleUniquePtr<nvinfer1::ISerializationConfig>(cudaEngine-
   >createSerializationConfig());
   auto serializationFlag = serializationConfig->getFlags()
   serializationFlag &= ~(1<<
    static_cast<uint32_t>(nvinfer1::SerializationFlag::kEXCLUDE_WEIGHTS));
   serializationConfig->setFlags(serializationFlag)
   auto hostMemory = SampleUniquePtr<nvinfer1::IHostMemory>(cudaEngine-
   >serializeWithConfig(*serializationConfig));
   ```

   Python

   ```
   serialization_config = engine.create_serialization_config()
   serialization_config.flags &= ~(1 << int(trt.SerializationFlag.EXCLUDE_WEIGHTS))
   binary = engine.serialize_with_config(serialization_config)
   ```

   The application can now use the new full engine plan file for future inference.

6.4.2. Refitting a Weight-Stripped Engine Directly from ONNX

When working with weight-stripped engines created from ONNX models, the refit process can be done automatically with the `IParserRefitter` class from the ONNX parser library. The following steps show how to create the class and run the refit process.

1. Create your engine as described in Weight-Stripping, and create an `IRefitter` object.

   C++

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

   Python

   ```
   refitter = trt.Refitter(engine, TRT_LOGGER)
   ```

2. Create an `IParserRefitter` object.

   C++

   ```
   IParserRefitter* parserRefitter = createParserRefitter(*refitter, gLogger);
   ```

   Python

   ```
   parser_refitter = trt.OnnxParserRefitter(refitter, TRT_LOGGER)
   ```

3. Call the `refitFromFile()` function of the `IParserRefitter`. Ensure that the ONNX model provided is identical to the one used to create the weight-stripped engine. This function will return `true` if all the stripped-weights were found in the ONNX model, otherwise, it will return `false`.

   C++

   ```
   bool result = parserRefitter->refitFromFile("path_to_onnx_model");
   ```

   Python

   ```
   result = parser_refitter.refit_from_file("path_to_onnx_model")
   ```

4. Call the `refit` function of the `IRefitter` to complete the refit process.

   C++

   ```
   refitSuccess = refitter->refitCudaEngine();
   ```

   Python

   ```
   refit_success = refitter.refit_cuda_engine()
   ```

6.4.3. Weight-Stripping Work with Lean Runtime

Additionally, we can leverage the lean runtime to further reduce the package size for the weight-stripped engine. The lean runtime is the same runtime used in version compatible engines. The original purpose is to allow you to generate a TensorRT engine with version X and load it with an application built with version Y. The lean runtime library is relatively small, approximately 40 MiB. Therefore, software distributors on top of TensorRT only need to ship the weightless engine along with the 40 MiB lean runtime, when the weights are already available on the target customer machine.

The recommended approach to build the engine is as follows:

C++

```
builderConfig.setFlag(BuilderFlag::kVERSION_COMPATIBLE);
builderConfig.setFlag(BuilderFlag::kEXCLUDE_LEAN_RUNTIME);
builderConfig.setFlag(BuilderFlag::kSTRIP_PLAN);
IHostMemory* plan = builder->buildSerializedNetwork(network, config);
```

Python

```
builder_config.set_flag(tensorrt.BuilderFlag.VERSION_COMPATIBLE)
builder_config.set_flag(tensorrt.BuilderFlag.EXCLUDE_LEAN_RUNTIME)
builder_config.set_flag(tensorrt.BuilderFlag.STRIP_PLAN)

plan = builder.build_serialized_network(network, config)
```

Load the engine with the shared lean runtime library path:

C++

```
runtime->loadRuntime("your_lean_runtime_full_path")
```

Python

```
runtime.load_runtime("your_lean_runtime_full_path")
```

For more information about the lean runtime, refer to the Version Compatibility section.

6.4.4. Fine Grained Refit Build

When using the `kREFIT` builder configuration, all weights are marked as refittable. This is useful when it is difficult to distinguish between trainable and untrainable weights. However, marking all weights as refittable can lead to a performance trade-off. This is because certain optimizations are broken when weights are marked as refittable. For example, in the case of the GELU expression, TensorRT can encode all GELU coefficients in a single CUDA kernel. However, if all coefficients are marked as refittable, TensorRT

may no longer be able to fuse the Conv-GELU operations into a single kernel. To address this, we have introduced the fine-grained refit API. This API provides precise control over which weights are marked as refittable, allowing for more efficient optimization.

Here is an example of marking weights as refittable in the `INetworkDefinition`:

C++

```
...
network->setWeightsName(Weights(weights), "conv1_filter"));
network->markWeightsRefittable("conv1_filter"); assert(network-
>areWeightsMarkedRefittable("conv1_filter"));
```

Python

```
...
network.set_weights_name(conv_filter, "conv1_filter")
 network.mark_weights_refittable("conv1_filter")
assert network.are_weights_marked_refittable("conv1_filter")
```

Later, we need update the builder configuration like this:

C++

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

Python

```
...
config.set_flag(trt.BuilderFlag.REFIT_INDIVIDUAL)
builder.build_serialized_network(network, config)
```

The remaining refit code follows the same steps as refitting all weights workflow.

6.4.5. Stripping Weights with Fine Grained Refit Build

The fine grained refit build also works with the weights stripping flag. To run this, we need to enable both builder flags in the code after marking necessary weights as refittable.

Here is an example:

C++

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

Python

```
config.flags |= 1 << int(trt.BuilderFlag.STRIP_PLAN)
config.flags |= 1 << int(trt.BuilderFlag.REFIT_INDIVIDUAL)
builder.build_serialized_network(network, config)
```

The remaining refit code and inference code are the same as the Weight-Stripping section.

6.5. Algorithm Selection and Reproducible Builds

The default behavior of TensorRT's optimizer is to choose the algorithms that globally minimize the execution time of the engine. It does this by timing each implementation,

and sometimes, and when implementations have similar timings, it is possible that system noise will determine which will be chosen on any particular run of the builder. Different implementations will typically use different order of accumulation of floating point values, and two implementations may use different algorithms or even run at different precisions. Thus, different invocations of the builder will typically not result in engines that return bit-identical results.

Sometimes it is important to have a deterministic build, or to recreate the algorithm choices of an earlier build. By providing an implementation of the `IAlgorithmSelector` interface and attaching it to a builder configuration with `setAlgorithmSelector`, you can guide algorithm selection manually.

The method `IAlgorithmSelector::selectAlgorithms` receives an *`AlgorithmContext`* containing information about the algorithm requirements for a layer, and a set of *`Algorithm`* choices meeting those requirements. It returns the set of algorithms which TensorRT should consider for the layer.

The builder selects from these algorithms the one that minimizes the global runtime for the network. If no choice is returned and `BuilderFlag::kREJECT_EMPTY_ALGORITHMS` is unset, TensorRT interprets this to mean that any algorithm may be used for this layer. To override this behavior and generate an error if an empty list is returned, set the `BuilderFlag::kREJECT_EMPTY_ALGORITHMS` flag.

After TensorRT has finished optimizing the network for a given profile, it calls `reportAlgorithms`, which can be used to record the final choice made for each layer.

To build a TensorRT engine deterministically, return a single choice from `selectAlgorithms`. To replay choices from an earlier build, use `reportAlgorithms` to record the choices in that build, and return them in `selectAlgorithms`.

`sampleAlgorithmSelector` demonstrates how to use the algorithm selector to achieve determinism and reproducibility in the builder.

Note:

- The notion of a "layer" in algorithm selection is different from `ILayer` in `INetworkDefinition`. The "layer" in the former can be equivalent to a collection of multiple network layers due to fusion optimizations.
- Picking the fastest algorithm in `selectAlgorithms` may not produce the best performance for the overall network, as it may increase reformatting overhead.
- The timing of an `IAlgorithm` is `0` in `selectAlgorithms` if TensorRT found that layer to be a no-op.
- `reportAlgorithms` does not provide the timing and workspace information for an `IAlgorithm` that are provided to `selectAlgorithms`.

6.6. Creating a Network Definition from Scratch

Instead of using a parser, you can also define the network directly to TensorRT using the Network Definition API. This scenario assumes that the per-layer weights are ready in host memory to pass to TensorRT during the network creation.

The following examples create a simple network with Input, Convolution, Pooling, MatrixMultiply, Shuffle, Activation, and SoftMax layers.

For more information regarding layers, refer to the NVIDIA TensorRT Operator's Reference.

6.6.1. C++

In this example, the weights are loaded into a `weightMap` data structure used in the following code.

First create the builder and network objects. Note that in the following example, the logger is initialized using the `logger.cpp` file common to all C++ samples. The C++ sample helper classes and functions can be found in the `common.h` header file.

```
    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));
```

Refer to the Explicit Versus Implicit Batch section for more information about the `kEXPLICIT_BATCH` flag.

Add the Input layer to the network by specifying the name, datatype, and full dimensions of the input tensor. A network can have multiple inputs, although in this sample there is only one:

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

Add the Convolution layer with hidden layer input nodes, strides, and weights for filter and bias.

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

Note: Weights passed to TensorRT layers are in host memory.

Add the Pooling layer; note that the output from the previous layer is passed as input.

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

Add a Shuffle layer to reshape the input in preparation for a matrix multiplication:

```
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}});
```

Now, add a MatrixMultiply layer. Here, the model exporter provided transposed weights, so the `kTRANSPOSE` option is specified for those.

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

Add the bias, which will broadcast across the batch dimension.

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

Add the ReLU Activation layer:

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

Add the SoftMax layer to calculate the final probabilities:

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

Add a name for the output of the SoftMax layer so that the tensor can be bound to a memory buffer at inference time:

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

Mark it as the output of the entire network:

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

The network representing the MNIST model has now been fully constructed. Refer to sections Building an Engine and Deserializing a Plan for how to build an engine and run inference with this network.

6.6.2. Python

Code corresponding to this section can be found in network_api_pytorch_mnist.

This example uses a helper class to hold some of metadata about the model:

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

In this example, the weights are imported from the PyTorch MNIST model.

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

Create the logger, builder, and network classes.

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

Refer to the Explicit Versus Implicit Batch section for more information about the `kEXPLICIT_BATCH` flag.

Next, create the input tensor for the network, specifying the name, datatype, and shape of the tensor.

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

Add a convolution layer, specifying the inputs, number of output maps, kernel shape, weights, bias, and stride:

```
    conv1_w = weights["conv1.weight"].cpu().numpy()
conv1_b = weights["conv1.bias"].cpu().numpy()
conv1 = network.add_convolution_nd(
    input=input_tensor, num_output_maps=20, kernel_shape=(5, 5), kernel=conv1_w,
bias=conv1_b
)
conv1.stride_nd = (1, 1)
```

Add a pooling layer, specifying the inputs (the output of the previous convolution layer), pooling type, window size, and stride:

```
   pool1 = network.add_pooling_nd(input=conv1.get_output(0), type=trt.PoolingType.MAX,
window_size=(2, 2))
pool1.stride_nd = trt.Dims2(2, 2)
```

Add the next pair of convolution and pooling layers:

```
    conv2_w = weights["conv2.weight"].cpu().numpy()
conv2_b = weights["conv2.bias"].cpu().numpy()
conv2 = network.add_convolution_nd(pool1.get_output(0), 50, (5, 5), conv2_w, conv2_b)
conv2.stride_nd = (1, 1)

pool2 = network.add_pooling_nd(conv2.get_output(0), trt.PoolingType.MAX, (2, 2))
pool2.stride_nd = trt.Dims2(2, 2)
```

Add a Shuffle layer to reshape the input in preparation for a matrix multiplication:

```
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)
```

Now, add a MatrixMultiply layer. Here, the model exporter provided transposed weights, so the `kTRANSPOSE` option is specified for those.

```
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);
```

Add bias, which will broadcast across the batch dimension:

```
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)
```

Add the ReLU activation layer:

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

Add the final fully connected layer, and mark the output of this layer as the output of the entire network:

```
    fc2_w = weights['fc2.weight'].numpy()
    fc2_b = weights['fc2.bias'].numpy()
    fc2 = add_matmul_as_fc(network, 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))
```

The network representing the MNIST model has now been fully constructed. Refer to sections Building an Engine and Performing Inference for how to build an engine and run inference with this network.

6.7. Strongly Typed Networks

By default, TensorRT autotunes tensor types to generate the fastest engine. This can result in accuracy loss when model accuracy requires a layer to run in a higher precision than TensorRT chooses. One approach is to use the `ILayer::setPrecision` and `ILayer::setOutputType` APIs to control a layer's I/O types and hence its execution precision. This approach works but it can be challenging to figure out which layers need to be run at high precision to get the best accuracy.

An alternative approach is to specify low precision use in the model itself, using for example, Automatic mixed precision training or Quantization aware training, and have TensorRT adhere to the precision specifications. TensorRT will still autotune over different data layouts to find an optimal set of kernels for the network.

When you specify to TensorRT that a network is strongly typed, it infers a type for each intermediate and output tensor using the rules in the operator type specification. Inferred types are adhered to while building the engine. As types are not autotuned, an engine built from a strongly typed network can be slower than one where TensorRT chooses tensor types. On the other hand, the build time may improve, as fewer kernel alternatives are evaluated.

Strongly typed networks are not supported with DLA.

You can create a strongly typed network as follows:

C++

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

Python

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

For strongly typed networks, the layer APIs `setPrecision` and `setOutputType` are not permitted, nor are the builder precision flags `kFP16`, `kBF16`, `kFP8`, and `kINT8`. The builder flag `kTF32` is permitted as it controls TF32 Tensor Core usage for FP32 types, rather than controlling use of TF32 data types.

6.8. Reduced Precision in Weakly Typed Networks

6.8.1. Network-Level Control of Precision

By default, TensorRT works in 32-bit precision, but can also execute operations using 16-bit floating point, and 8-bit quantized floating point. Using lower precision requires less memory and enables faster computation.

Reduced precision support depends on your hardware (refer to Hardware and Precision). You can query the builder to check the supported precision support on a platform:

C++

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

Python

```
if builder.platform_has_fp16:
```

Setting flags in the builder configuration informs TensorRT that it may select lower-precision implementations:

C++

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

Python

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

There are three precision flags: FP16, INT8, and TF32, and they may be enabled independently. Note that TensorRT will still choose a higher-precision kernel if it results in overall lower runtime, or if no low-precision implementation exists.

When TensorRT chooses a precision for a layer, it automatically converts weights as necessary to run the layer.

While using FP16 and TF32 precisions is relatively straightforward, there is additional complexity when working with INT8. Refer to the Working with Quantized Types chapter for more details.

Note that even if the precision flags are enabled, the input/output bindings of the engine defaults to FP32. Refer to the I/O Formats section about how to set the data types and formats of the input/output bindings.

6.8.2. Layer-Level Control of Precision

The builder flags provide permissive, coarse-grained control. However, sometimes part of a network requires higher dynamic range or is sensitive to numerical precision. You can constrain the input and output types per layer:

C++

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

Python

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

This provides a *preferred type* (here, `DataType::kFP16`) for the inputs and outputs.

You may further set preferred types for the layer's outputs:

C++

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

Python

```
layer.set_output_type(out_tensor_index, trt.fp32)
```

The computation will use the same floating-point type as is preferred for the inputs. Most TensorRT implementations have the same floating-point types for input and output; however, Convolution, Deconvolution, and FullyConnected can support quantized INT8 input and unquantized FP16 or FP32 output, as sometimes working with higher-precision outputs from quantized inputs is necessary to preserve accuracy.

Setting the precision constraint hints to TensorRT that it should select a layer implementation whose inputs and outputs match the preferred types, inserting reformat operations if the outputs of the previous layer and the inputs to the next layer do not match the requested types. Note that TensorRT will only be able to select an implementation with these types if they are also enabled using the flags in the builder configuration.

By default, TensorRT chooses such an implementation only if it results in a higher-performance network. If another implementation is faster, TensorRT uses it and issues a warning. You can override this behavior by preferring the type constraints in the builder configuration.

C++

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

Python

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

If the constraints are preferred, TensorRT obeys them unless there is no implementation with the preferred precision constraints, in which case it issues a warning and uses the fastest available implementation.

To change the warning to an error, use `OBEY` instead of `PREFER`:

C++

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

Python

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

sampleINT8API illustrates the use of reduced precision with these APIs.

Precision constraints are optional - you can query to determine whether a constraint has been set using `layer->precisionIsSet()` in C++ or `layer.precision_is_set` in Python. If a precision constraint is not set, then the result returned from `layer->getPrecision()` in C++, or reading the `precision` attribute in Python, is not meaningful. Output type constraints are similarly optional.

If no constraints are set using `ILayer::setPrecision` or `ILayer::setOutputType` API, then `BuilderFlag::kPREFER_PRECISION_CONSTRAINTS` or `BuilderFlag::kOBEY_PRECISION_CONSTRAINTS` are ignored. A layer is free to choose from any precision or output types based on allowed builder precisions.

Note that the `ITensor::setType()` API does not set the precision constraint of a tensor, unless it is one of the input/output tensors of the network. Also, there is a distinction between `layer->setOutputType()` and `layer->getOutput(i)->setType()`. The former is an optional type that constrains the implementation that TensorRT will choose for a layer. The latter specifies the type of a network's input/output and is ignored if the tensor is not a network input/output. If they are different, TensorRT will insert a cast to ensure that both specifications are respected. Thus if you are calling `setOutputType()` for a layer that produces a network output, you should in general also configure the corresponding network output to have the same type.

6.8.3. TF32

TensorRT allows the use of TF32 Tensor Cores by default. When computing inner products, such as during convolution or matrix multiplication, TF32 execution does the following:

- Rounds the FP32 multiplicands to FP16 precision but keeps the FP32 dynamic range.
- Computes an exact product of the rounded multiplicands.
- Accumulates the products in an FP32 sum.

TF32 Tensor Cores can speed up networks using FP32, typically with no loss of accuracy. It is more robust than FP16 for models that require an HDR (high dynamic range) for weights or activations.

There is no guarantee that TF32 Tensor Cores are actually used, and there is no way to force the implementation to use them - TensorRT can fall back to FP32 at any time and always falls back if the platform does not support TF32. However you can disable their use by clearing the TF32 builder flag.

C++

```
config->clearFlag(BuilderFlag::kTF32);
```

Python

```
config.clear_flag(trt.BuilderFlag.TF32)
```

Setting the environment variable `NVIDIA_TF32_OVERRIDE=0` when building an engine disables the use of TF32, despite setting `BuilderFlag::kTF32`. This environment variable, when set to `0`, overrides any defaults or programmatic configuration of NVIDIA libraries, so they never accelerate FP32 computations with TF32 Tensor Cores. This is meant to be a debugging tool only, and no code outside NVIDIA libraries should change the behavior based on this environment variable. Any other setting besides `0` is reserved for future use.

WARNING: Setting the environment variable `NVIDIA_TF32_OVERRIDE` to a different value when the engine is run can cause unpredictable precision/performance effects. It is best left unset when an engine is run.

Note: Unless your application requires the higher dynamic range provided by TF32, FP16 will be a better solution since it almost always yields faster performance.

6.8.4. BF16

TensorRT supports the `bfloat16` (brain float) floating point format on NVIDIA Ampere and later architectures. Like other precisions, it can be enabled using the corresponding builder flag:

C++

```
config->setFlag(BuilderFlag::kBF16);
```

Python

```
config.set_flag(trt.BuilderFlag.BF16)
```

Note that not all layers support `bfloat16`. Refer to the TensorRT Operator documentation for details.

6.9. Control of Computational Precision

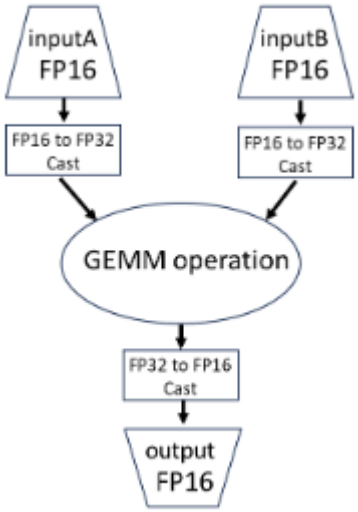
Sometimes, in addition to setting the input and output precisions for an operator, it is desirable to control the internal precision of the computation. By default, TensorRT selects the computational precision based on the layer input type and global performance considerations.

There are two layers where TensorRT provides additional capabilities to control computational precision:

The `INormalizationLayer` provides a `setPrecision` method to control precision of accumulation. By default, to avoid overflow errors, TensorRT accumulates in FP32, even in mixed precision mode regardless of builder flags. You can use this method to specify FP16 accumulation instead.

For the `IMatrixMultiplyLayer`, TensorRT by default selects accumulation precision based on the input types and performance considerations, although the accumulation type is guaranteed to have a range at least as great as the input types. When using strongly-typed mode, you can enforce the use of FP32 precision for FP16 GEMMs by casting the inputs to FP32. TensorRT recognizes this specific pattern, and fuses the casts with the GEMM, resulting in a single kernel with FP16 inputs and FP32 accumulation.

Figure 1. Creating a Graph for FP32 Accumulation Request



6.10. I/O Formats

TensorRT optimizes a network using many different data formats. In order to allow efficient passing of data between TensorRT and a client application, these underlying data formats are exposed at network I/O boundaries, that is, for Tensors marked as network input or output, and when passing data to and from plugins. For other tensors, TensorRT picks formats that result in the fastest overall execution, and may insert reformats to improve performance.

You can assemble an optimal data pipeline by profiling the available I/O formats in combination with the formats most efficient for the operations preceding and following TensorRT.

To specify I/O formats, you specify one or more formats in the form of a bitmask.

The following example sets the input tensor format to `TensorFormat::kHWC8`. Note that this format only works for `DataType::kHALF`, so the data type must be set accordingly.

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
```

Note that calling `setAllowedFormats()` or `setType()` on a tensor that is not a network input/output, has no effect and is ignored by TensorRT.

It is possible to make TensorRT avoid inserting reformatting at the network boundaries, by setting the builder configuration flag `DIRECT_IO`. This flag is generally counter-productive for two reasons:

- The resulting engine might be slower than if TensorRT had been allowed to insert reformatting. Reformatting may sound like wasted work, but it can allow coupling of the most efficient kernels.
- The build will fail if TensorRT cannot build an engine without introducing such reformatting. The failure may happen only for some target platforms, because of what formats are supported by kernels for those platforms.

The flag exists for the sake of users who want full control over whether reformatting happens at I/O boundaries, such as to build engines that run solely on DLA without falling back to the GPU for reformatting.

sampleIOFormats illustrates how to specify I/O formats using C++.

The following table shows the supported formats.

Table 1. Supported I/O Formats

Format	kINT32	kFLOAT	kHALF	kINT8	kBOOL	kUINT8	kINT64
kLINEAR	Only for GPU	Supported	Supported	Supported	Supported	Supported	Supported
kCHW2	Not Applicable	Not Applicable	Only for GPU	Not Applicable	Not Applicable	Not Applicable	Not Applicable
kCHW4	Not Applicable	Not Applicable	Supported	Supported	Not Applicable	Not Applicable	Not Applicable
kHWC8	Not Applicable	Not Applicable	Only for GPU	Not Applicable	Not Applicable	Not Applicable	Not Applicable
kCHW16	Not Applicable	Not Applicable	Supported	Not Applicable	Not Applicable	Not Applicable	Not Applicable
kCHW32	Not Applicable	Only for GPU	Only for GPU	Supported	Not Applicable	Not Applicable	Not Applicable
kDHWC8	Not Applicable	Not Applicable	Only for GPU	Not Applicable	Not Applicable	Not Applicable	Not Applicable
kCDHW32	Not Applicable	Not Applicable	Only for GPU	Only for GPU	Not Applicable	Not Applicable	Not Applicable
kHWC	Not Applicable	Only for GPU	Not Applicable	Not Applicable	Not Applicable	Supported	Supported
kDLA_LINEAR	Not Applicable	Not Applicable	Only for DLA	Only for DLA	Not Applicable	Not Applicable	Not Applicable
kDLA_HWC4	Not Applicable	Not Applicable	Only for DLA	Only for DLA	Not Applicable	Not Applicable	Not Applicable
kHWC16	Not Applicable	Not Applicable	Only for NVIDIA Ampere Architecture GPUs and later	Not Applicable	Not Applicable	Not Applicable	Not Applicable
kDHWC	Not Applicable	Only for GPU	Not Applicable	Not Applicable	Not Applicable	Not Applicable	Not Applicable

Note that for the vectorized formats, the channel dimension must be zero-padded to the multiple of the vector size. For example, if an input binding has dimensions of `[16,3,224,224]`, `kHALF` data type, and `kHWC8` format, then the actual-required size of the binding buffer would be `16*8*224*224*sizeof(half)` bytes, even though the `engine->getBindingDimension()` API will return tensor dimensions as `[16,3,224,224]`. The values in the padded part (that is, where `C=3,4,…,7` in this example) must be filled with zeros.

Refer to Data Format Descriptions for how the data are actually laid out in memory for these formats.

6.11. Explicit Versus Implicit Batch

TensorRT supports two modes for specifying a network: explicit batch and implicit batch.

In *implicit batch* mode, every tensor has an implicit batch dimension and all other dimensions must have constant length. This mode was used by early versions of TensorRT, and is now deprecated but continues to be supported for backwards compatibility.

In *explicit batch* mode, all dimensions are explicit and can be dynamic, that is their length can change at execution time. Many new features, such as dynamic shapes and loops, are available only in this mode. It is also required by the ONNX parser.

For example, consider a network that processes N images of size HxW with 3 channels, in NCHW format. At runtime, the input tensor has dimensions [N,3,H,W]. The two modes differ in how the `INetworkDefinition` specifies the tensor's dimensions:

- In explicit batch mode, the network specifies [N,3,H,W].
- In implicit batch mode, the network specifies only [3,H,W]. The batch dimension N is implicit.

Operations that "talk across a batch" are impossible to express in implicit batch mode because there is no way to specify the batch dimension in the network. Examples of inexpressible operations in implicit batch mode:

- reducing across the batch dimension
- reshaping the batch dimension
- transposing the batch dimension with another dimension

The exception is that a tensor can be *broadcast* across the entire batch, through the `ITensor::setBroadcastAcrossBatch` method for network inputs, and implicit broadcasting for other tensors.

Explicit batch mode erases the limitations - the batch axis is axis 0. A more accurate term for explicit batch would be "batch oblivious," because in this mode, TensorRT attaches no special semantic meaning to the leading axis, except as required by specific operations. Indeed in explicit batch mode there might not even be a batch dimension (such as a network that handles only a single image) or there might be multiple batch dimensions of unrelated lengths (such as comparison of all possible pairs drawn from two batches).

The choice of explicit versus implicit batch must be specified when creating the `INetworkDefinition`, using a flag. Here is the C++ code for explicit batch mode:

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

For implicit batch, use `createNetwork` or pass a 0 to `createNetworkV2`.

Here is the Python code for explicit batch mode:

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

For implicit batch, omit the argument or pass a 0.

6.12. Sparsity

NVIDIA Ampere Architecture GPUs support Structured Sparsity. To make use of this feature to achieve higher inference performance, the weights must have at least 2 zeros in every four-entry vector. For TensorRT, the requirements are:

- For Convolution, for each output channel and for each spatial pixel in the kernel weights, every four input channels must have at least two zeros. In other words, assuming that the kernel weights have the shape `[K, C, R, S]` and `C % 4 == 0`, then the requirement is verified using the following algorithm:

```
hasSparseWeights = True
for k in range(0, K):
    for r in range(0, R):
        for s in range(0, S):
            for c_packed in range(0, C // 4):
                if numpy.count_nonzero(weights[k, c_packed*4:(c_packed+1)*4, r, s]) > 2 :
                    hasSparseWeights = False
```

- For MatrixMultiply of which an input is produced by Constant, every four elements of the reduction axis (`K`) must have at least two zeros.

Polygraphy (`polygraphy inspect sparsity`) can be used to detect whether the operation weights in an ONNX model follow the 2:4 structured sparsity pattern.

To enable the sparsity feature, set the `kSPARSE_WEIGHTS` flag in the builder config and make sure that kFP16 or kINT8 modes are enabled. For example:

C++

```
config->setFlag(BuilderFlag::kSPARSE_WEIGHTS);
config->setFlag(BuilderFlag::kFP16);
config->setFlag(BuilderFlag::kINT8);
```

Python

```
config.set_flag(trt.BuilderFlag.SPARSE_WEIGHTS)
config.set_flag(trt.BuilderFlag.FP16)
config.set_flag(trt.BuilderFlag.INT8)
```

At the end of the TensorRT logs when the TensorRT engine is built, TensorRT reports which layers contain weights that meet the structured sparsity requirement, and in which layers TensorRT selects tactics that make use of the structured sparsity. In some cases, tactics with structured sparsity can be slower than normal tactics and TensorRT will choose normal tactics in these cases. The following output shows an example of TensorRT logs showing information about sparsity:

```
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) Found 3 layer(s) eligible to use sparse tactics:
 conv1, conv2, conv3
[03/23/2021-00:14:05] [I] [TRT] (Sparsity) Chose 2 layer(s) using sparse tactics: conv2,
 conv3
```

Forcing kernel weights to have structured sparsity patterns can lead to accuracy loss. To recover lost accuracy with further fine-tuning, refer to the Automatic Sparsity tool in PyTorch.

To measure inference performance with structured sparsity using `trtexec`, refer to the trtexec section.

6.13. Empty Tensors

TensorRT supports empty tensors. A tensor is an empty tensor if it has one or more dimensions with length zero. Zero-length dimensions usually get no special treatment. If a rule works for a dimension of length L for an arbitrary positive value of L, it usually works for L=0 too.

For example, when concatenating two tensors with dimensions [x,y,z] and [x,y,w] along the last axis, the result has dimensions [x,y,z+w], regardless of whether x, y, z, or w is zero.

Implicit broadcast rules remain unchanged since only unit-length dimensions are special for broadcast. For example, given two tensors with dimensions [1,y,z] and [x,1,z], their sum computed by `IElementWiseLayer` has dimensions [x,y,z], regardless of whether x, y, or z is zero.

If an engine binding is an empty tensor, it still needs a non-null memory address, and different tensors should have different addresses. This is consistent with the C++ rule that every object has a unique address, for example, `new float[0]` returns a non-null pointer. If using a memory allocator that might return a null pointer for zero bytes, ask for at least one byte instead.

Refer to the NVIDIA TensorRT Operator's Reference for any per-layer special handling of empty tensors.

6.14. Reusing Input Buffers

TensorRT allows specifying a CUDA event to be signaled once the input buffers are free to be reused. This allows the application to immediately start refilling the input buffer region for the next inference in parallel with finishing the current inference. For example:

C++

```
context->setInputConsumedEvent(&inputReady);
```

Python

```
context.set_input_consumed_event(inputReady)
```

6.15. Engine Inspector

TensorRT provides the `IEngineInspector` API to inspect the information inside a TensorRT engine. Call the `createEngineInspector()` from a deserialized engine to create an engine inspector, and then call `getLayerInformation()` or `getEngineInformation()` inspector APIs to get the information of a specific layer in the engine or the entire engine, respectively. You can print out the information of the first layer of the given engine, as well as the overall information of the engine, as follows:

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.
```

Note that the level of detail in the engine/layer information depends on the `ProfilingVerbosity` builder config setting when the engine is built. By default, `ProfilingVerbosity` is set to `kLAYER_NAMES_ONLY`, so only the layer names will be printed. If `ProfilingVerbosity` is set to `kNONE`, then no information will be printed; if it is set to `kDETAILED`, then detailed information will be printed.

Below are some examples of layer information printed by `getLayerInformation()` API depending on the `ProfilingVerbosity` setting:

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_vect_c_32_nchw_vect_c_32_ti
  "TacticValue": "0x11bde0e1d9f2f35d"
}
```

In addition, when the engine is built with dynamic shapes, the dynamic dimensions in the engine information will be shown as `-1` and the tensor format information will not be shown because these fields depend on the actual shape at inference

phase. To get the engine information for a specific inference shape, create an `IExecutionContext`, set all the input dimensions to the desired shapes, and then call `inspector->setExecutionContext(context)`. After the context is set, the inspector will print the engine information for the specific shape set in the context.

The `trtexec` tool provides the `--profilingVerbosity`, `--dumpLayerInfo`, and `--exportLayerInfo` flags that can be used to get the engine information of a given engine. Refer to the trtexec section for more details.

Currently, only binding information and layer information, including the dimensions of the intermediate tensors, precisions, formats, tactic indices, layer types, and layer parameters, are included in the engine information. More information may be added into the engine inspector output as new keys in the output JSON object in future TensorRT versions. More specifications about the keys and the fields in the inspector output will also be provided.

In addition, some subgraphs are handled by a next-generation graph optimizer that is not yet integrated with the engine inspector. Therefore, the layer information within these layers is not currently shown. This will be improved in a future TensorRT version.

6.16. Optimizer Callbacks

The optimizer callback API feature allows you to monitor the progress of the TensorRT build process, for example to provide user feedback in interactive applications. To enable progress monitoring, create an object that implements the `IProgressMonitor` interface, then attach it to the `IBuilderConfig`, for example:

C++

```
builderConfig->setProgressMonitor(&monitor);
```

Python

```
context.set_progress_monitor(monitor)
```

Optimization is divided into hierarchically nested phases, each consisting of a number of steps. At the start of each phase, the `phaseStart()` method of `IProgressMonitor` is called, telling you the phase name and how many steps it has. The `stepComplete()` function is called when each step completes, and `phaseFinish()` is called when the phase finishes.

Returning false from `stepComplete()` cleanly forces the build to terminate early.

6.17. Preview Features

The preview feature API is an extension of `IBuilderConfig` to allow the gradual introduction of new features to TensorRT. Selected new features are exposed under this API, allowing you to opt in or opt out. A preview feature remains in preview status for one or two TensorRT release cycles, and is then either integrated as a mainstream feature, or dropped. When a preview feature is fully integrated into TensorRT, it is no longer controllable through the preview API.

Preview features are defined using a 32-bit `PreviewFeature` enumeration. Feature identifiers are a concatenation of the feature name and the TensorRT version.

```
<FEATURE_NAME>_XXYY
```

Where `XX` and `YY` are the TensorRT major and minor versions, respectively, of the TensorRT release which first introduced the feature. The major and minor versions are specified using two digits with leading-zero padding when necessary.

If the semantics of a preview feature change from one TensorRT release to another, the older preview feature is deprecated and the revised feature is assigned a new enumeration value and name.

Deprecated preview features are marked in accordance with the deprecation policy.

For more information about the C++ API, refer to `nvinfer1::PreviewFeature`, `IBuilderConfig::setPreviewFeature`, and `IBuilderConfig::getPreviewFeature`.

The Python API has similar semantics using the `PreviewFeature` enum and `set_preview_feature`, and `get_preview_feature` functions.

6.18. Debug Tensors

The debug tensor feature allows you to inspect intermediate tensors as the network executes. There are a few key differences between using debug tensors and marking all required tensors as outputs:

1. Marking all tensors as outputs requires you to provide memory to store tensors in advance, while debug tensors can be turned off during runtime if unneeded.
2. When debug tensors are turned off, the performance impact on execution of the network is minimized.
3. For a debug tensor in a loop, values are emitted every time it is written.

To enable this feature, perform the following steps:

1. Mark the target tensors before the network is compiled.

   C++

   ```
   networkDefinition->markDebug(&tensor);
   ```

   Python

   ```
   network.mark_debug(tensor)
   ```

2. Define a `DebugListener` class deriving from `IDebugListener`, and implement the virtual function for processing the tensor.

   C++

   ```
   virtual void processDebugTensor(
                                   void const* addr,
                                   TensorLocation location,
                                   DataType type,
                                   Dims const& shape,
                                   char const* name,
                                   cudaStream_t stream) = 0;
   ```

   Python

   ```
   process_debug_tensor(self, addr, location, type, shape, name, stream)
   ```

When the function is invoked during execution, the debug tensor is passed via the parameters:

```
location: TensorLocation of the tensor
addr: pointer to buffer
type: data Type of the tensor
shape: shape of the tensor
name: name of the tensor
stream: Cuda stream object
```

The data will be in linear format.

3. Attach your listener to `IExecutionContext`.

   C++

   ```
   executionContext->setDebugListener(&debugListener);
   ```

   Python

   ```
   execution_context.set_debug_listener(debugListener)
   ```

   Because the function is executed as part of `enqueue()`, you must use the stream to synchronize reading of the data by, for example, invoking a device function on the stream to process or copy the data.

4. Set the debug state for the tensors of interest to on before execution of the engine.

   C++

   ```
   executionContext->setDebugState(tensorName, flag);
   ```

   Python

   ```
   execution_context.set_debug_state(tensorName, flag)
   ```

6.19. Weight Streaming

The weight streaming feature allows you to offload some weights from device memory to host memory. During network execution, these weights are streamed from the host to the device as needed. This technique can free up device memory, enabling you to run larger models or process larger batch sizes.

To enable this feature, during engine building, create network with `kSTRONGLY_TYPED` and set `kWEIGHT_STREAMING` to builder config:

C++

```
...
builder->createNetworkV2(1U <<
 static_cast<uint32_t>(NetworkDefinitionCreationFlag::kSTRONGLY_TYPED));
config->setFlag(BuilderFlag::kWEIGHT_STREAMING);
```

Python

```
builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.STRONGLY_TYPED))
config.set_flag(trt.BuilderFlag.WEIGHT_STREAMING)
```

During runtime, deserialization allocates a host buffer to store all the weights instead of uploading them directly to the device. This can increase the peak memory usage of the host. You can use `IStreamReader` to deserialize directly from the engine file, avoiding the need for a temporary buffer, which helps reduce the peak memory usage.

After deserializing the engine, set the device memory budget for weights by:

C++

```
...
engine->setWeightStreamingBudgetV2(size)
```

Python

```
...
engine.weight_streaming_budget_v2 = size
```

The following APIs can help to determine the budget:

- `getStreamableWeightsSize()` returns the total size of streamable weights.
- `getWeightStreamingScratchMemorySize()` returns the extra scratch memory size for a context when weight streaming is enabled.
- `getDeviceMemorySizeV2()` returns the total scratch memory size required by a context. If this API is called before enabling weight streaming by `setWeightStreamingBudgetV2()`, the return value will not include the extra scratch memory size required by weight streaming, which can be obtained using `getWeightStreamingScratchMemorySize()`. Otherwise, it will include this extra memory.

Additionally, you can combine information about the current free device memory size, context number, and other allocation needs.

TensorRT can also automatically determine a memory budget by `getWeightStreamingAutomaticBudget()`. However, due to limited information about the user's specific memory allocation requirements, this automatically determined budget may be suboptimal and could potentially lead to out-of-memory errors.

If the budget set by `setWeightStreamingBudgetV2` is larger than the total size of streamable weights obtained by `getStreamableWeightsSize()`, the budget will be clipped to the total size, effectively disabling weight streaming.

You can query the budget setted by `getWeightStreamingBudgetV2()`.

The budget can be adjusted by set again when there is no active context of the engine.

After setting the budget, TensorRT will automatically determine which weights to retain on the device memory to maximize the overlap between computation and weights fetching.

Chapter 7. Working with Quantized Types

7.1. Introduction to Quantization

TensorRT supports the use of low precision types to represent quantized floating point values. The quantization scheme is *symmetric* quantization - quantized values are represented in signed INT8, FP8E4M3 (FP8 for short), or signed INT4, and the transformation from quantized to unquantized values is simply a multiplication. In the reverse direction, quantization uses the reciprocal scale, followed by clamping and rounding (for integers) or casting (for FP8).

TensorRT quantizes activations as well as weights to INT8 and FP8. For INT4, weight-only-quantization is supported.

7.1.1. Quantization Workflows

There are two workflows for creating quantized networks:

Post-training quantization (PTQ) derives scale factors after the network has been trained. TensorRT provides a workflow for PTQ, called *calibration*, where it measures the distribution of activations within each activation tensor as the network executes on representative input data, and then uses that distribution to estimate scale values for each tensor.

Quantization-aware training (QAT) computes the scale factors during training, using a technique called fake-quantization which simulates the quantization and dequantization process. This allows the training process to compensate for the effects of the quantization and dequantization operations.

TensorRT's Quantization Toolkit is a PyTorch library that helps produce QAT models that can be optimized by TensorRT. You can also use the toolkit's PTQ recipe to perform PTQ in PyTorch and export to ONNX.

7.1.2. Explicit Versus Implicit Quantization

Quantized networks can be processed in two (mutually exclusive) ways: using either implicit quantization or explicit quantization. The main difference between the two

processing modes is whether you require explicit control over quantization, or instead let the TensorRT builder choose which operations and tensors to quantize (implicit). The sections below provide more details. Implicit quantization is only supported when quantizing for INT8 and cannot be used together with strong typing (because types are not auto-tuned and the only method to convert activations to and from INT8 is via Q/DQ operators).

TensorRT uses explicit quantization mode when a network has `QuantizeLayer` and `DequantizeLayer` layers. TensorRT uses implicit quantization mode when there are no `QuantizeLayer` or `DequantizeLayer` layers in the network and INT8 is enabled in the builder configuration. Only INT8 is supported in implicit quantization mode.

In *implicitly quantized* networks, each activation tensor that is a candidate for quantization has an associated scale that is deduced by a calibration process or assigned by the API function `setDynamicRange`. TensorRT will use this scale if it decides to quantize the tensor.

When processing implicitly quantized networks, TensorRT treats the model as a floating-point model when applying the graph optimizations, and uses INT8 opportunistically to optimize layer execution time. If a layer runs faster in INT8 and has assigned quantization scales on its data inputs and outputs, then a kernel with INT8 precision is assigned to that layer. Otherwise, a high-precision floating-point (that is, FP32, FP16 or BF16) kernel is assigned. Where a high precision floating point is required for accuracy at the expense of performance, this can be specified using the APIs `Layer::setOutputType` and `Layer::setPrecision`.

In *explicitly quantized* networks, the quantization and dequantization operations are represented explicitly by `IQuantizeLayer` (C++, Python) and `IDequantizeLayer` (C++, Python) nodes in the graph - these will henceforth be referred to as Q/DQ nodes. By contrast with implicit quantization, the explicit form specifies exactly where conversion to and from a quantized type is performed, and the optimizer will perform only conversions to and from quantized types that are dictated by the semantics of the model, even if:

- Adding extra conversions could increase layer precision (for example, choosing an FP16 kernel implementation over a quantized type implementation).
- Adding or removing conversions results in an engine that executes faster (for example, choosing a quantized type kernel implementation to execute a layer specified as having high-precision, or vice versa).

ONNX uses an explicitly quantized representation: when a model in PyTorch or TensorFlow is exported to ONNX, each fake-quantization operation in the framework's graph is exported as Q followed by DQ. Since TensorRT preserves the semantics of these layers, users can expect accuracy very close to that seen in the framework. While optimizations preserve the arithmetic semantics of quantization and dequantization operators, they may change the order of floating-point operations in the model, so results will not be bitwise identical.

TensorRT's PTQ capability generates a calibration cache that is used with implicit quantization.

By contrast, performing either QAT or PTQ in a deep learning framework and then exporting to ONNX will result in an explicitly quantized model.

Table 2. Implicit Vs Explicit Quantization

	Implicit Quantization	Explicit Quantization
Supported quantized data-types	INT8	INT8, FP8, INT4
User control over precision	Global builder flags and per-layer precision APIs.	Encoded directly in the model.
API	▶ Model + Scales (dynamic range API) ▶ Model + Calibration data	Model with Q/DQ layers.
Quantization scales	Weights: ▶ Set by TensorRT (internal) ▶ Per-channel quantization ▶ INT8 range `[-127, 127]` Activations: ▶ Set by calibration or specified by the user ▶ Per-tensor quantization ▶ INT8 range `[-128, 127]`	Weights and activations: ▶ Specified using Q/DQ ONNX operators ▶ INT8 range `[-128, 127]` ▶ FP8 range: `[-448, 448]` ▶ INT4 range: `[-8, 7]` Activations use per-tensor quantization. Weights use either per-tensor quantization, per-channel quantization or block quantization.

For more background on quantization, refer to the following papers:

- Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation
- FP8 Formats for Deep Learning

7.1.3. Quantization Schemes

INT8

Given scale s, we can represent quantization and dequantization operations as follows:

$x_q = \text{quantize}(x, s) = \text{roundWithTiesToEven}(\text{clip}(\frac{x}{s}, -128, 127))$ where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized INT8 value in range [-128,127]. Refer to Explicit Versus Implicit Quantization for more information.
- roundWithTiesToEven is described here.

$x = \text{dequantize}(x_q, s) = x_q * s$

In explicit quantization, you are responsible for choosing all scales. In implicit quantization mode the activation scale is configured by you or determined using one of TensorRT's calibration algorithms (refer to Post-Training Quantization Using Calibration). The weight scale is computed by TensorRT according to the following formula:

$s = \frac{\max(\text{abs}(x_{min}^{ch}), \text{abs}(x_{max}^{ch}))}{127}$ where x_{min}^{ch} and x_{max}^{ch} are floating point minimum and maximum values for ch of the weights tensor.

Using FP8 and INT8 in the same network is not allowed.

FP8

When using FP8 only explicit quantization is supported, and therefore you are responsible for the values of the quantization scales.

$x_q = \text{quantize}(x, s) = \text{roundWithTiesToEven}(\text{clip}(\frac{x}{s}, -448, 448))$ where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized E4M3 FP8 value in range [-448, 448].
- s is the quantization scale expressed using a 16-bit or 32-bit floating point.
- roundWithTiesToEven is described here.

$x = \text{dequantize}(x_q, s) = x_q * s$

Using FP8 and INT8 in the same network is not allowed.

INT4

When using INT4 only explicit quantization is supported, and therefore you are responsible for the values of the quantization scales.

$x_q = \text{quantize}(x, s) = \text{roundWithTiesToEven}(\text{clip}(\frac{x}{s}, -8, 7))$ where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized INT4 value in range [-8, 7].
- s is the quantization scale expressed using a 16-bit or 32-bit floating point.
- roundWithTiesToEven is described here.

$x = \text{dequantize}(x_q, s) = x_q * s$

TensorRT only supports INT4 for weight-only quantization (refer to Q/DQ Layer-Placement Recommendations).

7.1.4. Quantization Modes

There are three supported quantization scale granularities:

- *Per-tensor quantization*: in which a single scale value (scalar) is used to scale the entire tensor.
- *Per-channel quantization*: in which a scale tensor is broadcast along the given axis - for convolutional neural networks, this is typically the channel axis.
- *Block quantization*: in which the tensor is divided to fixed-size 1-dimensional blocks along a single dimension. A scale factor is defined for each block.

The quantization scale must consist of all positive high-precision float coefficients (FP32, FP16 or BF16). The rounding method is round-to-nearest ties-to-even and clamps to the valid range, which is `[-128, 127]` for INT8, `[-448, 448]` for FP8, and `[-8, 7]` for INT4.

With explicit quantization, activations can only be quantized using per-tensor quantization. Weights can be quantized in any of the quantization modes.

In implicit quantization, weights are quantized by TensorRT during engine optimization and only per-channel quantization is used. TensorRT quantizes weights for convolution, deconvolution, fully connected layers, and MatMul, where the second input is constant and both input matrices are 2D.

When using per-channel quantization with Convolutions, the axis of quantization must be the output-channel axis. For example, when the weights of 2D convolution are described using `KCRS` notation, `K` is the output-channel axis, and the weights quantization can be described as:

```
For each k in K:
    For each c in C:
        For each r in R:
            For each s in S:
                output[k,c,r,s] := clamp(round(input[k,c,r,s] / scale[k]))
```

The scale is a vector of coefficients and must have the same size as the quantization axis.

Dequantization is performed similarly except for the pointwise operation that is defined as:

```
output[k,c,r,s] := input[k,c,r,s] * scale[k]
```

Block Quantization

In block quantization, elements are grouped into 1-D blocks with all of the elements in a block sharing a common scale factor. Block quantization is supported for only 2-D weight-only-quantization (WoQ) with INT4.

When using block quantization, the scale tensor dimensions are equal to the data tensor dimensions except for one dimension over which blocking is performed (the blocking axis). For example, given a 2-D RS weights input, R (dimension 0) as the blocking axis and B as the block size, the scale in the blocking axis is repeated according to the block size, and can be described like this:

```
For each r in R:
    For each s in S:
        output[r,s] = clamp(round(input[r,s] / scale[r//B, s]))
```

The scale in this case is a 2D array of coefficients, with dimensions (R//B, S).

Dequantization is performed similarly, except for the pointwise operation that is defined as:

```
output[r,s] = input[r,s] * scale[r//B, s]
```

7.2. Setting Dynamic Range

The dynamic range API is only applicable to INT8 quantization.

TensorRT provides APIs to set *dynamic range* (the range that must be represented by the quantized tensor) directly, to support implicit quantization where these values have been calculated outside TensorRT.

The API allows setting the dynamic range for a tensor using minimum and maximum values. Since TensorRT currently supports only symmetric range, the scale is calculated using `max(abs(min_float), abs(max_float))`. Note that when `abs(min_float) != abs(max_float)`, TensorRT uses a larger dynamic-range than configured, which may increase the rounding error.

You can set the dynamic range for a tensor as follows:

C++

```
tensor->setDynamicRange(min_float, max_float);
```

Python

```
tensor.dynamic_range = (min_float, max_float)
```

sampleINT8API illustrates the use of these APIs in C++.

7.3. Post-Training Quantization Using Calibration

Calibration is only applicable to INT8 quantization.

In post-training quantization, TensorRT computes a scale value for each tensor in the network. This process, called *calibration,* requires you to supply representative input data on which TensorRT runs the network to collect statistics for each activation tensor.

The amount of input data required is application-dependent, but experiments indicate that about 500 images are sufficient for calibrating ImageNet classification networks.

Given the statistics for an activation tensor, deciding on the best scale value is not an exact science - it requires balancing two sources of error in the quantized representation: *discretization error* (which increases as the range represented by each quantized value becomes larger) and *truncation error* (where values are clamped to the limits of the representable range.) Thus, TensorRT provides multiple different calibrators that calculate the scale in different ways. Older calibrators also performed layer fusion for GPU to optimize away unneeded Tensors before performing calibration. This can be problematic when using DLA, where fusion patterns may be different, and can be overridden using the `kCALIBRATE_BEFORE_FUSION` quantization flag.

Calibration batch size can also affect the *truncation error* for `IInt8EntropyCalibrator2` and `IInt8EntropyCalibrator`. For example, calibrating using multiple small batches of

calibration data may result in reduced histogram resolution and poor scale value. For each calibration step, TensorRT updates the histogram distribution for each activation tensor. If it encounters a value in the activation tensor, larger than the current histogram max, the histogram range is increased by a power of two to accommodate the new maximum value. This approach works well unless histogram reallocation occurs in the last calibration step, resulting in a final histogram with half the bins empty. Such a histogram can produce poor calibration scales. This also makes calibration susceptible to the order of calibration batches, that is, a different order of calibration batches can result in the histogram size being increased at different points, producing slightly different calibration scales. To avoid this issue, calibrate with as large a single batch as possible, and ensure that calibration batches are well randomized and have similar distribution.

`IInt8EntropyCalibrator2`
: Entropy calibration chooses the tensor's scale factor to optimize the quantized tensor's information-theoretic content, and usually suppresses outliers in the distribution. This is the current and recommended entropy calibrator and is required for DLA. Calibration happens before Layer fusion by default. Calibration batch size may impact the final result. It is recommended for CNN-based networks.

`IInt8MinMaxCalibrator`
: This calibrator uses the entire range of the activation distribution to determine the scale factor. It seems to work better for NLP tasks. Calibration happens before Layer fusion by default. This is recommended for networks such as NVIDIA BERT (an optimized version of Google's official implementation).

`IInt8EntropyCalibrator`
: This is the original entropy calibrator. It is less complicated to use than the LegacyCalibrator and typically produces better results. Calibration batch size may impact the final result. Calibration happens after Layer fusion by default.

`IInt8LegacyCalibrator`
: This calibrator is for compatibility with TensorRT 2.0 EA. This calibrator requires user parameterization and is provided as a fallback option if the other calibrators yield poor results. Calibration happens after Layer fusion by default. You can customize this calibrator to implement percentile max, for example, 99.99% percentile max is observed to have best accuracy for NVIDIA BERT and NeMo ASR model QuartzNet.

When building an INT8 engine, the builder performs the following steps:

1. Build a 32-bit engine, run it on the calibration set, and record a histogram for each tensor of the distribution of activation values.
2. Build from the histograms a calibration table providing a scale value for each tensor.
3. Build the INT8 engine from the calibration table and the network definition.

Calibration can be slow; therefore the output of step 2 (the calibration table) can be cached and reused. This is useful when building the same network multiple times on a given platform and is supported by all calibrators.

Before running calibration, TensorRT queries the calibrator implementation to see if it has access to a cached table. If so, it proceeds directly to step 3. Cached data is passed as a pointer and length.

The calibration cache data is portable across different devices as long as the calibration happens before layer fusion. Specifically, the calibration cache is portable when using the `IInt8EntropyCalibrator2` or `IInt8MinMaxCalibrator` calibrators, or when `QuantizationFlag::kCALIBRATE_BEFORE_FUSION` is set. This can simplify the workflow, for example by building the calibration table on a machine with a discrete GPU and then reusing it on an embedded platform. Fusions are not guaranteed to be the same across platforms or devices, so calibrating after layer fusion may not result in a portable calibration cache. The calibration cache is in general not portable across TensorRT releases.

As well as quantizing activations, TensorRT must also quantize weights. It uses symmetric quantization with a quantization scale calculated using the maximum absolute values found in the weight tensor. For convolution, deconvolution, and fully connected weights, scales are per-channel.

Note: When the builder is configured to use INT8 I/O, TensorRT still expects calibration data to be in FP32. You can create FP32 calibration data by casting INT8 I/O calibration data to FP32 precision. Also ensure that FP32 cast calibration data is in the range `[-128.0F, 127.0F]` and so can be converted to INT8 data without any precision loss.

INT8 calibration can be used along with the dynamic range APIs. Setting the dynamic range manually overrides the dynamic range generated from INT8 calibration.

Note: Calibration is deterministic - that is, if you provide TensorRT with the same input to calibration in the same order on the same device, the scales generated will be the same across different runs. The data in the calibration cache will be bitwise identical when generated using the same device with the same batch size when provided with identical calibration inputs. The exact data in the calibration cache is not guaranteed to be bitwise identical when generated using different devices, different batch sizes, or using different calibration inputs.

7.3.1. INT8 Calibration Using C++

To provide calibration data to TensorRT, implement the `IInt8Calibrator` interface.

The builder invokes the calibrator as follows:

- First, it queries the interface for the batch size and calls `getBatchSize()` to determine the size of the input batch to expect.
- Then, it repeatedly calls `getBatch()` to obtain batches of input. Batches must be exactly the batch size by `getBatchSize()`. When there are no more batches, `getBatch()` must return `false`.

After you have implemented the calibrator, you can configure the builder to use it:

```
config->setInt8Calibrator(calibrator.get());
```

To cache the calibration table, implement the `writeCalibrationCache()` and `readCalibrationCache()` methods.

7.3.2. Calibration Using Python

The following steps illustrate how to create an INT8 calibrator object using the Python API.

1. Import TensorRT:
```
import tensorrt as trt
```
2. Similar to test/validation datasets, use a set of input files as a calibration dataset. Make sure that the calibration files are representative of the overall inference data files. For TensorRT to use the calibration files, you must create a `batchstream` object. A `batchstream` object is used to configure the calibrator.
```
NUM_IMAGES_PER_BATCH = 5
batchstream = ImageBatchStream(NUM_IMAGES_PER_BATCH, calibration_files)
```
3. Create an `Int8_calibrator` object with input nodes names and batch stream:
```
Int8_calibrator = EntropyCalibrator(["input_node_name"], batchstream)
```
4. Set INT8 mode and INT8 calibrator:
```
config.set_flag(trt.BuilderFlag.INT8)
config.int8_calibrator = Int8_calibrator
```

7.3.3. Quantization Noise Reduction

For networks with implicit quantization, TensorRT attempts to reduce quantization noise in the output by forcing some layers near the network outputs to run in FP32, even if INT8 implementations are available.

The heuristic attempts to ensure that INT8 quantization is smoothed out by summation of multiple quantized values. Layers considered to be "smoothing layers" are convolution, deconvolution, a fully connected layer, or matrix multiplication before reaching the network output. For example, if a network consists of a series of (convolution + activation + shuffle) subgraphs and the network output has type FP32, the last convolution will output FP32 precision, even if INT8 is allowed and faster.

The heuristic does not apply in the following scenarios:

- The network output has type INT8.
- An operation on the path (inclusively) from the last smoothing layer to the output is constrained by `ILayer::setOutputType` or `ILayer::setPrecision` to output INT8.
- There is no smoothing layer with a path to the output, or said that path has an intervening plugin layer.
- The network uses explicit quantization.

7.4. Explicit Quantization

When TensorRT detects the presence of Q/DQ layers in a network, it builds an engine using explicit-precision processing logic and precision-control build flags are not required.

In explicit-quantization, network changes of representation to and from the quantized data type are explicit, therefore INT8 and FP8 must not be used as type constraints.

For a Strongly Typed Networks, builder flags are neither required nor allowed.

7.4.1. Quantized Weights

Weights of Q/DQ models may be specified using a high precision data type (FP32, FP16, or BF16) or a low precision quantized type (INT8, FP8, INT4). When TensorRT builds an engine, high-precision weights are quantized using the scale of `IQuantizeLayer` that operates on the weights and the quantized (low precision) weights are stored in the engine plan file. When using pre-quantized weights (that is, low precision) an `IDequantizeLayer` is required between the weights and the linear operator using the weights.

INT4 quantized weights are stored by packing two elements per byte. The first element is stored in the 4 least-significant bits and the second element is stored in the 4 most-significant bits.

7.4.2. ONNX Support

When a model trained in PyTorch or TensorFlow using Quantization Aware Training (QAT) is exported to ONNX, each fake-quantization operation in the framework's graph is exported as a pair of `QuantizeLinear`and `DequantizeLinear` ONNX operators.When TensorRT imports ONNX models, the ONNX `QuantizeLinear` operator is imported as an `IQuantizeLayer` instance, and the ONNX `DequantizeLinear` operator is imported as an `IDequantizeLayer` instance.

ONNX introduced support for `QuantizeLinear`/`DequantizeLinear` in opset 10, and a quantization-axis attribute was added in opset 13 (required for per-channel quantization). PyTorch 1.8 introduced support for exporting PyTorch models to ONNX using opset 13.

ONNX opset 19 added four FP8 formats, of which TensorRT supports `E4M3FN` (also referred to as `tensor (float8e4m3fn)` in the ONNX operator schema). The latest Pytorch version (Pytorch 2.0) does not support FP8 formats nor does it support export to ONNX using opset 19. In order to bridge the gap, TransformerEngine exports its FP quantization functions as custom ONNX Q/DQ operators that belong to the "trt" domain (`TRT_FP8 QuantizeLinear` and `TRT_FP8 DequantizeLinear`). TensorRT is able to parse both the custom operators and standard opset 19 Q/DQ operators, however, note that opset 19 is not fully supported by TensorRT. Other tools, such as ONNX Runtime, are not able to parse the custom operators. ONNX opset 21 added support for INT4 data type and block quantization.

WARNING: The ONNX GEMM operator is an example that can be quantized per channel. PyTorch `torch.nn.Linear` layers are exported as an ONNX GEMM operator with `(K, C)` weights layout and with the `transB` GEMM attribute enabled (this transposes the weights before performing the GEMM operation). TensorFlow, on the other hand, pretransposes the weights `(C, K)` before ONNX export:

- PyTorch: $y = xW^T$

- TensorFlow: $y = xW$

PyTorch weights are therefore transposed by TensorRT. The weights are quantized by TensorRT before they are transposed, so GEMM layers originating from ONNX QAT models that were exported from PyTorch use dimension `0` for per-channel quantization (axis `K = 0`); while models originating from TensorFlow use dimension `1` (axis `K = 1`).

TensorRT does not support prequantized ONNX models that use INT8/FP8 quantized operators. Specifically, the following ONNX quantized operators are *not* supported and generates an import error if they are encountered when TensorRT imports the ONNX model:

- `QLinearConv`/`QLinearMatmul`
- `ConvInteger`/`MatmulInteger`

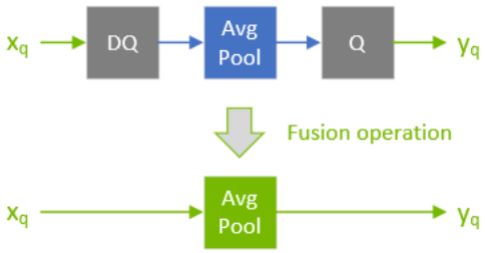
7.4.3. TensorRT Processing of Q/DQ Networks

When TensorRT optimizes a network in Q/DQ-mode, the optimization process is limited to optimizations that do not change the arithmetic correctness of the network. Bit-level accuracy is rarely possible since the order of floating-point operations can produce different results (for example, rewriting $a * s + b * s$ as $(a + b) * s$ is a valid optimization). Allowing these differences is fundamental to backend optimization in general, and this also applies to converting a graph with Q/DQ layers to use quantized operations.

Q/DQ layers control the compute and data precision of a network. An `IQuantizeLayer` instance converts a high-precision floating-point tensor to a quantized tensor by employing quantization, and an `IDequantizeLayer` instance converts a quantized tensor to a high-precision floating-point tensor by means of dequantization. TensorRT expects a Q/DQ layer pair on each of the inputs of quantizable-layers. Quantizable-layers are deep-learning layers that can be converted to quantized layers by fusing with `IQuantizeLayer` and `IDequantizeLayer` instances. When TensorRT performs these fusions, it replaces the quantizable-layers with quantized layers that actually operate on quantized data using compute operations suitable for quantized types.

For the diagrams used in this chapter, green designates low precision (quantized) and blue designates high precision. Arrows represent network activation tensors and squares represent network layers.

Figure 2. A quantizable `AveragePool` layer (in blue) is fused with a DQ layer and a Q layer. All three layers are replaced by a quantized `AveragePool` layer (in green).



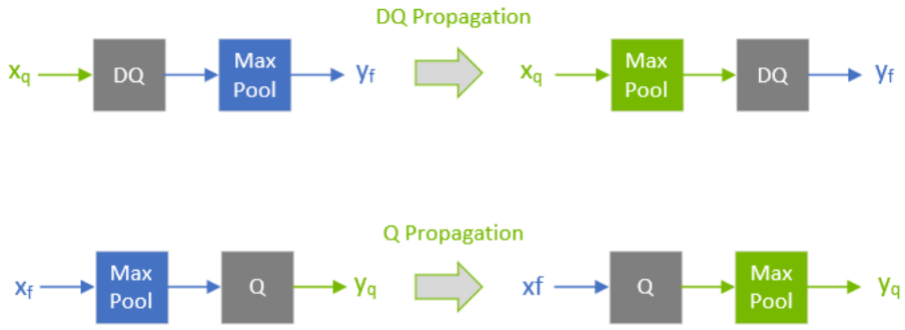
During network optimization, TensorRT moves Q/DQ layers in a process called Q/DQ propagation. The goal in propagation is to maximize the proportion of the graph that can be processed at low precision. Thus, TensorRT propagates Q nodes backwards (so that quantization happens as early as possible) and DQ nodes forward (so that dequantization happens as late as possible). Q-layers can swap places with layers that commute-with-Quantization and DQ-layers can swap places with layers that commute-with-Dequantization.

A layer Op commutes with quantization if $\mathrm{Q}\left(\mathrm{Op}\left(x\right)\right) == \mathrm{Op}\left(\mathrm{Q}\left(x\right)\right)$

Similarly, a layer Op commutes with dequantization if $\mathrm{Op}\left(\mathrm{DQ}\left(x\right)\right) == \mathrm{DQ}\left(\mathrm{Op}\left(x\right)\right)$

The following diagram illustrates DQ forward-propagation and Q backward-propagation. These are legal rewrites of the model because Max Pooling has an INT8 implementation and because Max Pooling commutes with DQ and with Q.

Figure 3. An illustration depicting a DQ forward-propagation and Q backward-propagation.



Note:

To understand Max Pooling commutation, let us look at the output of the maximum-pooling operation applied to some arbitrary input. Max Pooling is applied to groups of

input coefficients and outputs the coefficient with the maximum value. For group `i` composed of coefficients $\{x_0 .. x_m\}$:

$$\text{output}_i := \max(\{x_0, x_1, \ldots x_m\}) = \max(\{\max(\{\max(\{x_0, x_1\}), x_2\}), \ldots x_m\})$$

It is therefore enough to look at two arbitrary coefficients without loss of generality (WLOG):

$$x_j = \max(\{x_j, x_k\}) \text{ for } x_j >= x_k$$

For quantization function $Q(a, \text{scale}, x_{max}, x_{min}) := \text{truncate}(\text{round}(a / \text{scale}), x_{max}, x_{min})$, with $\text{scale} > 0$, note that (without providing proof, and using simplified notation):

$$Q(x_j, \text{scale}) >= Q(x_k, \text{scale}) \text{ for } x_j >= x_k$$

Therefore:

$$\max(\{Q(x_j, \text{scale}), Q(x_k, \text{scale})\}) = Q(x_j, \text{scale}) \text{ for } x_j >= x_k$$

However, by definition:

$$Q(\max(\{x_j, x_k\}), \text{scale}) = Q(x_j, \text{scale}) \text{ for } x_j >= x_k$$

Function max commutes-with-quantization and so does Max Pooling.

Similarly for dequantization, function $DQ(a, \text{scale}) := a * \text{scale}$ with $\text{scale} > 0$ we can show that:

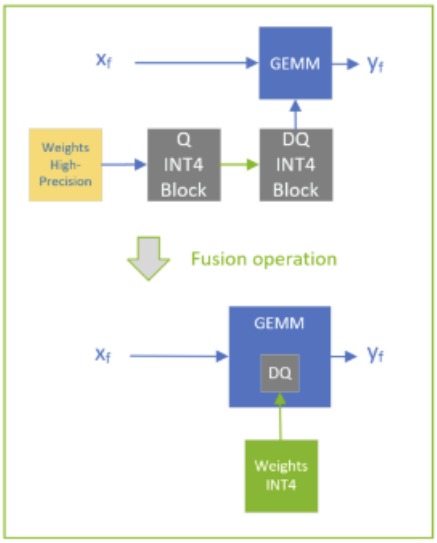
$$\max(\{DQ(x_j, \text{scale}), DQ(x_k, \text{scale})\}) = DQ(x_j, \text{scale}) = DQ(\max(\{x_j, x_k\}), \text{scale}) \text{ for } x_j >= x_k$$

There is a distinction between how quantizable-layers and commuting-layers are processed. Both types of layers can be computed in INT8/FP8, but quantizable-layers also fuse with DQ input layers and a Q output layer. For example, an `AveragePooling` layer (quantizable) does not commute with either Q or DQ, so it is quantized using Q/DQ fusion as illustrated in the first diagram. This is in contrast to how Max Pooling (commuting) is quantized.

7.4.4. Weight-Only Quantization

Weight-only quantization (WoQ) is an optimization that is useful when memory bandwidth limits the performance of GEMM operations or when GPU memory is scarce. In WoQ, GEMM weights are quantized to INT4 precision while the GEMM input data and compute operation remain in high-precision. TensorRT's WoQ kernels read the 4-bit weights from memory and dequantize them just before performing the dot product in high-precision.

Figure 4. Weight-only Quantization (WoQ)



WoQ is available only for INT4 block quantization with GEMM layers. The GEMM data input is specified in high-precision (FP32, FP16, BF16) and the weights are quantized using Q/DQ as usual. TensorRT creates an engine having INT4 weights and a high-precision GEMM operation. The engine reads the low-precision weights and dequantizes them before performing the GEMM operation in high-precision.

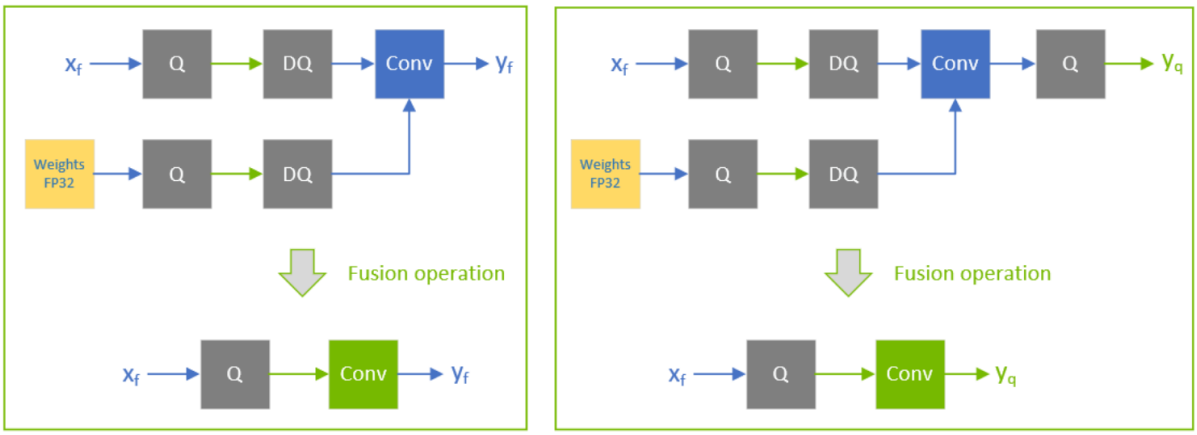
7.4.5. Q/DQ Layer-Placement Recommendations

The placement of Q/DQ layers in a network affects performance and accuracy. Aggressive quantization can lead to degradation in model accuracy because of the error introduced by quantization. But quantization also enables latency reductions. Listed here are some recommendations for placing Q/DQ layers in your network.

Note that older devices may not have low precision kernel implementations for all layers and you may encounter a `could not find any implementation` error while building your engine. To resolve this, remove the Q/DQ nodes which quantize the failing layers.

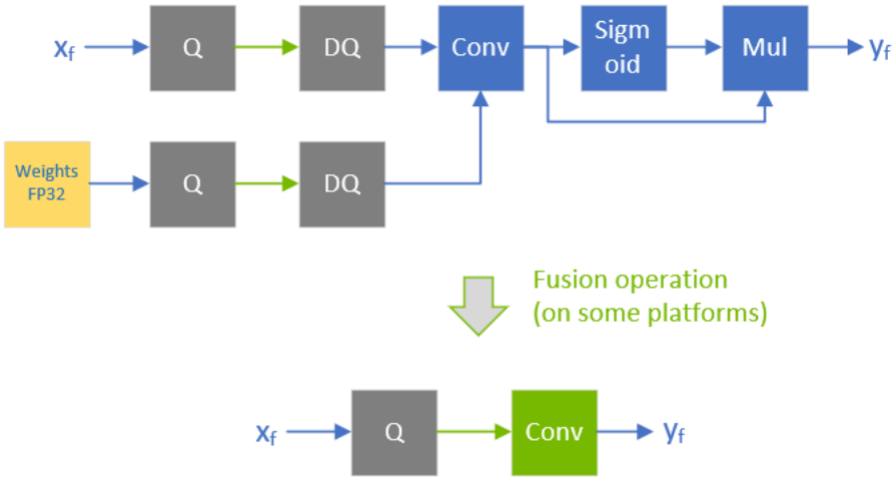
Quantize all inputs of weighted-operations (Convolution, Transposed Convolution, and GEMM). Quantization of the weights and activations reduces bandwidth requirements and also enables INT8 computation to accelerate bandwidth-limited and compute-limited layers.

Figure 5. Two examples of how TensorRT fuses convolutional layers. On the left only the inputs are quantized; and on the right both inputs and output are quantized.



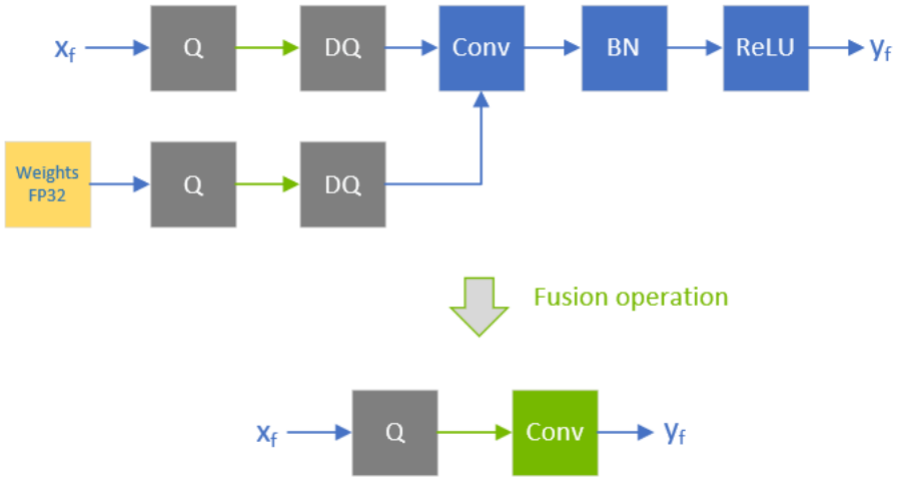
By default, do not quantize the outputs of weighted-operations. It is sometimes useful to preserve the higher-precision dequantized output. For example, if the linear operation is followed by an activation function (SiLU, in the following diagram) that requires higher precision input to produce acceptable accuracy.

Figure 6. Example of a linear operation followed by an activation function.



Do not simulate batch-normalization and ReLU fusions in the training framework because TensorRT optimizations guarantee to preserve the arithmetic semantics of these operations.

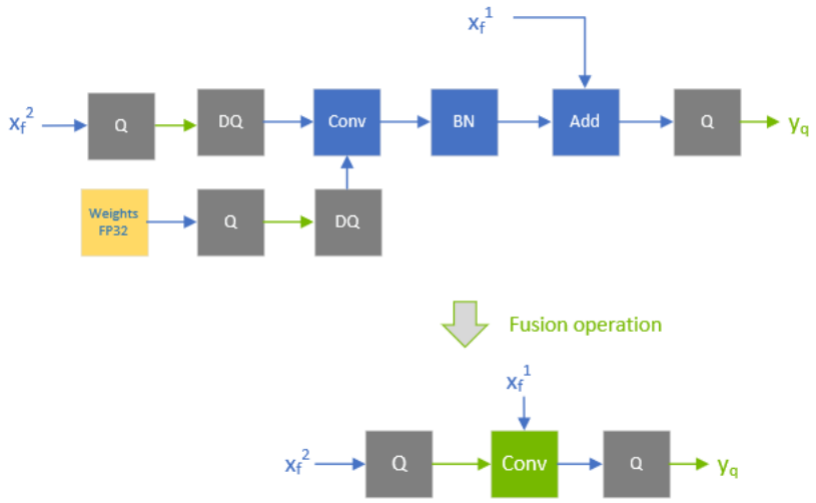
Figure 7. Batch normalization is fused with convolution and ReLU while keeping the same execution order as defined in the pre-fusion network. There is no need to simulate BN-folding in the training network.



Quantize the residual input in skip-connections. TensorRT can fuse element-wise addition following weighted layers, which are useful for models with skip connections like ResNet and EfficientNet. The precision of the first input to the element-wise addition layer determines the precision of the output of the fusion.

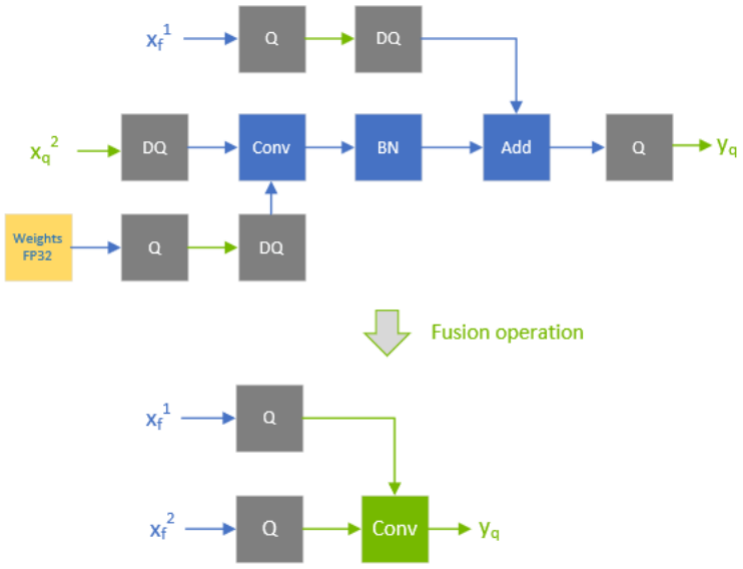
For example, in the following diagram, the precision of xf1 is floating point, so the output of the fused convolution is limited to floating-point, and the trailing Q-layer cannot be fused with the convolution.

Figure 8. The precision of x_f^1 is floating point, so the output of the fused convolution is limited to floating-point, and the trailing Q-layer cannot be fused with the convolution.



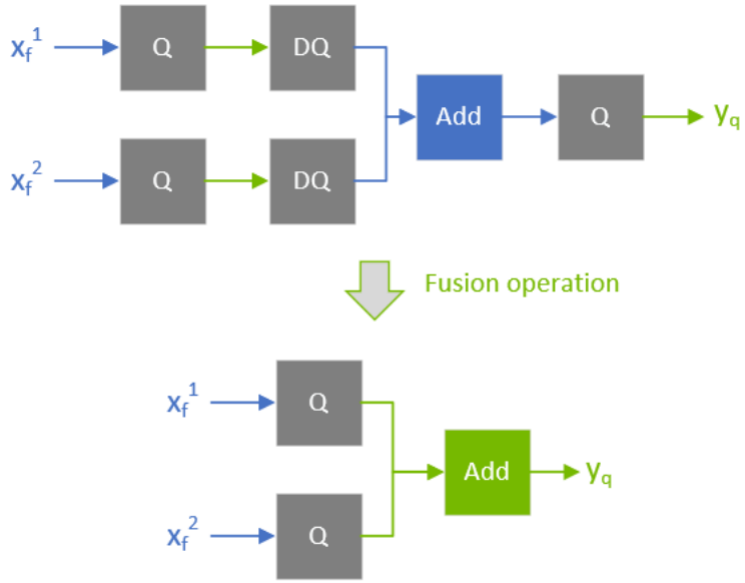
In contrast, when x_f^1 is quantized to INT8, as depicted in the following diagram, the output of the fused convolution is also INT8, and the trailing Q-layer is fused with the convolution.

Figure 9. When x_f^1 is quantized to INT8, the output of the fused convolution is also INT8, and the trailing Q-layer is fused with the convolution.



For extra performance, try quantizing layers that do not commute with Q/DQ. Currently, non-weighted layers that have INT8 inputs also require INT8 outputs, so quantize both inputs and outputs.

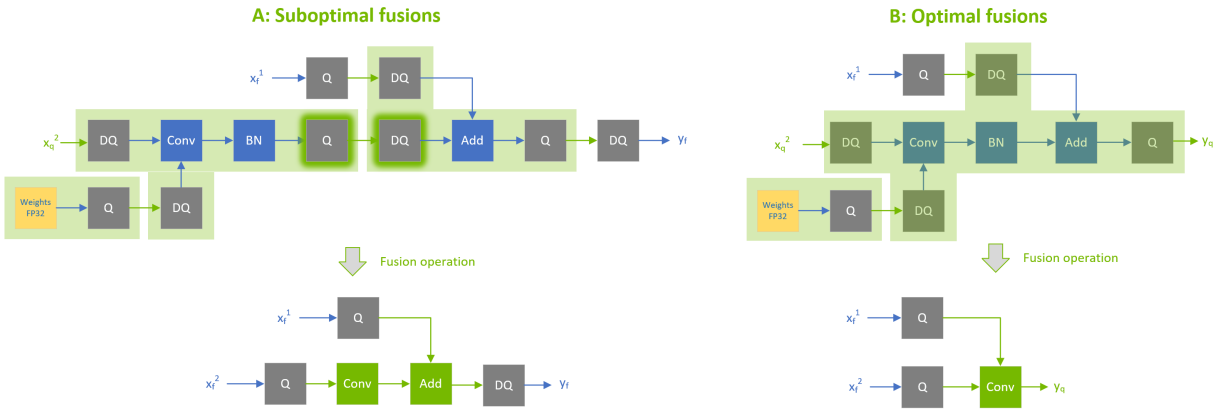
Figure 10. An example of quantizing a quantizable operation. An element-wise addition is fused with the input DQs and the output Q.



Performance can decrease if TensorRT cannot fuse the operations with the surrounding Q/DQ layers, so be conservative when adding Q/DQ nodes and experiment with accuracy and TensorRT performance in mind.

The following figure is an example of suboptimal fusions (the highlighted light green background rectangles) that can result from extra Q/DQ operations. Contrast the following figure with Figure 9, which shows a more performant configuration. The convolution is fused separately from the element-wise addition because each of them is surrounded by Q/DQ pairs. The fusion of the element-wise addition is shown in Figure 10.

Figure 11. An example of suboptimal quantization fusions: contrast the suboptimal fusion in A and the optimal fusion in B. The extra pair of Q/DQ operations (highlighted with a glowing-green border) forces the separation of the convolution from the element-wise addition.



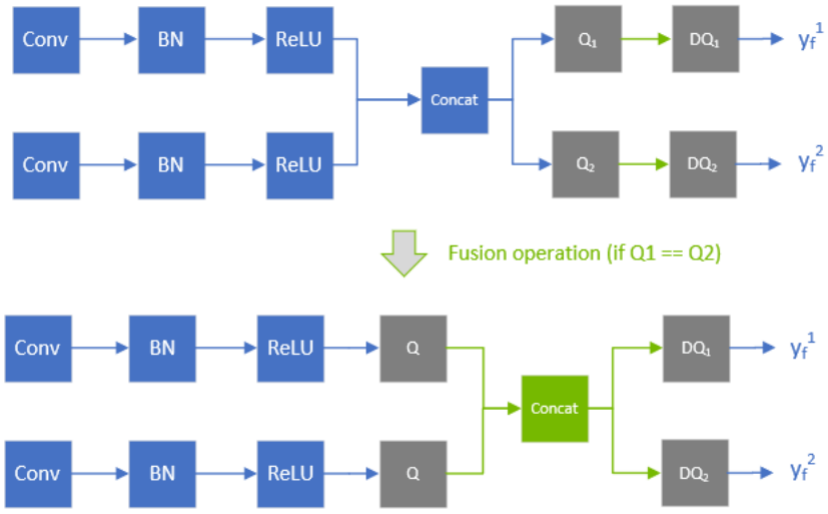
Use per-tensor quantization for activations; and per-channel quantization for weights.This configuration has been demonstrated empirically to lead to the best quantization accuracy.

You can further optimize engine latency by enabling FP16. TensorRT attempts to use FP16 instead of FP32 whenever possible (this is not currently supported for all layer types).

7.4.6. Q/DQ Limitations

A few of the Q/DQ graph-rewrite optimizations that TensorRT performs compare the values of quantization scales between two or more Q/DQ layers and only perform the graph-rewrite if the compared quantization scales are equal. When a refittable TensorRT engine is refitted, the scales of Q/DQ nodes can be assigned new values. During the refitting operation of Q/DQ engines, TensorRT checks if Q/DQ layers that participated in scale-dependent optimizations are assigned new values that break the rewrite optimizations and throws an exception if true.

Figure 12. An example showing scales of Q1 and Q2 are compared for equality, and if equal, they are allowed to propagate backward. If the engine is refitted with new values for Q1 and Q2 such that `Q1 != Q2`, then an exception aborts the refitting process.



7.4.7. Q/DQ Interaction with Plugins

Plugins extend the capabilities of TensorRT by allowing the replacement of a group of layers with a custom and proprietary implementation. You can decide what functionality to include in the plugin and what to leave for TensorRT to handle.

The same follows for a TensorRT network with Q/DQ layers: when a plugin consumes quantized inputs (INT8/FP8) and generates quantized quantized outputs, the input DQ and output Q nodes must be included as part of the plugin and removed from the network.

Consider a simple case of a sequential graph consisting of a single INT8 plugin (aptly named `MyInt8Plugin`) sandwiched between two convolution layers (ignoring weights quantization):

```
Input > Q -> DQ > Conv > Q -> DQ_i > MyInt8Plugin > Q_o -> DQ > Conv > Output
```

The `>` arrows indicate activation tensors with FP32 precision and the `->` arrows indicate INT8 precision.

When TensorRT optimizes this graph, it fuses the layers to the following graph (square brackets indicate TensorRT fusions):

```
Input > Q -> [DQ → Conv → Q] -> DQ_i > MyInt8Plugin > Q_o -> [DQ → Conv] > Output
```

In the graph above, the plugin consumes and generates FP32 inputs and outputs. Since the plugin `MyInt8Plugin` uses INT8 precision, the subsequent procedure involves the manual integration of `DQ_i` and `Q_o` with the `MyInt8Plugin`, followed by invoking the `setOutputType(kINT8)` method for this particular plugin layer; TensorRT will see a network like this:

```
Input > Q -> DQ > Conv > Q -> MyInt8Plugin -> DQ > Conv > Output
```

Which it will fuse to:

```
Input > Q -> [DQ → Conv → Q] > MyInt8Plugin -> [DQ → Conv] > Output
```

When "manually fusing" `DQ_i`, you take the input quantization scale and give it to your plugin, so it will know how to dequantize (if needed) the input. The same follows for using the scale from `Q_o` in order to quantize your plugin's output.

7.4.8. QAT Networks Using TensorFlow

We provide an open-source TensorFlow-Quantization Toolkit to perform QAT in TensorFlow 2 Keras models following NVIDIA's QAT recipe. This leads to optimal model acceleration with TensorRT on NVIDIA GPUs and hardware accelerators. More details can be found in the NVIDIA TensorFlow-Quantization Toolkit User Guide.

TensorFlow 1 does not support per-channel quantization (PCQ). PCQ is recommended for weights in order to preserve the accuracy of the model.

7.4.9. QAT Networks Using PyTorch

PyTorch 1.8.0 and forward support ONNX QuantizeLinear/DequantizeLinear support per channel scales. You can use pytorch-quantization to do INT8 calibration, run quantization aware fine-tuning, generate ONNX and finally use TensorRT to run inference on this ONNX model. More detail can be found in NVIDIA PyTorch-Quantization Toolkit User Guide.

7.4.10. QAT Networks Using TransformerEngine

We provide TransformerEngine, an open-source library for accelerating training, inference and exporting of transformer models. It includes APIs for building a Transformer layer as well as a framework agnostic library in C++ including structs and kernels needed for FP8 support. Modules provided by TransformerEngine internally maintain scaling factors and other values needed for FP8 training. You can use TransformerEngine to train a mixed precision model, export an ONNX model, and finally use TensorRT to run inference on this ONNX model.

7.5. Quantized Types Rounding Modes

Backend	Compute Kernel Quantization (FP32 to INT8/FP8)	Weights Quantization (FP32 to INT8/FP8)	
		Quantized Network (QAT)	Dynamic Range API / Calibration
GPU	round-to-nearest-with-ties-to-even	round-to-nearest-with-ties-to-even (INT8, FP8, INT4)	round-to-nearest-with-ties-to-positive-infinity (INT8 only)

Backend	Compute Kernel Quantization (FP32 to INT8/FP8)	Weights Quantization (FP32 to INT8/FP8)	
		Quantized Network (QAT)	Dynamic Range API / Calibration
DLA	`round-to-nearest-with-ties-to-even`	Not Applicable	`round-to-nearest-with-ties-to-even (INT8 only)`

Chapter 8. Working with Dynamic Shapes

Dynamic Shapes is the ability to defer specifying some or all tensor dimensions until runtime. Dynamic shapes can be used through both the C++ and Python interfaces.

The following sections provide greater detail; however, here is an overview of the steps for building an engine with dynamic shapes:

1. The network definition must not have an implicit batch dimension.
   C++
   Create the `INetworkDefinition` by calling
   ```
   IBuilder::createNetworkV2(1U <<
           static_cast<int>(NetworkDefinitionCreationFlag::kEXPLICIT_BATCH))
   ```
   Python
   Create the `tensorrt.INetworkDefinition` by calling
   ```
   create_network(1 <<
           int(tensorrt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
   ```
   These calls request that the network not have an implicit batch dimension.
2. Specify each runtime dimension of an input tensor by using `-1` as a placeholder for the dimension.
3. Specify one or more *optimization profiles* at build time that specify the permitted range of dimensions for inputs with runtime dimensions, and the dimensions for which the auto-tuner will optimize. For more information, refer to Optimization Profiles.
4. To use the engine:
   a). Create an execution context from the engine, the same as without dynamic shapes.
   b). Specify one of the optimization profiles from step 3 that covers the input dimensions.
   c). Specify the input dimensions for the execution context. After setting input dimensions, you can get the output dimensions that TensorRT computes for the given input dimensions.
   d). Enqueue work.

To change the runtime dimensions, repeat steps 4b and 4c, which do not have to be repeated until the input dimensions change.

When the preview features (`PreviewFeature::kFASTER_DYNAMIC_SHAPES_0805`) is enabled, it can potentially, for dynamically shaped networks:

- reduce the engine build time,
- reduce runtime, and
- decrease device memory usage and engine size.

Models most likely to benefit from enabling `kFASTER_DYNAMIC_SHAPES_0805` are transformer-based models and models containing dynamic control flows.

8.1. Specifying Runtime Dimensions

When building a network, use `-1` to denote a runtime dimension for an input tensor. For example, to create a 3D input tensor named `foo` where the last two dimensions are specified at runtime, and the first dimension is fixed at build time, issue the following.

C++

```
networkDefinition.addInput("foo", DataType::kFLOAT, Dims3(3, -1, -1))
```

Python

```
network_definition.add_input("foo", trt.float32, (3, -1, -1))
```

At run time, you must set the input dimensions after choosing an optimization profile (refer to Optimization Profiles). Let the input have dimensions `[3,150,250]`. After setting an optimization profile for the previous example, you would call:

C++

```
context.setInputShape("foo", Dims{3, {3, 150, 250}})
```

Python

```
context.set_input_shape("foo", (3, 150, 250))
```

At runtime, asking the engine for binding dimensions returns the same dimensions used to build the network, meaning, you get a `-1` for each runtime dimension. For example:

C++

`engine.getTensorShape("foo")` returns a `Dims` with dimensions `{3, -1, -1}`..

Python

`engine.get_tensor_shape("foo")` returns `(3, -1, -1)`.

To get the actual dimensions, which are specific to each execution context, query the execution context:

C++

`context.getTensorShape("foo")` returns a `Dims` with dimensions `{3, 150, 250}`.

Python

`context.get_tensor_shape(0)` returns `(3, 150, 250)`.

Note: The return value of `setInputShape` for an input only indicates consistency with respect to the optimization profile set for that input. After all input binding dimensions are specified, you can check whether the entire network is consistent with respect to the dynamic input shapes by querying the dimensions of the output bindings of the network. Here is an example that retrieves the dimensions of an output named `bar`:

```
nvinfer1::Dims outDims = context->getTensorShape("bar");

if (outDims.nbDims == -1) {
```

```
    gLogError << "Invalid network output, this might be caused by inconsistent input
 shapes." << std::endl;
    // abort inference
}
```

If a dimension *k* is data-dependent, for example, it depends on the output of `INonZeroLayer`, `outDims.d[`*k*`]` will be -1. For more information, refer to Dynamically Shaped Output for how to deal with such outputs.

8.2. Named Dimensions

Both constant and runtime dimensions can be named. Naming dimensions provides two benefits:

- For runtime dimensions, error messages use the dimension's name. For example, if an input tensor `foo` has dimensions `[n,10,m]`, it is more helpful to get an error message about `m` instead of `(#2 (SHAPE foo))`.
- Dimensions with the same name are implicitly equal, which can help the optimizer generate a more efficient engine, and diagnoses mismatched dimensions at runtime. For example, if two inputs have dimensions `[n,10,m]` and `[n,13]`, the optimizer knows the lead dimensions are always equal, and accidentally use of the engine with mismatched values for `n` will be reported as an error.

You can use the same name for both constant and runtime dimensions as long as they are always equal at runtime.

The following syntax examples sets the name of the third dimension of tensor to `m`.

C++

```
tensor.setDimensionName(2, "m")
```

Python

```
tensor.set_dimension_name(2, "m")
```

There are corresponding methods to get a dimensions name:

C++

```
tensor.getDimensionName(2) // returns the name of the third dimension of tensor, or
 nullptr if it does not have a name.
```

Python

```
tensor.get_dimension_name(2) # returns the name of the third dimension of tensor, or None
 if it does not have a name.
```

When the input network is imported from an ONNX file, the ONNX parser automatically sets the dimension names using the names in the ONNX file. Therefore, if two dynamic dimensions are expected to be equal at runtime, specify the same name for these dimensions when exporting the ONNX file.

8.3. Dimension Constraint using IAssertionLayer

Sometimes, two dynamic dimensions are not equal but are guaranteed to be equal at runtime. Letting TensorRT know they are equal can help it build a more efficient engine. There are two ways to convey the equality constraint to TensorRT:

- Give the dimensions the same name, as described in Named Dimensions.
- Use `IAssertionLayer` to express the constraint. This technique is more general since it can convey trickier equalities.

For example, if the first dimension of tensor A is guaranteed to be one more than the first dimension of tensor B, then the constraint can be established by:

C++

```
// Assumes A and B are  ITensor* and n is a INetworkDefinition&.
auto shapeA = n.addShape(*A)->getOutput(0);
auto firstDimOfA = n.addSlice(*shapeA, Dims{1, {0}}, Dims{1, {1}}, Dims{1, {1}})-
>getOutput(0);
auto shapeB = n.addShape(*B)->getOutput(0);
auto firstDimOfB = n.addSlice(*shapeB, Dims{1, {0}}, Dims{1, {1}}, Dims{1, {1}})-
>getOutput(0);
static int32_t const oneStorage{1};
auto one = n.addConstant(Dims{1, {1}}, Weights{DataType::kINT32, &oneStorage, 1})-
>getOutput(0);
auto firstDimOfBPlus1 = n.addElementWise(*firstDimOfB, *one, ElementWiseOperation::kSUM)-
>getOutput(0);
auto areEqual = n.addElementWise(*firstDimOfA, *firstDimOfBPlus1,
 ElementWiseOperation::kEQUAL)->getOutput(0);
n.addAssertion(*areEqual, "oops");
```

Python

```
# Assumes `a` and `b` are ITensors and `n` is an INetworkDefinition
shape_a = n.add_shape(a).get_output(0)
first_dim_of_a = n.add_slice(shape_a, (0, ), (1, ), (1, )).get_output(0)
shape_b = n.add_shape(b).get_output(0)
first_dim_of_b = n.add_slice(shape_b, (0, ), (1, ), (1, )).get_output(0)
one = n.add_constant((1, ), np.ones((1, ), dtype=np.int32)).get_output(0)
first_dim_of_b_plus_1 = n.add_elementwise(first_dim_of_b, one,
 trt.ElementWiseOperation.SUM).get_output(0)
are_equal = n.add_elementwise(first_dim_of_a, first_dim_of_b_plus_1,
 trt.ElementWiseOperation.EQUAL).get_output(0)
n.add_assertion(are_equal, "oops")
```

If the dimensions violate the assertion at runtime, TensorRT will throw an error.

8.4. Optimization Profiles

An *optimization profile* describes a range of dimensions for each network input and the dimensions that the auto-tuner will use for optimization. When using runtime dimensions, you must create at least one optimization profile at build time. Two profiles can specify disjoint or overlapping ranges.

For example, one profile might specify a minimum size of `[3,100,200]`, a maximum size of `[3,200,300]`, and optimization dimensions of `[3,150,250]` while another profile

might specify min, max and optimization dimensions of `[3,200,100]`, `[3,300,400]`, and `[3,250,250]`.

Note: Based on the dimensions specified by the `min`, `max`, and `opt` parameters, the memory usage for different profiles can change dramatically. There are some operations that have tactics that only work for `MIN=OPT=MAX`, so when these values differ, the tactic is disabled.

To create an optimization profile, first construct an `IOptimizationProfile`. Then set the min, optimization, and max dimensions, and add it to the network configuration. The shapes defined by the optimization profile must define valid input shapes for the network. Here are the calls for the first profile mentioned previously for an input `foo`:

C++

```
IOptimizationProfile* profile = builder.createOptimizationProfile();
profile->setDimensions("foo", OptProfileSelector::kMIN, Dims3(3,100,200);
profile->setDimensions("foo", OptProfileSelector::kOPT, Dims3(3,150,250);
profile->setDimensions("foo", OptProfileSelector::kMAX, Dims3(3,200,300);

config->addOptimizationProfile(profile)
```

Python

```
profile = builder.create_optimization_profile();
profile.set_shape("foo", (3, 100, 200), (3, 150, 250), (3, 200, 300))
config.add_optimization_profile(profile)
```

At runtime, you must set an optimization profile before setting input dimensions. Profiles are numbered in the order that they were added, starting at `0`. Note that each execution context must use a separate optimization profile.

To choose the first optimization profile in the example, use:

C++

```
context.setOptimizationProfileAsync(0, stream)
```

Python

```
context.set_optimization_profile_async(0, stream)
```

The provided `stream` argument should be the same CUDA stream that will be used for the subsequent `enqueue()`, `enqueueV2()`, or `enqueueV3()` invocation in this context. This ensures that the context executions happen after the optimization profile setup is done.

If the associated CUDA engine has dynamic inputs, the optimization profile must be set at least once with a unique profile index that is not used by other execution contexts that are not destroyed. For the first execution context that is created for an engine, profile 0 is chosen implicitly.

`setOptimizationProfileAsync()` can be called to switch between profiles. It must be called after any `enqueue()`, `enqueueV2()`, or `enqueueV3()` operations finish in the current context. When multiple execution contexts run concurrently, it is allowed to switch to a profile that was formerly used but already released by another execution context with different dynamic input dimensions.

`setOptimizationProfileAsync()` function replaces the now deprecated version of the API `setOptimizationProfile()`. Using `setOptimizationProfile()` to switch between optimization profiles can cause GPU memory copy operations in the subsequent `enqueue()` or `enqueueV2()` operations. To avoid these calls during enqueue, use `setOptimizationProfileAsync()` API instead.

8.5. Dynamically Shaped Output

If an output of a network has a dynamic shape, there are several strategies available to allocate the output memory.

If the dimensions of the output are computable from the dimensions of inputs, use `IExecutionContext::getTensorShape()` to get the dimensions of the output, after providing dimensions of the input tensors and Shape Tensor I/O (Advanced). Use the `IExecutionContext::inferShapes()` method to check if you forgot to supply the necessary information.

Otherwise, or if you do not know if the dimensions of the output are computable in advance or calling `enqueueV3`, associate an `IOutputAllocator` with the output. More specifically:

1. Derive your own allocator class from `IOutputAllocator`.
2. Override the `reallocateOutput` and `notifyShape` methods. TensorRT calls the first when it needs to allocate the output memory, and the second when it knows the output dimensions. For example, the memory for the output of `INonZeroLayer` is allocated before the layer runs.

Here is an example derived class:

```
class MyOutputAllocator : nvinfer1::IOutputAllocator
{
public:
    void* reallocateOutput(
        char const* tensorName, void* currentMemory,
        uint64_t size, uint64_t alignment) override
    {
        // Allocate the output. Remember it for later use.
        outputPtr = ... depends on strategy, as discussed later...
       return outputPtr;
    }

    void notifyShape(char const* tensorName, Dims const& dims)
    {
        // Remember output dimensions for later use.
        outputDims = dims;
    }

    // Saved dimensions of the output
    Dims outputDims{};

    // nullptr if memory could not be allocated
    void* outputPtr{nullptr};
};
```

Here's an example of how it might be used:

```
std::unordered_map<std::string, MyOutputAllocator> allocatorMap;

for (const char* name : names of outputs)
{
    Dims extent = context->getTensorShape(name);
    void* ptr;
    if (engine->getTensorLocation(name) == TensorLocation::kDEVICE)
    {
        if (extent.d contains a -1)
        {
```

```
            auto allocator = std::make_unique<MyOutputAllocator>();
            context->setOutputAllocator(name, allocator.get());
            allocatorMap.emplace(name, std::move(allocator));
        }
        else
        {
            ptr = allocate device memory per extent and format
                  }
    }
    else
    {
        ptr = allocate cpu memory per extent and format
    }
    context->setTensorAddress(name, ptr);
}
```

Several strategies can be used for implementing `reallocateOutput`:

A
Defer allocation until the size is known. Do not call `IExecution::setTensorAddress`, or call it with a `nullptr` for the tensor address.

B
Preallocate enough memory, based on what `IExecutionTensor::getMaxOutputSize` reports as an upper bound. That guarantees that the engine will not fail for lack of sufficient output memory, but the upper bound may be so high as to be useless.

C
Preallocate enough memory based on experience, use `IExecution::setTensorAddress` to tell TensorRT about it. Make `reallocateOutput` return `nullptr` if the tensor does not fit, which will cause the engine to fail gracefully.

D
Preallocate memory as in `C`, but have `reallocateOutput` return a pointer to a bigger buffer if there is a fit problem. This increases the output buffer as needed.

E
Defer allocation until the size is known, like `A`. Then, attempt to recycle that allocation in subsequent calls until a bigger buffer is requested, and then increase it like in `D`.

Here's an example derived class that implements `E`:

```
class FancyOutputAllocator : nvinfer1::IOutputAllocator
{
public:
    void reallocateOutput(
        char const* tensorName, void* currentMemory,
        uint64_t size, uint64_t alignment) override
    {
        if (size > outputSize)
        {
            // Need to reallocate
            cudaFree(outputPtr);
            outputPtr = nullptr;
            outputSize = 0;
            if (cudaMalloc(&outputPtr, size) == cudaSuccess)
            {
                outputSize = size;
            }
        }
        // If the cudaMalloc fails, outputPtr=nullptr, and engine
        // gracefully fails.
        return outputPtr;
    }

    void notifyShape(char const* tensorName, Dims const& dims)
    {
```

```
        // Remember output dimensions for later use.
        outputDims = dims;
    }

    // Saved dimensions of the output tensor
    Dims outputDims{};

    // nullptr if memory could not be allocated
    void* outputPtr{nullptr};

    // Size of allocation pointed to by output
    uint64_t outputSize{0};

    ~FancyOutputAllocator() override
    {
        cudaFree(outputPtr);
    }
};
```

8.5.1. Looking up Binding Indices for Multiple Optimization Profiles

You can skip this section if using `enqueueV3` instead of the deprecated `enqueueV2`, because the name-based methods such as `IExecutionContext::setTensorAddress` expect no profile suffix.

In an engine built from multiple profiles, there are separate binding indices for each profile. The names of I/O tensors for the Kth profile have `[profile K]` appended to them, with K written in decimal. For example, if the `INetworkDefinition` had the name `"foo"`, and `bindingIndex` refers to that tensor in the optimization profile with index `3`, `engine.getBindingName(bindingIndex)` returns `"foo [profile 3]"`.

Likewise, if using `ICudaEngine::getBindingIndex(name)` to get the index for a profile K beyond the first profile ($K=0$), append `"[profile K]"` to the name used in the `INetworkDefinition`. For example, if the tensor was called `"foo"` in the `INetworkDefinition`, then `engine.getBindingIndex("foo [profile 3]")` returns the binding index of Tensor `"foo"` in optimization profile `3`.

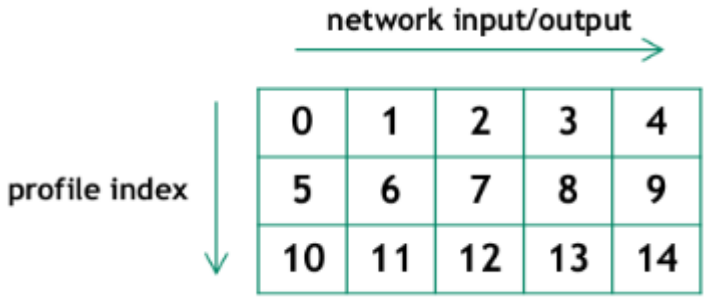
Always omit the suffix for $K=0$.

8.5.2. Bindings For Multiple Optimization Profiles

This section explains the deprecated interface `enqueueV2` and its binding indices. The newer interface `enqueueV3` does away with binding indices.

Consider a network with four inputs, one output, with three optimization profiles in the `IBuilderConfig`. The engine has 15 bindings, five for each optimization profile, conceptually organized as a table:

Figure 13. Optimization Profile



Each row is a profile. Numbers in the table denote binding indices. The first profile has binding indices 0..4, the second has 5..9, and the third has 10..14.

The interfaces have an "auto-correct" for the case that the binding belongs to the *first* profile, but another profile was specified. In that case, TensorRT warns about the mistake and then chooses the correct binding index from the same column.

For the sake of backward semi-compatibility, the interfaces have an "auto-correct" in the scenario where the binding belongs to the *first* profile, but another profile was specified. In this case, TensorRT warns about the mistake and then chooses the correct binding index from the same column.

8.6. Layer Extensions For Dynamic Shapes

Some layers have optional inputs that allow specifying dynamic shape information; `IShapeLayer` can be used for accessing the shape of a tensor at runtime. Furthermore, some layers allow calculating new shapes. The next section goes into semantic details and restrictions. Here is a summary of what you might find useful in conjunction with dynamic shapes.

`IShapeLayer` outputs a 1D tensor containing the dimensions of the input tensor. For example, if the input tensor has dimensions `[2,3,5,7]`, the output tensor is a four-element 1D tensor containing `{2,3,5,7}`. If the input tensor is a scalar, it has dimensions `[]`, and the output tensor is a zero-element 1D tensor containing `{}`.

`IResizeLayer` accepts an optional second input containing the desired dimensions of the output.

`IShuffleLayer` accepts an optional second input containing the reshape dimensions before the second transpose is applied. For example, the following network reshapes a tensor `Y` to have the same dimensions as `X`:

C++

```
auto* reshape = networkDefinition.addShuffle(Y);
reshape.setInput(1, networkDefintion.addShape(X)->getOutput(0));
```

Python

```
reshape = network_definition.add_shuffle(y)
reshape.set_input(1, network_definition.add_shape(X).get_output(0))
```

`ISliceLayer` accepts an optional second, third, and fourth input containing the start, size, and stride.

```
IConcatenationLayer, IElementWiseLayer, IGatherLayer, IIdentityLayer, and
        IReduceLayer
```

can be used to do calculations on shapes and create new shape tensors.

8.7. Restrictions For Dynamic Shapes

The following layer restrictions arise because the layer's weights have a fixed size:

- `IConvolutionLayer` and `IDeconvolutionLayer` require that the channel dimension be a build time constant.
- `IFullyConnectedLayer` requires that the last three dimensions be build-time constants.
- `Int8` requires that the channel dimension be a build time constant.
- Layers accepting additional shape inputs (`IResizeLayer`, `IShuffleLayer`, `ISliceLayer`) require that the additional shape inputs be compatible with the dimensions of the minimum and maximum optimization profiles as well as with the dimensions of the runtime data input; otherwise, it can lead to either a build-time or runtime error.

Values that must be build-time constants do not have to be constants at the API level. TensorRT's shape analyzer does element by element constant propagation through layers that do shape calculations. It is sufficient that the constant propagation discovers that a value is a build time constant.

For more information regarding layers, refer to the NVIDIA TensorRT Operator's Reference.

8.8. Execution Tensors Versus Shape Tensors

TensorRT 8.5 largely erases the distinctions between execution tensors and shape tensors. However, if designing a network or analyzing performance, it may help to understand the internals and where internal synchronization is incurred.

Engines using dynamic shapes employ a ping-pong execution strategy.

1. Compute the shapes of tensors on the CPU until a shape requiring GPU results is reached.
2. Stream work to the GPU until out of work or an unknown shape is reached If the latter, synchronize and go back to step 1.

An *execution tensor* is a traditional TensorRT tensor. A *shape tensor* is a tensor that is related to shape calculations. It must have type `Int32`, `Int64`, `Float`, or `Bool`, its shape must be determinable at build time, and it must have no more than 64 elements. Refer to Shape Tensor I/O (Advanced) for additional restrictions for shape tensors at

network I/O boundaries. For example, there is an `IShapeLayer` whose output is a 1D tensor containing the dimensions of the input tensor. The output is a shape tensor. `IShuffleLayer` accepts an optional second input that can specify reshaping dimensions. The second input must be a shape tensor.

Some layers are "polymorphic" with respect to the kinds of tensors that they handle. For example, `IElementWiseLayer` can sum two INT32 execution tensors or sum two INT32 shape tensors. The type of tensor depends on its ultimate use. If the sum is used to reshape another tensor, then it is a "shape tensor."

When TensorRT needs a shape tensor, but the tensor has been classified as an execution tensor, the runtime has to copy the tensor from the GPU to the CPU, which incurs synchronization overhead.

8.8.1. Formal Inference Rules

The formal inference rules used by TensorRT for classifying tensors are based on a type-inference algebra. Let `E` denote an execution tensor and `S` denote a shape tensor.

`IActivationLayer` has the signature:

```
IActivationLayer: E → E
```

since it takes an execution tensor as an input and an execution tensor as an output. `IElementWiseLayer` is polymorphic in this respect, with two signatures:

```
IElementWiseLayer: S × S → S, E × E → E
```

For brevity, let us adopt the convention that t is a variable denoting either class of tensor, and all t in a signature refers to the same class of tensor. Then, the two previous signatures can be written as a single polymorphic signature:

```
IElementWiseLayer: t × t → t
```

The two-input `IShuffleLayer` has a shape tensor as the second input and is polymorphic with respect to the first input:

```
IShuffleLayer (two inputs): t × S → t
```

`IConstantLayer` has no inputs, but can produce a tensor of either kind, so its signature is:

```
IConstantLayer: → t
```

The signature for `IShapeLayer` allows all four possible combinations `E→E`, `E→S`, `S→E`, and `S→S`, so it can be written with two independent variables:

```
IShapeLayer: t1 → t2
```

Here is the complete set of rules, which also serves as a reference for which layers can be used to manipulate shape tensors:

```
IAssertionLayer: S →
IConcatenationLayer: t × t × ...→ t
IIfConditionalInputLayer: t → t
IIfConditionalOutputLayer: t → t
IConstantLayer: → t
IActivationLayer: t → t
IElementWiseLayer: t × t → t
IFillLayer: S → t
IFillLayer: S × E × E → E
IGatherLayer: t × t → t
IIdentityLayer: t → t
IReduceLayer: t → t
IResizeLayer (one input): E → E
IResizeLayer (two inputs): E × S → E
```

```
ISelectLayer: t × t × t → t
IShapeLayer: t1 → t2
IShuffleLayer (one input): t → t
IShuffleLayer (two inputs): t × S → t
ISliceLayer (one input): t → t
ISliceLayer (two inputs): t × S → t
ISliceLayer (three inputs): t × S × S → t
ISliceLayer (four inputs): t × S × S × S → t
IUnaryLayer: t → t
all other layers: E × ... → E × ...
```

Because an output can be the input of more than one subsequent layer, the inferred "types" are not exclusive. For example, an `IConstantLayer` might feed into one use that requires an execution tensor and another use that requires a shape tensor. The output of `IConstantLayer` is classified as both and can be used in both phase 1 and phase 2 of the two-phase execution.

The requirement that the size of a shape tensor be known at build time limits how `ISliceLayer` can be used to manipulate a shape tensor. Specifically, if the third parameter, which specifies the size of the result, and is not a build-time constant, the length of the resulting tensor is unknown at build time, breaking the restriction that shape tensors have constant shapes. The slice will still work, but will incur synchronization overhead at runtime because the tensor is considered an execution tensor that has to be copied back to the CPU to do further shape calculations.

The rank of any tensor has to be known at build time. For example, if the output of `ISliceLayer` is a 1D tensor of unknown length that is used as the reshape dimensions for `IShuffleLayer`, the output of the shuffle would have unknown rank at build time, and hence such a composition is prohibited.

TensorRT's inferences can be inspected using methods `ITensor::isShapeTensor()`, which returns true for a shape tensor, and `ITensor::isExecutionTensor()`, which returns true for an execution tensor. Build the entire network first before calling these methods because their answer can change depending on what uses of the tensor have been added.

For example, if a partially built network sums two tensors, *T1* and *T2*, to create tensor *T3*, and none are yet needed as shape tensors, `isShapeTensor()` returns false for all three tensors. Setting the second input of `IShuffleLayer` to *T3* would cause all three tensors to become shape tensors because `IShuffleLayer` requires that its second optional input be a shape tensor, and if the output of `IElementWiseLayer` is a shape tensor, its inputs are too.

8.9. Shape Tensor I/O (Advanced)

Sometimes the need arises to use a shape tensor as a network I/O tensor. For example, consider a network consisting solely of an `IShuffleLayer`. TensorRT infers that the second input is a shape tensor. `ITensor::isShapeTensor` returns true for it. Because it is an input shape tensor, TensorRT requires two things for it:

- At build time: the optimization profile *values* of the shape tensor.
- At run time: the *values* of the shape tensor.

The shape of an input shape tensor is always known at build time. It is the values that must be described since they can be used to specify the dimensions of execution tensors.

The optimization profile values can be set using `IOptimizationProfile::setShapeValues`. Analogous to how min, max, and optimization dimensions must be supplied for execution tensors with runtime dimensions, min, max, and optimization values must be provided for shape tensors at build time.

The corresponding runtime method is `IExecutionContext::setTensorAddress`, which tells TensorRT where to look for the shape tensor values.

Because the inference of “execution tensor” vs “shape tensor” is based on ultimate use, TensorRT cannot infer whether a network output is a shape tensor. You must tell it using the method `INetworkDefinition::markOutputForShapes`.

Besides letting you output shape information for debugging, this feature is useful for composing engines. For example, consider building three engines, one each for sub-networks A, B, C, where a connection from A to B or B to C might involve a shape tensor. Build the networks in reverse order: C, B, and A. After constructing network C, you can use `ITensor::isShapeTensor` to determine if an input is a shape tensor, and use `INetworkDefinition::markOutputForShapes` to mark the corresponding output tensor in network B. Then check which inputs of B are shape tensors and mark the corresponding output tensor in network A.

Shape tensors at network boundaries must have type `Int32` or `Int64`. They cannot have type `Float` or `Bool`. A workaround for `Bool` is to use `Int32` for the I/O tensor, with zeros and ones, and convert to/from Bool using `IIdentityLayer`.

At runtime, whether a tensor is an I/O shape tensor can be determined via method `ICudaEngine::isShapeInferenceIO()`.

8.10. INT8 Calibration with Dynamic Shapes

To run INT8 calibration for a network with dynamic shapes, a calibration optimization profile must be set. Calibration is performed using kOPT values of the profile. Calibration input data size must match this profile.

To create a calibration optimization profile, first, construct an `IOptimizationProfile` the same way as it is done for a general optimization profile. Then set the profile to the configuration:

C++

```
config->setCalibrationProfile(profile)
```

Python

```
config.set_calibration_profile(profile)
```

The calibration profile must be valid or be `nullptr`. `kMIN` and `kMAX` values are overwritten by `kOPT`. To check the current calibration profile, use `IBuilderConfig::getCalibrationProfile`.

This method returns a pointer to the current calibration profile or nullptr if the calibration profile is unset. `getBatchSize()` calibrator method must return `1` when running calibration for a network with dynamic shapes.

Note: If the calibration optimization profile is not set, the first network optimization profile is used as a calibration optimization profile.

Chapter 9. Extending TensorRT with Custom Layers

NVIDIA TensorRT supports many types of layers and its functionality is continually extended; however, there can be cases in which the layers supported do not cater to the specific needs of a model. In such cases, TensorRT can be extended by implementing custom layers, often referred to as plugins.

TensorRT contains standard plugins that can be loaded into your application. For a list of open-source plugins, refer to GitHub: TensorRT plugins.

To use standard TensorRT plugins in your application, the `libnvinfer_plugin.so` (`nvinfer_plugin.dll` on Windows) library must be loaded, and all plugins must be registered by calling `initLibNvInferPlugins` in your application code. For more information about these plugins, refer to the `NvInferPlugin.h` file for reference.

If these plugins do not meet your needs, you can write and add your own.

9.1. Adding Custom Layers Using the C ++ API

There are four steps to ensure that your plugin is properly recognized by TensorRT:

1. Implement a plugin class by deriving from one of TensorRT's plugin base classes. Currently, the only recommended one is `IPluginV3`.
2. Implement a plugin creator class, which is tied to your plugin class, by deriving from one of TensorRT's plugin creator base classes. Currently, the only recommended one is `IPluginCreatorV3One`.
3. Register an instance of the plugin creator class with TensorRT's plugin registry.
4. Add an instance of the plugin class to a TensorRT network, either by directly using TensorRT's network APIs, or through the loading of an ONNX model by the TensorRT ONNX parser APIs.

The following sections explore each of these steps in detail.

9.1.1. Implementing a Plugin Class

You can implement a custom layer by deriving from one of TensorRT's plugin base classes.Starting in TensorRT 10.0, the only plugin interface recommended is `IPluginV3`, as others are deprecated. Therefore, this section mostly describes plugin implementation using `IPluginV3`. Refer to the Migrating V2 Plugins to IPluginV3 section for how plugins implementing V2 plugin interfaces can be migrated to `IPluginV3`.

`IPluginV3` is a wrapper for a set of *capability interfaces* that define three capabilities: core, build, and runtime.

Core capability
: Refers to plugin attributes and behaviors common to both build and runtime phases of a plugin's lifetime.

Build capability
: Refers to plugin attributes and behaviors that the plugin must exhibit for the TensorRT builder.

Runtime capability
: Refers to plugin attributes and behaviors that the plugin must exhibit for it to be executable, either during auto-tuning in the TensorRT build phase or inference in the TensorRT runtime phase.

`IPluginV3OneCore` (C++, Python), `IPluginV3OneBuild` (C++, Python), and `IPluginV3OneRuntime` (C++, Python) are the base classes that must be implemented by a `IPluginV3` plugin to display the core, build, and runtime capabilities, respectively.

9.1.2. Implementing a Plugin Creator Class

In order to use a plugin in a network, you must first register it with TensorRT's `PluginRegistry` (C++, Python). Rather than registering the plugin directly, you register an instance of a factory class for the plugin, derived from a child class of `IPluginCreatorInterface` (C++, Python). The plugin creator class also provides other information about the plugin: its name, version, and plugin field parameters.

`IPluginCreatorV3One` is the factory class for `IPluginV3`. That is, `IPluginCreatorV3One::createPlugin()`, which has the signature below, returns a plugin object of type `IPluginV3`.

C++

```
IPluginV3* createPlugin(AsciiChar const *name, PluginFieldCollection const *fc,
 TensorRTPhase phase)
```

Python

```
create_plugin(self: trt.IPluginCreatorV3, name: str, field_collection:
 trt.PluginFieldCollection, phase: trt.TensorRTPhase) -> trt.IPluginV3
```

`IPluginCreatorV3One::createPlugin()` may be called to create a plugin instance in either the build phase of TensorRT or the runtime phase of TensorRT, which is communicated by the `phase` argument of type `TensorRTPhase` (C++, Python).

- In both phases, the returned `IPluginV3` object must have a valid core capability.
- In the build phase, the returned `IPluginV3` object must have both a build and runtime capability.

- In the runtime phase, the returned `IPluginV3` object must have a runtime capability. A build capability is not required, and is ignored.

9.1.3. Registering a Plugin Creator with the Plugin Registry

There are two ways that you can register plugins with the registry:

- TensorRT provides a macro `REGISTER_TENSORRT_PLUGIN` that statically registers the plugin creator with the registry. Note that `REGISTER_TENSORRT_PLUGIN` always registers the creator under the default namespace ("").
- Dynamically register by creating your own entry-point similar to `initLibNvInferPlugins` and calling `registerCreator` on the plugin registry. This is preferred over static registration as it allows plugins to be registered under a unique namespace. This ensures that there are no name collisions during build time across different plugin libraries.

During serialization, the TensorRT engine internally stores the plugin name, plugin version, and namespace (if it exists) for all plugins, along with any plugin fields in the `PluginFieldCollection` returned by `IPluginV3OneRuntime::getFieldsToSerialize()`. During deserialization, TensorRT looks up a plugin creator with the same plugin name, version, and namespace from the plugin registry and invokes `IPluginCreatorV3One::createPlugin()` on it – the `PluginFieldCollection` which was serialized is passed back as the `fc` argument.

9.1.4. Adding a Plugin Instance to a TensorRT Network

You can add a plugin to the TensorRT network using `addPluginV3()`, which creates a network layer with the given plugin.

For example, you can add a plugin layer to your network as follows:

```
// Look up the plugin in the registry
// Cast to appropriate child class of IPluginCreatorInterface
auto creator = static_cast<IPluginCreatorV3One*>(getPluginRegistry()->getCreator(pluginName,
 pluginVersion, pluginNamespace));
PluginFieldCollection const* pluginFC = creator->getFieldNames();
// Populate the fields parameters for the plugin layer
// PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
// Create the plugin object using the layerName and the plugin meta data, for use by the
 TensorRT builder
IPluginV3 *pluginObj = creator->createPlugin(layerName, pluginData, TensorRTPhase::kBUILD);
// Add the plugin to the TensorRT network
auto layer = network.addPluginV3(inputs.data(), int(inputs.size()),  shapeInputs.data(),
 int(shapeInputs.size()), pluginObj);
… (build rest of the network and serialize engine)
// Delete the plugin object
delete pluginObj;
… (free allocated pluginData)
```

Note: The `createPlugin` method described previously creates a new plugin object on the heap and returns a pointer to it. Ensure you destroy the `pluginObj`, as shown previously, to avoid a memory leak.

When the engine is deleted, any clones of the plugin object, created during engine build, are destroyed by the engine. It is your responsibility to ensure the plugin object you created is freed after it is added to the network.

Note:

- Do not serialize all plugin parameters: only those required for the plugin to function correctly at runtime. Build time parameters can be omitted.
- If you are an automotive safety user, you must call `getSafePluginRegistry()` instead of `getPluginRegistry()`. You must also use the macro `REGISTER_SAFE_TENSORRT_PLUGIN` instead of `REGISTER_TENSORRT_PLUGIN`.

9.1.5. Example: Adding a Custom Layer with Dynamic Shapes using Using C++

Imagine that a custom layer is needed for a padding-like operation where each image in an input batch of images must be reshaped to 32 x 32. That is, the input tensor X would be of shape (B, C, H, W) and the output Y would be of shape (B, C, 32, 32). A TensorRT plugin can be written using the `IPluginV3` interface to accomplish this; let us call it `PadPlugin`.

Since a `IPluginV3` plugin must possess multiple capabilities, each defined by a separate interface, you could implement a plugin using the principle of composition or multiple inheritance. For most use cases, particularly when the coupling of build and runtime capabilities in a single class is tolerable, a multiple inheritance approach is easier.

Using multiple inheritance, `PadPlugin`, can be implemented as follows:

```
class PadPlugin : public IPluginV3, public IPluginV3OneCore, public IPluginV3OneBuild, public
 IPluginV3OneRuntime
{
 ...override inherited virtual methods.
};
```

The override of `IPluginV3::getCapabilityInterface` must return pointers to the individual capability interfaces. For each `PluginCapabilityType`, it is imperative to cast through the corresponding capability interface to remove ambiguity for the compiler.

```
IPluginCapability* PadPlugin::getCapabilityInterface(PluginCapabilityType type) noexcept
 override
{
    TRY
    {
        if (type == PluginCapabilityType::kBUILD)
        {
            return static_cast<IPluginV3OneBuild*>(this);
        }
        if (type == PluginCapabilityType::kRUNTIME)
        {
            return static_cast<IPluginV3OneRuntime*>(this);
        }
        ASSERT(type == PluginCapabilityType::kCORE);
        return static_cast<IPluginV3OneCore*>(this);
    }
    CATCH
    {
        // log error
```

```
    }
    return nullptr;
}
```

The methods that are of importance in this particular example are:

- `INetworkDefinition::addPluginV3`
- `IPluginV3OneBuild::getNbOutputs`
- `IPluginV3OneBuild::getOutputDataTypes`
- `IPluginV3OneBuild::getOutputShapes`
- `IPluginV3OneBuild::supportsFormatCombination`
- `IPluginV3OneBuild::configurePlugin`
- `IPluginV3OneRuntime::onShapeChange`
- `IPluginV3OneRuntime::enqueue`

To add the plugin to the network, `INetworkDefinition::addPluginV3` (C++, Python) can be used.

```
std::vector<ITensor*> inputs{X};

auto pluginLayer = network->addPluginV3(inputs.data(), inputs.size(), nullptr, 0, *plugin);
```

You can communicate that there is a single plugin output by overriding `IPluginV3OneBuild::getNbOutputs`.

```
int32_t PadPlugin::getNbOutputs() const noexcept override
{
    return 1;
}
```

The output will have the same data type as the input, and this can be communicated in the override of `IPluginV3OneBuild::getOutputDataTypes`.

```
int32_t PadPlugin::getOutputDataTypes(
        DataType* outputTypes, int32_t nbOutputs, DataType const* inputTypes, int32_t
 nbInputs) const noexcept override
{
    outputTypes[0] = inputTypes[0];
    return 0;
}
```

The override for `getOutputShapes` returns symbolic *expressions* for the output dimensions in terms of the input dimensions, except in the case of data-dependent output shapes, which will be covered later in Example: Adding a Custom Layer with a Data-Dependent and Shape Input-Dependent Shapes Using C++. In the current example, the first two dimensions of the output will be equal to the first two dimensions of the input, respectively, and the last two dimensions will be constants, each equal to 32. The `IExprBuilder` passed into `getOutputShapes` can be used to define constant symbolic expressions.

```
int32_t PadPlugin::getOutputShapes(DimsExprs const* inputs, int32_t nbInputs, DimsExprs
 const* shapeInputs, int32_t nbShapeInputs, DimsExprs* outputs, int32_t nbOutputs,
 IExprBuilder& exprBuilder) noexcept
{
    outputs[0].nbDims = 4;
    // first two output dims are equal to the first two input dims
    outputs[0].d[0] = inputs[0].d[0];
    outputs[0].d[1] = inputs[0].d[1];
```

```
    // The last two output dims are equal to 32
    outputs[0].d[2] = exprBuilder.constant(32);
    outputs[0].d[3] = exprBuilder.constant(32);
    return 0;
}
```

TensorRT uses `supportsFormatCombination` to ask whether a given type and format combination is accepted by the plugin, for a "connection" at a given position `pos`, given formats/types for lesser indexed connections. The interface indexes the inputs/outputs uniformly as "connections," starting at 0 for the first input, then the rest of the inputs in order, followed by numbering the outputs. In the example, the input is connection 0, and the output is connection 1.

For the sake of simplicity, the example supports only linear formats and FP32 types.

```
bool PadPlugin::supportsFormatCombination(
        int32_t pos, DynamicPluginTensorDesc const* inOut, int32_t nbInputs, int32_t
 nbOutputs) noexcept override
{
    assert(0 <= pos && pos < 2);
    return inOut[pos].desc.format == PluginFormat::kLINEAR && inOut[pos].desc.type ==
 DataType::kFLOAT;
}
```

TensorRT invokes two methods to allow the plugin to make any configuration choices before `enqueue()`, both during auto-tuning (in the engine build phase), as well as when the engine is being executed (in the runtime phase).

`IPluginV3OneBuild::configurePlugin`
Called when a plugin is being prepared for profiling (auto-tuning) but not for any specific input size. The `min`, `max`, and `opt` value of the `DynamicPluginTensorDesc` correspond to the bounds on the tensor shape and its shape for auto-tuning. The `desc.dims` field corresponds to the dimensions of the plugin specified at network creation, including any wildcards (-1) for dynamic dimensions.

`IPluginV3OneRuntime::onShapeChange`
Called during both the build-phase and runtime-phase before `enqueue()` to communicate the actual input and output shapes for the subsequent `enqueue()`. The output `PluginTensorDesc` will contain wildcards (-1) for any data-dependent dimensions specified through `getOutputShapes()`.

This plugin does not need `configurePlugin` and `onShapeChange` to do anything, so they are no-ops:

```
int32_t PadPlugin::configurePlugin(DynamicPluginTensorDesc const* in, int32_t nbInputs,
 DynamicPluginTensorDesc const* out, int32_t nbOutputs) noexcept override
{
    return 0;
}

int32_t PadPlugin::onShapeChange(PluginTensorDesc const* in, int32_t nbInputs,
 PluginTensorDesc const* out, int32_t nbOutputs) noexcept override
{
    return 0;
}
```

Finally, the override `PadPlugin::enqueue` has to do the work. Since shapes are dynamic, enqueue is handed a `PluginTensorDesc` that describes the actual dimensions, type, and format of each input and output.

```
int32_t enqueue(PluginTensorDesc const* inputDesc, PluginTensorDesc const* outputDesc, void
 const* const* inputs,
        void* const* outputs, void* workspace, cudaStream_t stream) noexcept override
{
    // populate outputs and return status code
}
```

9.1.6. Example: Adding a Custom Layer with a Data-Dependent and Shape Input-Dependent Shapes Using C++

This section shows an example of a plugin with data-dependent and shape-input dependent shapes. Note that data-dependent output shapes and adding shape inputs to a plugin are new features not present in V2 plugins.

Data-dependent Shapes (DDS)

The shape of a plugin output could depend on the values of the input tensors.

Shape inputs

A plugin could accept shape tensor inputs, besides device tensor inputs. These inputs are only visible to the plugin as arguments to `IPluginV3OneBuild::getOutputShapes()`. Therefore, their sole purpose is to aid the plugin in performing output shape calculations.

For example, `BarPlugin` is a plugin with one device input X, one shape input S, and an output Y, where:

- The first dimension of Y depends on the value of S
- The second dimension of Y is static
- The third dimension of Y is data-dependent
- The fourth dimension of Y depends on the shape of X

Similar to `PadPlugin` in the prior example, `BarPlugin` uses multiple inheritance.

To add the plugin to the network, `INetworkDefinition::addPluginV3` (C++, Python) can be used similarly. There are two additional arguments in `addPluginV3` for the specification of the shape tensor inputs, after the device tensor inputs.

```
std::vector<ITensor*> inputs{X};
std::vector<ITensor*> shapeInputs{S};

auto pluginLayer = network->addPluginV3(inputs.data(), inputs.size(), shapeInputs.data(),
 shapeInputs.size(), *plugin);
```

Note: The TensorRT ONNX parser provides an inbuilt feature to pass shape inputs to custom ops supported by `IPluginV3`-based plugins. The indices of the inputs to be interpreted as shape inputs must be indicated by a node attribute named `tensorrt_plugin_shape_input_indices` as a list of integers. For example, if the custom op has four inputs and the second and fourth inputs should be passed as shape inputs to the plugin, add a node attribute named `tensorrt_plugin_shape_input_indices` of type `onnx.AttributeProto.ints` containing the value `[1, 3]`.

In the override for `getOutputShapes`, plugins must declare both the position as well as the bounds of each data-dependent dimension of each output tensor. The bounds can be expressed in terms of a special output called a *size tensor*.

A size tensor is a scalar of either INT32 or INT64 data type, expressed through a value for auto-tuning and an upper bound; these values can either be constants or computed in terms of device input shapes or shape inputs values using `IExprBuilder`.

In this case, there is a singular data-dependent dimension, which we can represent using one size tensor. Note that any size tensor needed to express a data-dependent dimension counts as an output of the plugin; therefore, the plugin will have two outputs in total.

```
int32_t getNbOutputs() const noexcept override
{
    return 2;
}
```

Assume that output Y has the same type as the device input X and the size of the data-dependent dimension fits in INT32 (that is, the size tensor has type r). Then `BarPlugin` expresses the output data types like this:

```
int32_t getOutputDataTypes(
        DataType* outputTypes, int32_t nbOutputs, DataType const* inputTypes, int32_t
 nbInputs) const noexcept override
{
    outputTypes[0] = inputTypes[0];
    outputTypes[1] = DataType::kINT32;
    return 0;
}
```

The method `getOutputShapes` can build symbolic output shape expressions using the `IExprBuilder` passed to it. In what follows, note in particular that size tensors must be explicitly declared as 0-D.

```
int32_t BarPlugin::getOutputShapes(DimsExprs const* inputs, int32_t nbInputs, DimsExprs
 const* shapeInputs, int32_t nbShapeInputs, DimsExprs* outputs, int32_t nbOutputs,
 IExprBuilder& exprBuilder) noexcept
{
    outputs[0].nbDims = 4;
    // The first output dimension depends on the value of S.
    // The value of S is encoded as fictitious dimensions.
    outputs[0].d[0] = shapeInputs[0].d[0];
    // The third output dimension depends on the shape of X
    outputs[0].d[2] = inputs[0].d[0];
    // The second output dimension is static
    outputs[0].d[1] = exprBuilder.constant(3);

    auto upperBound = exprBuilder.operation(DimensionOperation::kPROD, *inputs[0].d[2],
 *inputs[0].d[3]);
    auto optValue = exprBuilder.operation(DimensionOperation::kFLOOR_DIV, *upperBound,
 *exprBuilder.constant(2));

    // output at index 1 is a size tensor
    outputs[1].nbDims = 0; // size tensors must be declared as 0-D
    auto sizeTensor = exprBuilder.declareSizeTensor(1, *optValue, *upperBound);

    // The fourth output dimension is data-dependent
    outputs[0].d[3] = sizeTensor;

    return 0;
}
```

The override of `supportsFormatCombination` imposes the following conditions:

- The devices input X must have `DataType::kFLOAT` or `DataType::kHALF`
- The output Y must have the same type as X
- The size tensor output has type `DataType::kINT32`

Note: Shape inputs passed to the plugin through `addPluginV3` (C++, Python) only show up as arguments to `getOutputShapes()` and are not counted or included among plugin inputs in any other plugin interface method.

```
bool BarPlugin::supportsFormatCombination(
        int32_t pos, DynamicPluginTensorDesc const* inOut, int32_t nbInputs, int32_t
 nbOutputs) noexcept override
    {
        assert(0 <= pos && pos < 3);
        auto const* in = inOut;
        auto const* out = inOut + nbInputs;

        bool typeOk{false};

        switch (pos)
        {
        case 0: typeOk = in[0].desc.type == DataType::kFLOAT || in[0].desc.type ==
 DataType::kHALF; break;
        case 1: typeOk = out[0].desc.type == in[0].desc.type; break;
        case 2: typeOk = out[1].desc.type == DataType::kINT32; break;
        }

        return inOut[pos].desc.format == PluginFormat::kLINEAR && typeOk;
    }
```

The local variables `in` and `out` here allow inspecting `inOut` by input or output number instead of connection number.

Important: The override inspects the format/type for a connection with an index less than `pos`, but must never inspect the format/type for a connection with an index greater than `pos`. The example uses `case 1` to check connection 1 against connection 0, and not use `case 0` to check connection 0 against connection 1.

`configurePlugin` and `onShapeChange` would be no ops here too; one thing to note is that in `onShapeChange`, the output's `PluginTensorDesc` will contain a wildcard (-1) for the data-dependent dimension.

Implementing `enqueue` with data-dependent output shapes is not much different from the static or dynamic shape cases. As with any other output, for an output with a data-dependent dimension, the output buffer passed to `enqueue` is guaranteed to be large enough to hold the corresponding output tensor (based on the upper-bound specified through `getOutputShapes`).

9.1.7. Example: Adding a Custom Layer with INT8 I/O Support Using C++

`PoolPlugin` is a plugin to demonstrate how to `addINT8` I/O for a custom pooling layer using `IPluginV3`. `PoolPlugin` multiply inherits from `IPluginV3`, `IPluginV3OneCore`, `IPluginV3OneBuild`, and `IPluginV3OneRuntime` similar to the `PadPlugin` and `BarPlugin` examples above.

The main methods that affect INT8 I/O are:

- `supportsFormatCombination`
- `configurePlugin`

The override for `supportsFormatCombination` must indicate which INT8 I/O combination is allowed. The usage of this interface is similar to Example: Adding a Custom Layer with Dynamic Shapes using Using C++. In this example, the supported I/O tensor format is linear CHW with FP32, FP16, BF16, FP8, or INT8 data type, but the I/O tensor must have the same data type.

```
bool PoolPlugin::supportsFormatCombination(
        int32_t pos, DynamicPluginTensorDesc const* inOut, int32_t nbInputs, int32_t
 nbOutputs)  noexcept override
{
    assert(nbInputs == 1 && nbOutputs == 1 && pos < nbInputs + nbOutputs);
    bool condition = inOut[pos].desc.format == PluginFormat::kLINEAR;
    condition &= (inOut[pos].desc.type == DataType::kFLOAT ||
                  inOut[pos].desc.type == DataType::kHALF ||
              inOut[pos].desc.type == DataType::kBF16 ||
                   inOut[pos].desc.type == DataType::kFP8 ||
                  inOut[pos].desc.type == DataType::kINT8);
    condition &= inOut[pos].desc.type == inOut[0].desc.type;
    return condition;
}
```

Important:

- If INT8 calibration must be used with a network with INT8 I/O plugins, the plugin must support FP32 I/O as TensorRT uses FP32 to calibrate the graph.
- If the FP32 I/O variant is not supported or INT8 calibration is not used, all required INT8 I/O tensors scales must be set explicitly.
- Calibration cannot determine the dynamic range of a plugin internal tensors. Plugins that operate on quantized data must calculate their own dynamic range for internal tensors.
- A plugin can be designed to accept both FP8 and INT8 I/O types, although note that in TensorRT 9.0 the builder does not allow networks that mix INT8 and FP8.

Information communicated by TensorRT through `configurePlugin` or `onShapeChange` can be used to obtain information about the pooling parameters and the input and output scales. These can be stored as member variables, serialized and then deserialized to be used during inference.

```
int32_t PoolPlugin::configurePlugin(DynamicPluginTensorDesc const* in, int32_t nbInputs,
 DynamicPluginTensorDesc const* out, int32_t nbOutputs) noexcept override
{
    ...
```

```
    mPoolingParams.mC = in.desc.d[1];
    mPoolingParams.mH = in.desc.d[2];
    mPoolingParams.mW = in.desc.d[3];
    mPoolingParams.mP = out.desc.d[2];
    mPoolingParams.mQ = ou.desc.d[3];
    mInHostScale = in[0].desc.scale >= 0.0F ? in[0].desc.scale : -1.0F;
    mOutHostScale = out[0].desc.scale >= 0.0F ? out[0].desc.scale : -1.0F;
}
```

INT8 I/O scales per tensor have been obtained from `PluginTensorDesc::scale`.

9.2. Adding Custom Layers using the Python API

Prior to TensorRT 9.0, custom layer implementations could only be done through the C ++ API; adding such a plugin to a TensorRT network in Python required loading a library containing the plugin and accessing the plugin creator through the plugin registry. It is now possible to implement custom layers entirely within Python, with no additional C++ code.

Implementing a plugin in Python is similar to C++ in that an implementation of `IPluginV3` and `IPluginCreatorV3One` is necessary. Furthermore, interface methods in Python have mostly similar APIs to their C++ counterparts; most differences are minor and self-explanatory.

The following list includes a few selected changes. More involved differences are described in subsequent subsections in detail.

- The following plugin APIs have been omitted in favor of reading/writing to an appropriately named attribute.

Class	Method	Replaced with Attribute
`IPluginV3OneCore`	`getPluginName()`	`plugin_name[str]`
`IPluginV3OneCore`	`getPluginNamespace()`	`plugin_namespace [str]`
`IPluginV3OneCore`	`getPluginVersion()`	`plugin_version [str]`
`IPluginV3OneBuild`	`getNbOutputs()`	`num_outputs [int]`
`IPluginV3OneBuild`	`getTimingCacheID()`	`timing_cache_id [str]`
`IPluginV3OneBuild`	`getMetadataString()`	`metadata_string [str]`
`IPluginV3OneBuild`	`getFormatCombinationLimit()`	`format_combination_limit [int]`
`IPluginCreatorV3One`	`getPluginNamespace()`	`plugin_namespace [str]`
`IPluginCreatorV3One`	`getFieldNames()`	`field_names [PluginFieldCollection]`
`IPluginCreatorV3One`	`getPluginName()`	`name [str]`
`IPluginCreatorV3One`	`getPluginVersion()`	`plugin_version [str]`

- Some methods have default implementations, these can be left unimplemented and the default behaviors outlined below will take effect:

```
class trt.IPluginV3:
    def destroy(self):
```

```
        pass

class trt.IPluginV3OneBuild:
    def get_valid_tactics(self):
        return []

    def get_workspace_size(self, input_desc, output_desc):
        return 0
```

- Methods that must return integer status codes in `IPluginV3OneBuild` and `IPluginV3OneRuntime` should raise exceptions in Python instead. For example:

  C++

  ```
  int32_t configurePlugin(DynamicPluginTensorDesc const* in, int32_t nbInputs,
   DynamicPluginTensorDesc const* out, int32_t nbOutputs)
  ```

  Python

  ```
  configure_plugin(self: trt.IPluginV3OneBuild, in: List[trt.DynamicPluginTensorDesc],
   out: List[trt.DynamicPluginTensorDesc]) -> None
  ```

  For example, you can raise a `ValueError` during `enqueue` if an input has an illegal value.

- The Python API `IPluginV3.destroy()` has no direct equivalent in the C++ API. Python plugins are expected to perform any functionality that would be performed in a `IPluginV3` C++ destructor within the `IPluginV3.destroy()` method.

For full examples demonstrating Python plugins, refer to the python_plugin sample.

9.2.1. Registration of a Python Plugin

Python plugins must be dynamically registered through the `IPluginRegistry.register_creator()` API. There is no analog to the `REGISTER_TENSORT_PLUGIN` available for static registration.

9.2.2. Building and Running TensorRT Engines Containing Python Plugins

It is possible to build TensorRT engines using Python-based plugins. However, it is currently not possible to run such engines outside of Python, since the plugin must be available in the scope where the engine is being deserialized. For example, you cannot use a tool like `trtexec` directly.

9.2.3. Implementing `enqueue` of a Python Plugin

The API for `IPluginV3OneRuntime::enqueue()` in C++ and Python are as follows:

C++

```
int32_t enqueue(PluginTensorDesc const *inputDesc, PluginTensorDesc const *outputDesc,
 void const *const *inputs, void *const *outputs, void *workspace, cudaStream_t stream)
```

Python

```
enqueue(self: trt.IPluginV3OneRuntime, input_desc: List[trt.PluginTensorDesc],
 output_desc: List[trt.PluginTensorDesc], inputs: List[int], outputs: List[int],
 workspace: int, stream: int) -> None
```

Here `inputs`, `outputs`, and `workspace` are passed-in as `intptr_t` casts of the respective device pointers. Similarly, `stream` is an `intptr_t` cast of a pointer to the CUDA stream

handle. There is flexibility within Python on how to read from and write to these buffers, and can be achieved depending on the particular use case. For example, with CUDA Python, this is quite simple since `cuda.cuLaunchKernel` accepts `int` representing the pointers wrapped in NumPy arrays:

```
d_input = np.array([inputs[0]], dtype=np.uint64)
d_output = np.array([outputs[0]], dtype=np.uint64)
stream_ptr = np.array([stream], dtype=np.uint64)
args = [d_input,  d_output]
kernel_args = np.array([arg.ctypes.data for arg in args], dtype=np.uint64)
…
checkCudaErrors(cuda.cuLaunchKernel(_float_kernel,
                                    num_blocks, 1, 1,
                                    block_size, 1, 1,
                                    0,
                                    stream_ptr,
                                    kernel_args , 0))
```

9.2.4. Translating Device Buffers/CUDA Stream Pointers in `enqueue` to other Frameworks

It is possible to construct CuPy arrays on top of device buffers using CuPy's `UnownedMemory` class.

```
def enqueue(self, input_desc, output_desc, inputs, outputs, workspace, stream):
...
inp_dtype = trt.nptype(input_desc[0].type)
inp_mem = cp.cuda.UnownedMemory(
    inputs[0], volume(input_desc[0].dims) * cp.dtype(inp_dtype).itemsize, self
)
out_mem = cp.cuda.UnownedMemory(
    outputs[0],
    volume(output_desc[0].dims) * cp.dtype(inp_dtype).itemsize,
    self,
)

inp_ptr = cp.cuda.MemoryPointer(inp_mem, 0)
out_ptr = cp.cuda.MemoryPointer(out_mem, 0)

inp = cp.ndarray((volume(input_desc[0].dims)), dtype=inp_dtype, memptr=inp_ptr)
out = cp.ndarray((volume(output_desc[0].dims)), dtype=inp_dtype, memptr=out_ptr)
```

If needed, `torch.as_tensor()` can then be used to construct a Torch array:

```
# inp_d = cp.ndarray(tuple(input_desc[0].dims), dtype=inp_dtype, memptr=inp_ptr)
inp_t = torch.as_tensor(inp_d, device='cuda')
```

Similarly, CuPy stream handles can be constructed from the passed-in `stream` pointer through CuPy's `ExternalStream` class.

```
cuda_stream = cp.cuda.ExternalStream(stream)
```

9.2.5. Automatic Downcasting

TTensorRT Python bindings will do automatic downcasting for custom types written in Python implementing interfaces like `IPluginCreatorV3One` or `IPluginResource`. For example, take the following method from `IPluginRegistry` as an example:

```
get_creator(self: trt.IPluginRegistry, name: string, version: string,
namespace: string = "") -> trt.IPluginCreatorInterface
```

The return type is indicated as `IPluginCreatorInterface`. However, in practice, if you were to write a class `MyPluginCreator` implementing `IPluginCreatorV3One` (which in

turn implements `IPluginCreatorInterface`), the `get_creator` method will return an automatically downcasted type of `MyPluginCreator`.

This extends to `trt.IPluginRegistry.all_creators`, which is a `List[trt.IPluginCreatorInterface]`. If you had registered a plugin creator of type `MyPluginCreator` as well as another of type `MyOtherPluginCreator`, both of those plugin creators will be present as those respective types in the list.

9.2.6. Example: Adding a Custom Layer to a TensorRT Network Using Python

The Python API has a function called `add_plugin_v3` that enables you to add a plugin node to a network. The following example illustrates this. It creates a simple TensorRT network and adds a leaky ReLU plugin node by looking up the TensorRT plugin registry.

```
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger()

trt.init_libnvinfer_plugins(TRT_LOGGER, '')
def get_trt_plugin(plugin_name, plugin_version, plugin_namespace):
    plugin = None
    plugin_creator = trt.get_plugin_registry().get_creator(plugin_name, plugin_version,
 plugin_namespace)
    # trt will automatically downcast to IPluginCreator or IPluginCreatorInterface
    # Can inspect plugin_creator.interface_info to make sure
    if plugin_creator is not None:
        lrelu_slope_field = trt.PluginField("epsilon", np.array([0.00000001],
 dtype=np.float32), trt.PluginFieldType.FLOAT32)
        field_collection = trt.PluginFieldCollection([lrelu_slope_field])
        plugin = plugin_creator.create_plugin(name=plugin_name,
 field_collection=field_collection, phase=trt.TensorRTPhase.BUILD)
    return plugin

def main():
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network()
    config = builder.create_builder_config()
    config.max_workspace_size = 2**20
    input_layer = network.add_input(name="input_layer", dtype=trt.float32, shape=(1, 1))
    plugin = network.add_plugin_v3(inputs=[input_layer], shape_inputs=[],
 plugin=get_trt_plugin("MY_PLUGIN", "1", ""))
    plugin.get_output(0).name = "outputs"
    network.mark_output(plugin.get_output(0))
```

9.3. Enabling Timing Caching and Using Custom Tactics

`IPluginV3` provides more control over the profiling of custom layers which were not available with V2 plugins and earlier. One such feature is enabling timing caching. If a TensorRT network contains multiple instances of the same plugin, identically configured (for example, same plugin attribute values) and handling identical input output shapes and types, then it would make sense to time (measure latency) of only one instance,

cache the latency, and skip timing the rest of the instances. This would potentially enable large savings in terms of engine build time.

Timing caching for `IPluginV3` plugins is an opt-in feature; to opt-in, the plugin must advertise a non-null timing cache ID.

C++

```
char const* FooPlugin::getTimingCacheID() noexcept override
{
    // return nullptr to disable timing caching (default behavior)
    // return non-null string to enable timing caching
}
```

Python

```
def FooPlugin(trt.IPluginV3, trt.IPluginV3OneBuild, ...):
    def __init__(self):
        # set to None to disable timing caching
        self.timing_cache_id = value
```

Note the following regarding the timing cache ID:

- The user-provided timing cache ID should be thought of as a suffix to a larger timing cache ID; TensorRT automatically forms a prefix by considering the input/output shape and format information of the plugin. In most cases, the user-provided timing cache ID could consist of plugin attributes and their values.
- It must only reflect the creation state of the plugin. That is, it must not evolve after the plugin has been created.

For V2 plugins, TensorRT only times the plugin for any (multiple) type/format combinations it claims to support. With `IPluginV3`, plugins also have the ability to make sure custom tactics are timed, and the fastest tactic is used by TensorRT. For example, the plugin may have one of two kernels to compute the output, and it may not be possible to predict which one would be fastest on a specific platform, and for specific input/output shapes and formats. It is possible to ask TensorRT to time the plugin for each tactic for each format combination, figure out the fastest such configuration and use that during inference.

Note:

- TensorRT may choose not to time the plugin at all if it only supports one type/format combination and either does not use custom tactics or only advertises one custom tactic.
- For `IPluginV3OneBuild`, TensorRT times a maximum of `getFormatCombinationLimit()` type/format combinations *for each tactic*; override this method to increase/decrease this limit depending on need.

To get started, advertise the custom tactics to TensorRT:

C++

```
int32_t FooPlugin::getNbTactics() noexcept override
{
    return 2; // return 0 to disable custom tactics (default behavior)
}

int32_t FooPlugin::getValidTactics(int32_t* tactics, int32_t nbTactics) noexcept override
{
    tactics[0] = 1;
    tactics[1] = 2;
    return 0;
```

```
}
```

Python

```
def get_valid_tactics(self):
    return [1, 2] # return empty vector to disable custom tactics (default behavior)
```

Any strictly positive integer could be used as a custom tactic value (0 is reserved as the default tactic by TensorRT).

When the plugin is being timed, it is guaranteed that `configurePlugin()` is called with the current input/output format combination before `getValidTactics()` is called. Therefore, it is possible to advertise a different set of tactics per each input/output format combination. For example, for a plugin which supports both FP32 and FP16, tactic 1 may be restricted to only FP16 while supporting both tactics 1 and 2 for FP32.

During the engine build, when auto-tuning the plugin, TensorRT will communicate the tactic to use for the subsequent `enqueue()` by invoking `IPluginV3OneRuntime::setTactic` (C++, Python). When an engine is deserialized, TensorRT will invoke `setTactic` once after the plugin has been created to communicate to the plugin the best tactic chosen. Note that even if custom tactics are not used, `setTactic` will be called with the default tactic value `0`.

9.4. Sharing Custom Resources Among Plugins

Starting in TensorRT 10.0, a key-value store is associated with the plugin registry, which can be used to store user-implemented `IPluginResource` (C++, Python) objects against a string key. This functionality can be used to share state or some resource among different plugins. Note that it is not tied to `IPluginV3` (or even to plugin interfaces).

Let us explore an example.

9.4.1. Example: Sharing Weights Downloaded Over a Network Among Different Plugins

Assume that several plugins need access to the same weights `W`. Due to licensing restrictions, you may prefer that these weights be downloaded when the engine is being run. But due to the large size of `W`, it is also desirable that only one copy is downloaded, and this copy shared among all plugins which need access.

1. Implement `SharedWeights` class which implements `IPluginResource`.
2. Each plugin that requires access to the weights requests an instance of initialized (downloaded) `SharedWeights` by calling `IPluginRegistry::acquirePluginResource(...)`.

   C++

   ```
   IPluginResource* acquirePluginResource(char const* key, IPluginResource* resource)
   ```

   (C++, Python)

Python

```
acquire_plugin_resource(self: trt.IPluginRegistry, key: str, resource:
 trt.IPluginResource) -> trt.IPluginResource
```

The first time `acquirePluginResource` is called against a particular `key`, TensorRT registers *a clone* of the provided plugin `resource` instead of the object passed as `resource`. That is, the object that is registered is the one obtained by invoking `resource->clone()`. Therefore, it is best practice to only initialize clones – in this case, the weight download can be done in `IPluginResource::clone()`.

3. After each plugin is done using the weights, it can call `IPluginRegistry::releasePluginResource()` to signal that it no longer wishes to use the weights.

C++

```
int32_t releasePluginResource(char const* key)
```

Python

```
release_plugin_resource(self: trt.IPluginRegistry, key: str) -> None
```

TensorRT performs reference counting on the `acquirePluginResource` and `releasePluginResource` calls made against a particular key, and will call `IPluginResource::release()` if and when the reference count reaches zero. In this example, this functionality can be leveraged to free up the memory used by the weights when all plugins have finished using it.

4. Finally, the `SharedWeights` class can be implemented as follows:

```
class SharedWeights : public IPluginResource
{
public:
    SharedWeights(bool init = false)
    {
        if(init)
        {
            PLUGIN_CHECK(cudaMalloc((void**) &cloned->mWeights, ...));
        }
    }

    int32_t release() noexcept override
    {
        TRY
        {
            if (mWeights != nullptr)
            {
                PLUGIN_CHECK(cudaFree(mWeights));
                mWeights = nullptr;
            }
        }
        CATCH
        {
            return -1;
        }
        return 0;
    }

    IPluginResource* clone() noexcept override
    {
        TRY
        {
            auto cloned = std::make_unique<SharedWeights>(/* init */ true);
            //
            // Download the weights
            //
```

```
            // Copy to device memory
            PLUGIN_CHECK(cudaMemcpy(cloned->mWeights, ...));
        }
        CATCH
        {
            return nullptr;
        }
        return cloned.release();
    }

    ~SharedWeights() override
    {
        if(mWeights)
        {
            release();
        }
    }

    float* mWeights{nullptr};
};
```

Say `FooPlugin` needs access to the weights. It can request the weights when it is being made ready for inference. This can be done in `IPluginV3OneRuntime::onShapeChange`, which will be called at least once for plugins about to be `enqueue()` during both the build phase and runtime phase.

```
int32_t onShapeChange(
    PluginTensorDesc const* in, int32_t nbInputs, PluginTensorDesc const* out, int32_t
 nbOutputs) noexcept override
{
    SharedWeights w{};
    mW = static_cast<SharedWeights*>(getPluginRegistry()->acquirePluginResource("W",
 &w))->mWeights;
    return 0;
}
```

The acquired weights (`mW`) can then be used in the subsequent `enqueue()`. To wrap up, the plugin can signal intent to release in its destructor (note that there is no separate release resource routine similar to `IPluginV2DynamicExt::terminate()` in `IPluginV3`).

```
~FooPlugin() override
{
    TRY
    {
        PLUGIN_CHECK(getPluginRegistry()->releasePluginResource("W"));
    }
    CATCH
    {
        // Error handling
    }
}
```

Essentially the same code above can be used by all plugins requiring access to the weights. The availability and proper freeing of the weights will be ensured by the reference counting mechanism.

9.5. Using Custom Layers When Importing a Model with a Parser

The ONNX parser automatically attempts to import unrecognized nodes as plugins. If a plugin with the same `op_type` as the node is found in the plugin registry, the parser forwards the attributes of the node to the plugin creator as plugin field parameters in order to create the plugin. By default, the parser uses `"1"` as the plugin version and `""` as the plugin namespace. This behavior can be overridden by setting a `plugin_version` and `plugin_namespace` string attribute in the corresponding ONNX node.

In some cases, you might want to modify an ONNX graph before importing it into TensorRT. For example, to replace a set of ops with a plugin node. To accomplish this, you can use the ONNX GraphSurgeon utility. For details on how to use ONNX-GraphSurgeon to replace a subgraph, refer to this example.

For more examples, refer to the onnx_packnet sample.

9.6. Plugin API Description

All new plugins should derive from both `IPluginCreatorV3One` and `IPluginV3` classes. In addition, new plugins should also be registered in the plugin registry, either dynamically by using `IPluginRegistry::registerCreator()` or statically using the `REGISTER_TENSORRT_PLUGIN(...)` macro. Custom plugin libraries can also consider implementing an `init` function equivalent to `initLibNvInferPlugins()` to perform bulk registration.

Note: Automotive safety users must use the `REGISTER_SAFE_TENSORRT_PLUGIN(...)` macro instead of `REGISTER_TENSORRT_PLUGIN(...)`.

9.6.1. `IPluginV3` API Description

The following section describes the functions of the `IPluginV3`and by extension `IPluginV3OneCore`, `IPluginV3OneBuild` and `IPluginV3OneRuntime`.

Since an `IPluginV3` object consists of different capabilities, `IPluginV3::getCapabilityInterface` may be called at anytime during its lifetime. An `IPluginV3` object added for the build phase must return a valid capability interface for all capability types: core, build and runtime. The build capability may be omitted for objects added for the runtime phase.

There are a few methods used to request identifying information about the plugin. They may also be called during any stage of the plugin's lifetime.

`IPluginV3OneCore::getPluginName`
Used to query for the plugin's name.

`IPluginV3OneCore::getPluginVersion`
Used to query for the plugin's version.

`IPluginV3OneCore::getPluginNamespace`
Used to query for the plugin's namespace.

`IPluginV3OneBuild::getMetadataString`
Used to query for a string representation of any metadata associated with the plugin, such as the values of its attributes.

To connect a plugin layer to neighboring layers and set up input and output data structures, the builder checks for the number of outputs and their shapes by calling the following plugins methods:

`IPluginV3OneBuild::getNbOutputs`
Used to specify the number of output tensors.

`IPluginV3OneBuild::getOutputShapes`
Used to specify the shapes of output as a function of the input shapes or constants. The exception is data-dependent shapes where an upper-bound and optimal tuning value is specified.

`IPluginV3OneBuild::supportsFormatCombination`
Used to check if a plugin supports a given data type and format combination.

`IPluginV3OneBuild::getOutputDataType`
Used to get the data types of the output tensors. The returned data types must have a format that is supported by the plugin.

Plugin layers can support the following data formats:

- `LINEAR` single-precision (FP32), half-precision (FP16), brain floating-point (BF16), 8-bit floating-point E4M3 (FP8), integer (INT8), and integer (INT32) tensors
- `CHW32` single-precision (FP32) and integer (INT8) tensors
- `CHW2`, `HWC8`,`HWC16`, and `DHWC8` half-precision (FP16) tensors
- `CHW4` half-precision (FP16), and integer (INT8) tensors
- HWC8, HWC4, NDHWC8, NC2HW brain floating-point (BF16) tensors

The formats are counted by `PluginFormat`.

Plugins that do not compute all data in place and need memory space in addition to input and output tensors can specify the additional memory requirements with the `getWorkspaceSize` method, which is called by the builder to determine and preallocate scratch space.

At build time, to discover optimal configurations, the layer is configured, executed, and destroyed. After the optimal configuration is selected for a plugin, during inference, the chosen tactic and concrete shape/format information (except for data-dependent dimensions) is communicated to the plugin, and it is executed as many times as needed for the lifetime of the inference application, and finally destroyed when the engine is destroyed.

These steps are controlled by the builder and runtime using the following plugin methods. Methods also called during inference are indicated by (*) – all others are only called by the builder.

`IPluginV3OneBuild::attachToContext*`
Used to request a plugin clone to be attached to an `ExecutionContext` and also to provide the opportunity for the plugin to access any context-specific resources.

`IPluginV3OneBuild::getTimingCacheId`
Used to query for any timing cached ID that may be used by TensorRT. Enables timing caching if provided (disabled by default).

`IPluginV3OneBuild::getValidTactics`
Used to query for any custom tactics the plugin may choose to use. The plugin will be profiled for each such tactic up to a maximum indicated by `IPluginV3OneBuild::getFormatCombinationLimit()`.

`IPluginV3OneBuild::getFormatCombinationLimit`
Used to query for the maximum number of format combinations that may be timed for each tactic (for the default tactic `0` if no custom tactics are advertised).

`IPluginV3OneRuntime::setTactic*`
Communicates the tactic to be used during the subsequent `enqueue()`. If no custom tactics were advertised, this would always be `0`.

`IPluginV3OneBuild::configurePlugin`
Communicates the number of inputs and outputs, and their shapes, data types, and formats. The min, opt, and max of each input or output's `DynamicPluginTensorDesc` correspond to the `kMIN`, `kOPT`, and `kMAX` value of the optimization profile that the plugin is being currently profiled for, with the `desc.dims` field corresponding to the dimensions of plugin inputs specified at network creation. Wildcard dimensions may exist during this phase in the `desc.dims` field.

At this point, the plugin may set up its internal state and select the most appropriate algorithm and data structures for the given configuration.

`IPluginV3OneRuntime::onShapeChange*`
Communicates the number of inputs and outputs, and their shapes, data types and formats. The dimensions are concrete, except if data-dependent dimensions exist, which will be indicated by wildcards.

`IPluginV3OneRuntime::enqueue*`
Encapsulates the actual algorithm and kernel calls of the plugin and provides pointers to input, output, and scratch space, and the CUDA stream to be used for kernel execution.

`IPluginV3::clone`
This is called every time a new builder, network, or engine is created that includes this plugin layer. It must return a new plugin object with the correct parameters.

After the builder completes profiling, before the engine is serialized `IPluginV3OneRuntime::getFieldsToSerialize` is called to query for any plugin fields that must be serialized into the engine. These are expected to be data that the plugin needs to properly function during the inference stage once the engine has been deserialized.

9.6.2. `IPluginCreatorV3One` API Description

The following methods in the `IPluginCreatorV3One` class are used to find and create the appropriate plugin from the plugin registry:

`getPluginName`
This returns the plugin name and should match the return value of `IPluginV3OneCore::getPluginName`.

getPluginVersion
Returns the plugin version. For all internal TensorRT plugins, this defaults to `1`.

getPluginNamespace
Returns the plugin namespace. Default can be "".

getFieldNames
To successfully create a plugin, it is necessary to know all the field parameters of the plugin. This method returns the `PluginFieldCollection` struct with the `PluginField` entries populated to reflect the field name and `PluginFieldType` (the data should point to `nullptr`).

createPlugin
This method is used to create a plugin: it is passed a `PluginFieldCollection` and a `TensorRTPhase` argument.

During engine deserialization, TensorRT calls this method with the `TensorRTPhase` argument set to `TensorRTPhase::kRUNTIME` and the `PluginFieldCollection` populated with the same `PluginFields` as in the one returned by `IPluginV3OneRuntime::getFieldsToSerialize()`. TensorRT takes ownership of plugin objects returned by `createPlugin` in this case.

You may also invoke `createPlugin` to produce plugin objects to add to a TensorRT network. In this case, it is recommended to set the phase argument to `TensorRTPhase::kBUILD`. The data passed with the `PluginFieldCollection` should be allocated by the caller and eventually freed by the caller before the program is destroyed. The ownership of the plugin object returned by the `createPlugin` function is passed to the caller and must be destroyed as well.

9.7. Migrating V2 Plugins to IPluginV3

`IPluginV2` and `IPluginV2Ext` have been deprecated since TensorRT 8.5 and `IPluginV2IOExt` and `IPluginV2DynamicExt` are deprecated in TensorRT 10.0. Therefore, new plugins should target `IPluginV3` and old ones refactored.

Keep in mind the following key points when migrating a `IPluginV2DynamicExt` plugin to `IPluginV3`:

- The plugin creator associated with the plugin must be migrated to `IPluginCreatorV3One`, which is the factory class for `IPluginV3` (`IPluginCreator` is the factory class for `IPluginV2` derivatives). This simply consists of migrating `IPluginCreator::deserializePlugin`. Refer to the Plugin Serialization and Deserialization section for more information.
- There is no equivalent to `IPluginV2::initialize()`, `IPluginV2::terminate()` and `IPluginV2::destroy()` in `IPluginV3`. Refer to the Plugin Initialization and Termination section for more information.
- There is no equivalent to `IPluginV2Ext::detachFromContext()` in `IPluginV3`. Refer to the Accessing Context-Specific Resources Provided by TensorRT for more information.
- `IPluginV3OneRuntime::attachToContext()` is markedly different from `IPluginV2Ext::attachToContext()`, both in terms of arguments and behavior.

Refer to the Accessing Context-Specific Resources Provided by TensorRT for more information.

- In `IPluginV3`, plugin serialization is through a `PluginFieldCollection` that gets passed to TensorRT by `IPluginV3OneRuntime::getFieldsToSerialize()` and deserialization is through the same `PluginFieldCollection` that gets passed back by TensorRT to `IPluginCreatorV3One::createPlugin(...)`. Refer to the Plugin Serialization and Deserialization section for more information.
- The `IPluginV3` equivalents of void return methods in `IPluginV2DynamicExt` will expect an integer status code as a return value (for example, `configurePlugin`).
- `supportsFormatCombination` and `getWorkspaceSize` get dynamic tensor descriptors (`DynamicPluginTensorDesc`) instead of static descriptors (`PluginTensorDesc`).
- `IPluginV2DynamicExt::getOutputDimensions()` becomes `IPluginV3OneBuild::getOutputShapes()`, and changes to an output parameter signature instead of return value. Also, it shifts from a per-output index querying to one-shot querying. A similar transition applies from `IPluginV2Ext::getOutputDataType` to `IPluginV3OneBuild::getOutputDataTypes`.

9.7.1. Plugin Initialization and Termination

`IPluginV2` provided several APIs for plugin initialization and termination: namely, `IPluginV2::initialize()`, `IPluginV2::terminate()`, and `IPluginV2::destroy()`. In `IPluginV3`, plugins are expected to be constructed in an initialized state; if your V2 plugin had any lazy initialization in `initialize`, it can be deferred to `onShapeChange` or `configurePlugin`. Any resource release or other termination logic in `IPluginV2::terminate()` or `IPluginV2::destroy()` can be moved to the class destructor. The exception to this is in the Python API; `IPluginV3.destroy()` is provided as an alternative for a C++ like destructor.

9.7.2. Accessing Context-Specific Resources Provided by TensorRT

`IPluginV2Ext::attachToContext()` provided plugins access to context-specific resources; namely the GPU allocator, and cuDNN and cuBLAS handles. `IPluginV3OneRuntime::attachToContext()` is meant to provide a similar service to plugins, but it instead provides a `IPluginResourceContext`, which in turn exposes resources that plugins may request.

In a departure from `IPluginV2Ext::attachToContext()`, cuDNN and cuBLAS handles are no longer provided by `IPluginResourceContext`; any plugins which depended on those should migrate to initialize their own cuDNN and cuBLAS resources. If sharing cuDNN/cuBLAS resources among plugins is preferred, you can utilize the functionality provided by `IPluginResource` and the plugin registry's key-value store to accomplish this. Refer to the Sharing Custom Resources Among Plugins for more information.

`IPluginV3OneRuntime::attachToContext(...)` is a clone-and-attach operation. It is asked to clone the entire `IPluginV3` object – not just the runtime capability. Therefore, if implemented as a separate class, the runtime capability object may need to hold a reference to the `IPluginV3` object of which it is a part of.

Any context-specific resource obtained through `IPluginResourceContext` may be used until the plugin is destroyed. Any termination logic implemented in `IPluginV2Ext::detachFromContext()` may therefore be moved to the plugin destructor.

9.7.3. Plugin Serialization and Deserialization

For V2 plugins, serialization and deserialization was determined by the implementation of `IPluginV2::serialize`, `IPluginV2::getSerializationSize` and `IPluginCreator::deserializePlugin`; these have been replaced by `IPluginV3OneRuntime::getFieldsToSerialize` and `IPluginCreatorV3One::createPlugin`. Note that the workflow has shifted from writing to/reading from a raw buffer, to constructing and parsing a `PluginFieldCollection`.

The serialization of types defined in `PluginFieldType` is handled by TensorRT. Custom types can be serialized as `PluginFieldType::kUNKNOWN`. For example:

```
struct DummyStruct
{
    int32_t a;
    float b;
};

DummyPlugin()
{
    // std::vector<nvinfer1::PluginField> mDataToSerialize;
    // int32_t mIntValue;
    // std::vector<float> mFloatVector;
    // DummyStruct mDummyStruct;
    mDataToSerialize.clear();
    mDataToSerialize.emplace_back(PluginField("intScalar", &mIntValue,
 PluginFieldType::kINT32, 1));
    mDataToSerialize.emplace_back(PluginField("floatVector", mFloatVector.data(),
 PluginFieldType::kFLOAT32, mFloatVector.size()));
    mDataToSerialize.emplace_back(PluginField("dummyStruct", &mDummyStruct,
 PluginFieldType::kUNKNOWN, sizeof(DummyStruct)));
    mFCToSerialize.nbFields = mDataToSerialize.size();
    mFCToSerialize.fields = mDataToSerialize.data();
}

nvinfer1::PluginFieldCollection const* DummyPlugin::getFieldsToSerialize() noexcept override
{
    return &mFCToSerialize;
}
```

9.7.4. Migrating Older V2 Plugins to IPluginV3

If migrating from `IPluginV2` or `IPluginV2Ext` to `IPluginV3`, it is easier to migrate first to `IPluginV2DynamicExt` and then follow the guidelines above to migrate to `IPluginV3`. The new features in `IPluginV2DynamicExt` are as follows:

```
virtual DimsExprs getOutputDimensions(int outputIndex, const DimsExprs* inputs, int nbInputs,
 IExprBuilder& exprBuilder) = 0;

virtual bool supportsFormatCombination(int pos, const PluginTensorDesc* inOut, int nbInputs,
 int nbOutputs) = 0;

virtual void configurePlugin(const DynamicPluginTensorDesc* in, int nbInputs, const
 DynamicPluginTensorDesc* out, int nbOutputs) = 0;

virtual size_t getWorkspaceSize(const PluginTensorDesc* inputs, int nbInputs, const
 PluginTensorDesc* outputs, int nbOutputs) const = 0;
```

```
virtual int enqueue(const PluginTensorDesc* inputDesc, const PluginTensorDesc* outputDesc,
 const void* const* inputs, void* const* outputs, void* workspace, cudaStream_t stream) = 0;
```

Guidelines for migration to `IPluginV2DynamicExt` are

- `getOutputDimensions` implements the expression for output tensor dimensions given the inputs.
- `supportsFormatCombination` checks if the plugin supports the format and datatype for the specified I/O.
- `configurePlugin` mimics the behavior of equivalent `configurePlugin` in `IPluginV2Ext` but accepts tensor descriptors.
- `getWorkspaceSize` and `enqueue` mimic the behavior of equivalent APIs in `IPluginV2Ext` but accept tensor descriptors.

9.8. Coding Guidelines for Plugins

Memory Allocation

Memory allocated in the plugin must be freed to ensure no memory leak. If resources are acquired in the plugin constructor or at a later stage like `onShapeChange`, they must be released, possibly in the plugin class destructor.

Another option is to request any additional workspace memory required through `getWorkspaceSize`, which will then be available during `enqueue`.

Add Checks to Ensure Proper Configuration and Validate Inputs

A common source for unexpected plugin behavior is improper configuration (for example, invalid plugin attributes) and invalid inputs. As such, it is good practice to add checks/assertions during the initial plugin development for cases where the plugin is not expected to work. The following are places where checks could be added:

- `createPlugin`: Plugin attributes checks
- `configurePlugin/onShapeChange`: Input dimension checks
- `enqueue`: Input value checks

Return Null at Errors for Methods That Creates a New Plugin Object

Methods like `createPlugin`, `clone`, and `attachToContext` may be expected to create and return new plugin objects. In these methods, make sure a null object (`nullptr` in C++) is returned in case of any error or failed check. This ensures that non-null plugin objects are not returned when the plugin is incorrectly configured.

Avoid Device Memory Allocations in `clone()`

Since `clone` is called multiple times in the builder, device memory allocations could be significantly expensive. One option is to do persistent memory allocations in

the constructor, copy to device when the plugin is ready-to-use (for example, in `configurePlugin`), and release during destruction.

Serializing Arbitrary Pieces of Data and Custom Types

Plugin authors can utilize `PluginField` of `PluginFieldType::kUNKNOWN` to indicate arbitrary pieces of data to be serialized. In this case, the `length` of the respective `PluginField` should be the number of bytes corresponding to the buffer pointed to by `data`. The serialization of non-primitive types can be achieved in this way.

9.9. Plugin Shared Libraries

TensorRT contains built-in plugins that can be loaded statically into your application.

You can explicitly register custom plugins with TensorRT using the `REGISTER_TENSORRT_PLUGIN` and `registerCreator` interfaces (refer to Adding Custom Layers Using the C++ API). However, you may want TensorRT to manage registration of a plugin library, and, in particular, serialize plugin libraries with the plan file so they are automatically loaded when the engine is created. This can be especially useful when you want to include the plugins in a version compatible engine, so that you do not need to manage them after building the engine. In order to take advantage of this, you can build shared libraries with specific entry points recognized by TensorRT.

9.9.1. Generating Plugin Shared Libraries

To create a shared library for plugins, the library must have the following public symbols defined:

```
extern "C" void setLoggerFinder(ILoggerFinder& finder);
extern "C" IPluginCreator* const* getPluginCreators(int32_t& nbCreators) const;
```

Note extern `"C"` above is only used to prevent name mangling, and the methods themselves should be implemented in C++. Consult your compiler's ABI documentation for more details.

`setLoggerFinder()` should set a global pointer of `ILoggerFinder` in the library for logging in the plugin code. `getPluginCreators()` returns a list of plugin creators your library contains. An example of implementation of these entry points can be found in `plugin/common/vfcCommon.h/cpp`.

To serialize your plugin libraries with your engine plan, provide the plugin libraries paths to TensorRT using `setPluginsToSerialize()` in `BuilderConfig`.

When building version compatible engines, you may also want to package plugins in the plan. The packaged plugins will have the same lifetime as the engine and will be automatically registered/deregistered when running the engine.

9.9.2. Using Plugin Shared Libraries

After building your shared libraries, you can configure the builder to serialize the libraries with the engine. Next time, when you load the engine into TensorRT, the serialized plugin libraries will be loaded and registered automatically.

Note: `IPluginRegistry`'s `loadLibrary()` (C++, Python) functionality, demonstrated below, is not supported for plugin shared libraries containing V3 plugin creators (`IPluginCreatorV3One`). As a workaround,define the entry point `IPluginCreatorInterface* const* getCreators()` in your library and then query this to enumerate over each plugin creator and register it manually using `IPluginRegistry`'s `registerCreator()` (C++, Python).

Load the plugins for use with the builder prior to building the engine:

C++

```
for (size_t i = 0; i < nbPluginLibs; ++i)
{
    builder->getPluginRegistry().loadLibrary(pluginLibs[i]);
}
```

Python

```
for plugin_lib in plugin_libs:
    builder.get_plugin_registry().load_library(plugin_lib)
```

Next, decide if the plugins should be included with the engine or shipped externally. You can serialize the plugins with the plan as follows:

C++

```
IBuilderConfig *config = builder->createBuilderConfig();
...
config->setPluginsToSerialize(pluginLibs, nbPluginLibs);
```

Python

```
config = builder.create_builder_config()
...
config.plugins_to_serialize = plugin_libs
```

Alternatively, you can keep the plugins external to the engine. You will need to ship these libraries along with the engine when it is deployed, and load them explicitly in the runtime prior to deserializing the engine:

C++

```
// In this example, getExternalPluginLibs() is a user-implemented method which retrieves
 the list of libraries to use with the engine
std::vector<std::string> pluginLibs = getExternalPluginLibs();
for (auto const &pluginLib : pluginLibs)
{
     runtime->getPluginRegistry().loadLibrary(pluginLib.c_str())
}
```

Python

```
# In this example, get_external_plugin_libs() is a user-implemented method which retrieves
 the list of libraries to use with the engine
plugin_libs = get_external_plugin_libs()
for plugin_lib in plugin_libs:
    runtime.get_plugin_registry().load_library(plugin_lib)
```

Chapter 10. Working with Loops

NVIDIA TensorRT supports loop-like constructs, which can be useful for recurrent networks. TensorRT loops support scanning over input tensors, recurrent definitions of tensors, and both "scan outputs" and "last value" outputs.

10.1. Defining a Loop

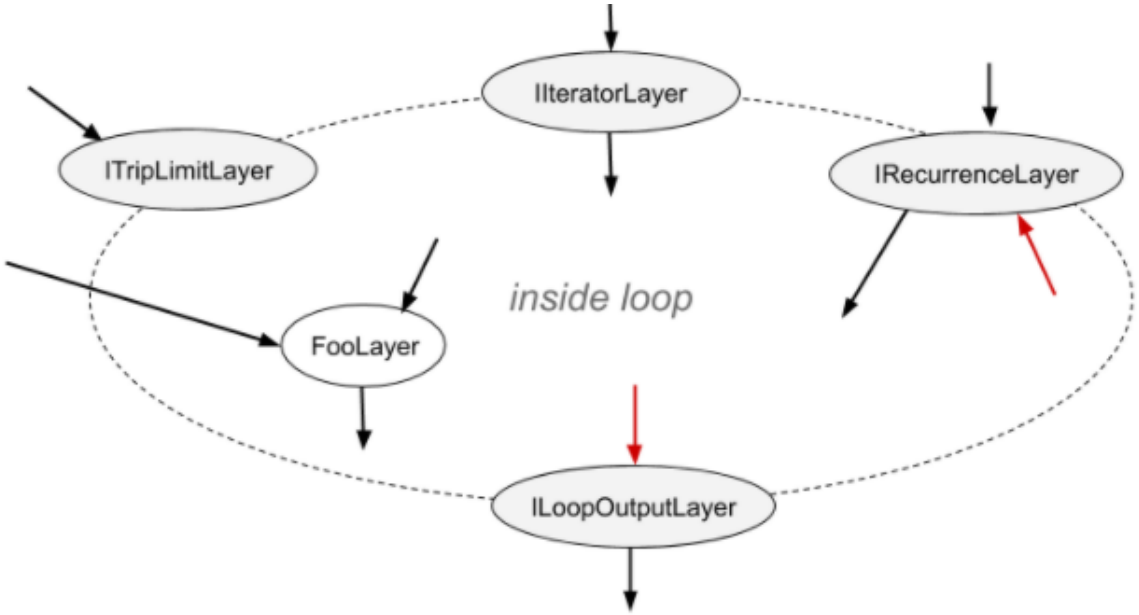
A loop is defined by *loop boundary layers*.

- `ITripLimitLayer` specifies how many times that the loop iterates.
- `IIteratorLayer` enables a loop to iterate over a tensor.
- `IRecurrenceLayer` specifies a recurrent definition.
- `ILoopOutputLayer` specifies an output from the loop.

Each of the boundary layers inherits from class `ILoopBoundaryLayer`, which has a method `getLoop()` for getting its associated `ILoop`. The `ILoop` object identifies the loop. All loop boundary layers with the same `ILoop` belong to that loop.

Figure 14 depicts the structure of a loop and data flow at the boundary. Loop-invariant tensors can be used inside the loop directly, such as shown for FooLayer.

Figure 14. A TensorRT loop is set by loop boundary layers. Dataflow can leave the loop only by `ILoopOutputLayer`. The only back edges allowed are the second input to `IRecurrenceLayer`.



A loop can have multiple `IIteratorLayer`, `IRecurrenceLayer`, and `ILoopOutputLayer`, and at most two `ITripLimitLayers` as explained later. A loop with no `ILoopOutputLayer` has no output and is optimized by TensorRT.

Layers For Flow-Control Constructs describes the TensorRT layers that may be used in the loop interior.

Interior layers are free to use tensors defined inside or outside the loop. The interior can contain other loops (refer to Nested Loops) and other conditional constructs (refer to Conditionals Nesting).

To define a loop, first, create an `ILoop` object with the method `INetworkDefinition::addLoop`. Then add the boundary and interior layers. The rest of this section describes the features of the boundary layers, using *`loop`* to denote the `ILoop*` returned by `INetworkDefinition::addLoop`.

`ITripLimitLayer` supports both counted loops and while-loops.

- *`loop`*`->addTripLimit(`*`t`*`,TripLimit::kCOUNT)` creates an `ITripLimitLayer` whose input *`t`* is a 0D INT32 tensor that specifies the number of loop iterations.
- *`loop`*`->addTripLimit(`*`t`*`,TripLimit::kWHILE)` creates an `ITripLimitLayer` whose input *`t`* is a 0D Bool tensor that specifies whether an iteration should occur. Typically, *`t`* is either the output of an `IRecurrenceLayer` or a calculation based on said output.

A loop can have at most one of each kind of limit.

`IIteratorLayer` supports iterating forwards or backward over any axis.

- *`loop`*`->addIterator(`*`t`*`)` adds an `IIteratorLayer` that iterates over axis 0 of tensor *`t`*. For example, if the input is the matrix:

```
2 3 5
4 6 8
```

the output is the 1D tensor `{2, 3, 5}` on the first iteration and `{4, 6, 8}` for the second iteration. It is invalid to iterate beyond the tensor's bounds.

- *loop*->addIterator(*t*,*axis*) is similar, but the layer iterates over the given axis. For example, if `axis=1` and the input is a matrix, each iteration delivers a column of the matrix.
- *loop*->addIterator(*t*,*axis*,*reverse*) is similar, but the layer produces its output in reverse order if *reverse*=true.

`ILoopOutputLayer` supports three forms of loop output:

- *loop*->addLoopOutput(*t*,LoopOutput::kLAST_VALUE) outputs the last value of *t*, where `t` must be the output of a `IRecurrenceLayer`.
- *loop*->addLoopOutput(*t*,LoopOutput::kCONCATENATE,*axis*) outputs the concatenation of each iteration's input to *t*. For example, if the input is a 1D tensor, with value `{a,b,c}` on the first iteration and `{d,e,f}` on the second iteration, and *axis*=0, the output is the matrix:

  ```
  a b c
  d e f
  ```

  If *axis*=1, the output is:

  ```
  a d
  b e
  c f
  ```

- *loop*->addLoopOutput(*t*,LoopOutput::kREVERSE,*axis*) is similar, but reverses the order.

Both the `kCONCATENATE` and `kREVERSE` forms of `ILoopOutputLayer` require a second input, which is a 0D INT32 shape tensor specifying the length of the new output dimension. When the length is greater than the number of iterations, the extra elements contain arbitrary values. The second input, for example *u*, should be set using `ILoopOutputLayer::setInput(1,`*u*`)`.

Finally, there is `IRecurrenceLayer`. Its first input specifies the initial output value, and its second input specifies the next output value. The first input must come from outside the loop; the second input usually comes from inside the loop. For example, the TensorRT analog of this C++ fragment:

```
for (int32_t i = j; ...; i += k) ...
```

could be created by these calls, where `j` and `k` are `ITensor*`.

```
ILoop* loop = n.addLoop();
IRecurrenceLayer* iRec = loop->addRecurrence(j);
ITensor* i = iRec->getOutput(0);
ITensor* iNext = addElementWise(*i, *k,
    ElementWiseOperation::kADD)->getOutput(0);
iRec->setInput(1, *iNext);
```

The second input to `IRecurrenceLayer` is the only case where TensorRT allows a back edge. If such inputs are removed, the remaining network must be acyclic.

10.2. Formal Semantics

TensorRT has applicative semantics, meaning there are no visible side effects other than engine inputs and outputs. Because there are no side effects, intuitions about loops from imperative languages do not always work. This section defines formal semantics for TensorRT's loop constructs.

The formal semantics is based on *lazy sequences* of tensors. Each iteration of a loop corresponds to an element in the sequence. The sequence for a tensor X inside the loop is denoted `#`X_0`, `X_1`, `X_2`, ...#`. Elements of the sequence are evaluated lazily, meaning, as needed.

The output from `IIteratorLayer(X) is #X[0], X[1], X[2], ...#` where `X[i]` denotes subscripting on the axis specified for the `IIteratorLayer`.

The output from `IRecurrenceLayer(X,Y)is #X, `Y_0`, `Y_1`, `Y_2`, ...#`.

The input and output from an `ILoopOutputLayer` depend on the kind of `LoopOutput`.

- `kLAST_VALUE`: Input is a single tensor `X`, and output is X_n for an n-trip loop.
- `kCONCATENATE`: The first input is a tensor `X`, and the second input is a scalar shape tensor `Y`. The result is the concatenation of X_0, X_1, X_2, ... X_{n-1} with post padding, if necessary, to the length specified by `Y`. It is a runtime error if `Y < n`. `Y` is a build time constant. Note the inverse relationship with `IIteratorLayer`. `IIteratorLayer` maps a tensor to a sequence of subtensors; `ILoopOutputLayer` with `kCONCATENATE` maps a sequence of sub tensors to a tensor.
- `kREVERSE`: Similar to `kCONCATENATE`, but the output is in the reverse direction.

The value of `n` in the definitions for the output of `ILoopOutputLayer` is determined by the `ITripLimitLayer` for the loop:

- For counted loops, it is the iteration count, meaning the input to the `ITripLimitLayer`.
- For while loops, it is the least n such that X_n is false, where `X` is the sequence for the `ITripLimitLayer`'s input tensor.

The output from a non-loop layer is a sequence-wise application of the layer's function. For example, for a two-input non-loop layer `F(X,Y) = #f(`X_0`,`Y_0`), f(`X_1`,`Y_1`), f(`X_2`,`Y_2`)...#`. If a tensor comes from outside the loop, that is, a loop invariant, then the sequence for it is created by replicating the tensor.

10.3. Nested Loops

TensorRT infers the nesting of the loops from the data flow. For instance, if loop B uses values defined *inside* loop A, then B is considered to be nested inside of A.

TensorRT rejects networks where the loops are not cleanly nested, such as if loop A uses values defined in the interior of loop B and vice versa.

10.4. Limitations

A loop that refers to more than one dynamic dimension can take an unexpected amount of memory.

In a loop, memory is allocated as if all dynamic dimensions take on the maximum value of any of those dimensions. For example, if a loop refers to two tensors with dimensions `[4,x,y]` and `[6,y]`, memory allocation for those tensors is as if their dimensions were `[4,max(x,y),max(x,y)]` and `[6,max(x,y)]`.

The input to a `LoopOutputLayer` with `kLAST_VALUE` must be the output from an `IRecurrenceLayer`.

The loop API supports only FP32 and FP16 precision.

10.5. Replacing `IRNNv2Layer` with Loops

`IRNNv2Layer` was deprecated in TensorRT 7.2.1 and will be removed in TensorRT 9.0. Use the loop API to synthesize a recurrent sub network. For an example, refer to `sampleCharRNN`, method `SampleCharRNNLoop::addLSTMCell`. You can express general recurrent networks instead of being limited to the prefabricated cells in `IRNNLayer` and `IRNNv2Layer` using the loop API.

Refer to sampleCharRNN for more information.

Chapter 11. Working with Conditionals

NVIDIA TensorRT supports conditional if-then-else flow control. TensorRT conditionals are used to implement conditional execution of network subgraphs.

11.1. Defining a Conditional

An if-conditional is defined by conditional boundary layers:

- `IConditionLayer` represents the predicate and specifies whether the conditional should execute the true-branch (then-branch) or the false-branch (else-branch).
- `IIfConditionalInputLayer` specifies an input to one of the two conditional branches.
- `IIfConditionalOutputLayer` specifies an output from a conditional.

Each of the boundary layers inherits from class `IIfConditionalBoundaryLayer`, which has a method `getConditional()` for getting its associated `IIfConditional`. The `IIfConditional` instance identifies the conditional. All conditional boundary layers with the same `IIfConditional` belong to that conditional.

A conditional must have exactly one instance of `IConditionLayer`, zero, or more instances of `IIfConditionalInputLayer`, and at least one instance of `IIfConditionalOutputLayer`.

`IIfConditional` implements an if-then-else flow-control construct that provides conditional-execution of a network subgraph based on a dynamic boolean input. It is defined by a boolean scalar predicate `condition`, and two branch subgraphs: a `trueSubgraph` which is executed when `condition` evaluates to `true`, and a `falseSubgraph` which is executed when `condition` evaluates to `false`:

```
If condition is true then:
 output = trueSubgraph(trueInputs);
Else
 output = falseSubgraph(falseInputs);
Emit output
```

Both the true-branch and the false-branch must be defined, similar to the ternary operator in many programming languages.

To define an if-conditional, create an `IIfConditional` instance with the method `INetworkDefinition::addIfConditional`, then add the boundary and branch layers.

```
IIfConditional* simpleIf = network->addIfConditional();
```

The `IIfConditional::setCondition` method takes a single argument: the condition tensor. This 0D boolean tensor (scalar) can be computed dynamically by earlier layers in the network. It is used to decide which of the branches to execute. An `IConditionLayer` has a single input (the condition) and no outputs since it is used internally by the conditional implementation.

```
// Create a condition predicate that is also a network input.
auto cond = network->addInput("cond", DataType::kBOOL, Dims{0});
IConditionLayer* condition = simpleIf->setCondition(*cond);
```

TensorRT does not support a subgraph abstraction for implementing conditional branches and instead uses `IIfConditionalInputLayer` and `IIfConditionalOutputLayer` to define the boundaries of conditionals.

- An `IIfConditionalInputLayer`abstracts a single input to one or both of the branch subgraphs of an `IIfConditional`. The output of a specific `IIfConditionalInputLayer`can feed both branches.

  ```
  // Create an if-conditional input.
  // x is some arbitrary Network tensor.
  IIfConditionalInputLayer* inputX = simpleIf->addInput(*x);
  ```

  Inputs to the then-branch and the else-branch do not have to be of the same type and shape. Each branch can independently include zero or more inputs.

  `IIfConditionalInputLayer`is optional and is used to control which layers will be part of the branches (refer to Conditional Execution). If all of a branch's outputs do not depend on an `IIfConditionalInputLayer`instance, that branch is empty. An empty else-branch can be useful when there are no layers to evaluate when the condition is `false`, and the network evaluation should proceed following the conditional (refer to Conditional Examples).

- An `IIfConditionalOutputLayer`abstracts a single output of the if-conditional. It has two inputs: an output from the true-subgraph (input index 0) and an output from the false-subgraph (input index 1). The output of an `IIfConditionalOutputLayer` can be thought of as a placeholder for the final output that will be determined during runtime.

  `IIfConditionalOutputLayer` serves a role similar to that of a Φ (Phi) function node in traditional SSA control-flow graphs. Its semantics are: choose either the output of the true-subgraph or the false-subgraph.

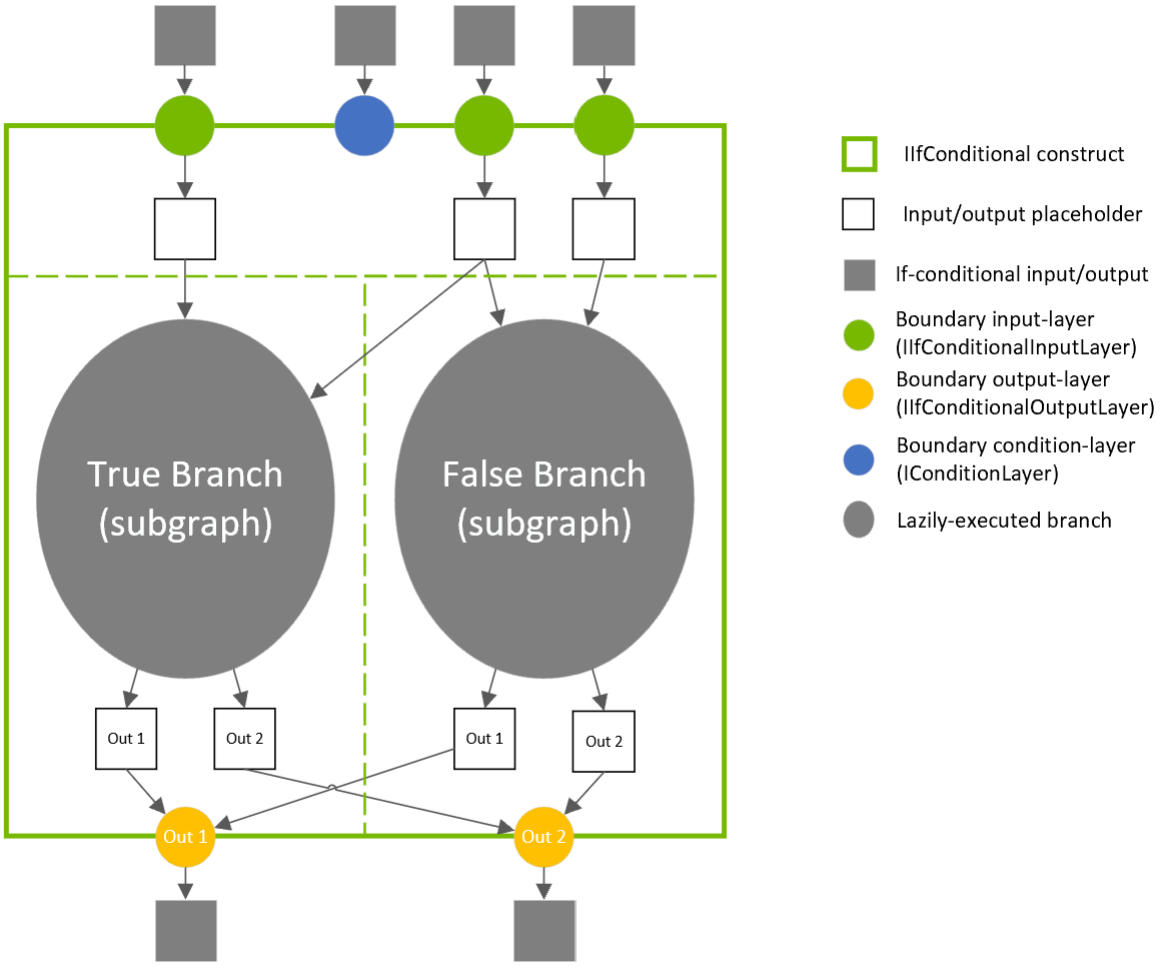
  ```
  // trueSubgraph and falseSubgraph represent network subgraphs
  IIfConditionalOutputLayer* outputLayer = simpleIf->addOutput(
      *trueSubgraph->getOutput(0),
      *falseSubgraph->getOutput(0));
  ```

  All outputs of an `IIfConditional` must be sourced at an `IIfConditionalOutputLayer` instance.

  An if-conditional without outputs has no effect on the rest of the network, therefore, it is considered ill-formed. Each of the two branches (subgraphs) must also have at least one output. The output of an if-conditional can be marked as the output of the network, unless that if-conditional is nested inside another if-conditional or loop.

The diagram below provides a graphical representation of the abstract model of an if-conditional. The green rectangle represents the interior of the conditional, which is limited to the layer types listed in Layers For Flow-Control Constructs.

Figure 15. An If-Conditional Construct Abstract Model



11.2. Conditional Execution

Conditional execution of network layers is a network evaluation strategy in which branch-layers (the layers belonging to a conditional subgraph) are executed only if the values of the branch outputs are needed. In conditional-execution, either the true-branch or the false-branch is executed and allowed to change the network state.

In contrast, in predicated-execution, both the true-branch and the false-branch are executed and only one of these is allowed to change the network evaluation state, depending on the value of the condition predicate (that is, only the outputs of one of the subgraphs is fed into the following layers).

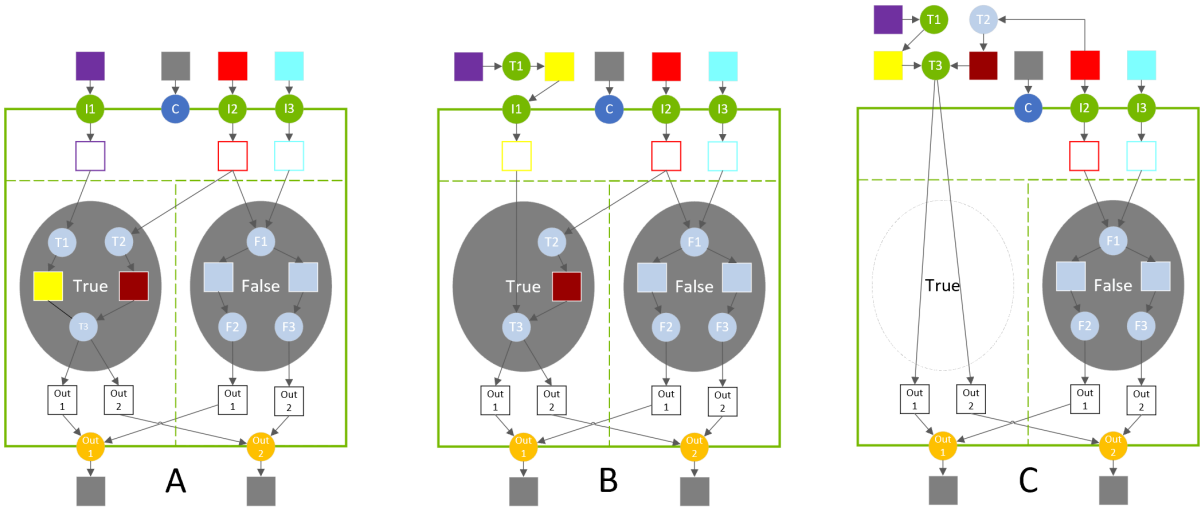
Conditional execution is sometimes called *lazy evaluation*, and predicated-execution is sometimes referred to as *eager evaluation*.

Instances of `IIfConditionalInputLayer` can be used to specify which layers are invoked eagerly and which are invoked lazily. This is done by tracing the network

layers backwards, starting with each of the conditional outputs. Layers that are data-dependent on the output of at least one `IIfConditionalInputLayer` are considered internal to the conditional and are therefore evaluated lazily. In the extreme case that no instances of `IIfConditionalInputLayer` are added to the conditional, all of the layers are executed eagerly, similarly to `ISelectLayer`.

The three diagrams below depict how the choice of `IIfConditionalInputLayer` placement controls execution scheduling.

Figure 16. Controlling Conditional-Execution using `IIfConditionalInputLayer` Placement



In diagram A, the true-branch is composed of three layers (T1, T2, T3). These layers execute lazily when the condition evaluates to `true`.

In diagram B, input-layer I1 is placed after layer T1, which moves T1 out of the true-branch. Layer T1 executes eagerly before evaluating the if-construct.

In diagram C, input-layer I1 is removed altogether, which moves T3 outside the conditional. T2's input is reconfigured to create a legal network, and T2 also moves out of the true-branch. When the condition evaluates to `true`, the conditional does not compute anything since the outputs have already been eagerly computed (but it does copy the conditional relevant inputs to its outputs).

11.3. Nesting and Loops

Conditional branches may nest other conditionals and may also nest loops. Loops may nest conditionals. As in loop nesting, TensorRT infers the nesting of the conditionals and loops from the data flow. For example, if conditional B uses a value defined inside loop A, then B is considered to be nested inside of A.

There can be no cross-edges connecting layers in the true-branch to layers in the false-branch, and vice versa. In other words, the outputs of one branch cannot depend on layers in the other branch.

For example, refer to Conditional Examples for how nesting can be specified.

11.4. Limitations

The number of output tensors in both true/false subgraph branches must be the same. The type and shape of each output tensor from the branches must be the same.

Note that this is more constrained than the ONNX specification, which requires that the true/false subgraphs have the same number of outputs and use the same output types, but allows for different output shapes.

11.5. Conditional Examples

11.5.1. Simple If-Conditional

The following example shows how to implement a simple conditional that conditionally performs an arithmetic operation on two tensors.

Conditional

```
condition = true
If condition is true:
        output = x + y
Else:
        output = x - y
```

Example

```
ITensor* addCondition(INetworkDefinition& n, bool predicate)
{
    // The condition value is a constant int32 input that is cast to boolean because TensorRT
 doesn't support boolean constant layers.

    static const Dims scalarDims = Dims{0, {}};
    static float constexpr zero{0};
    static float constexpr one{1};

    float* const val = predicate ? &one : &zero;

    ITensor* cond =
        n.addConstant(scalarDims, DataType::kINT32, val, 1})->getOutput(0);

    auto* cast = n.addIdentity(cond);
    cast->setOutputType(0, DataType::kBOOL);
    cast->getOutput(0)->setType(DataType::kBOOL);

    return cast->getOutput(0);
}

IBuilder* builder = createInferBuilder(gLogger);
```

```
INetworkDefinition& n = *builder->createNetworkV2(0U);
auto x = n.addInput("x", DataType::kFLOAT, Dims{1, {5}});
auto y = n.addInput("y", DataType::kFLOAT, Dims{1, {5}});
ITensor* cond = addCondition(n, true);

auto* simpleIf = n.addIfConditional();
simpleIf->setCondition(*cond);

// Add input layers to demarcate entry into true/false branches.
x = simpleIf->addInput(*x)->getOutput(0);
y = simpleIf->addInput(*y)->getOutput(0);

auto* trueSubgraph = n.addElementWise(*x, *y, ElementWiseOperation::kSUM)->getOutput(0);
auto* falseSubgraph = n.addElementWise(*x, *y, ElementWiseOperation::kSUB)->getOutput(0);

auto* output = simpleIf->addOutput(*trueSubgraph, *falseSubgraph)->getOutput(0);
n.markOutput(*output);
```

11.5.2. Exporting from PyTorch

The following example shows how to export scripted PyTorch code to ONNX. The code in function `sum_even` performs an if-conditional nested in a loop.

```
import torch.onnx
import torch
import tensorrt as trt
import numpy as np

TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
EXPLICIT_BATCH = 1 << (int)(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)

@torch.jit.script
def sum_even(items):
    s = torch.zeros(1, dtype=torch.float)
    for c in items:
        if c % 2 == 0:
            s += c
    return s

class ExampleModel(torch.nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, items):
        return sum_even(items)

def build_engine(model_file):
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network(EXPLICIT_BATCH)
    config = builder.create_builder_config()
    parser = trt.OnnxParser(network, TRT_LOGGER)

    with open(model_file, 'rb') as model:
        assert parser.parse(model.read())
        return builder.build_engine(network, config)

def export_to_onnx():
    items = torch.zeros(4, dtype=torch.float)
    example = ExampleModel()
    torch.onnx.export(example, (items), "example.onnx", verbose=False, opset_version=13,
 enable_onnx_checker=False, do_constant_folding=True)

export_to_onnx()
build_engine("example.onnx")
```

Chapter 12. Working with DLA

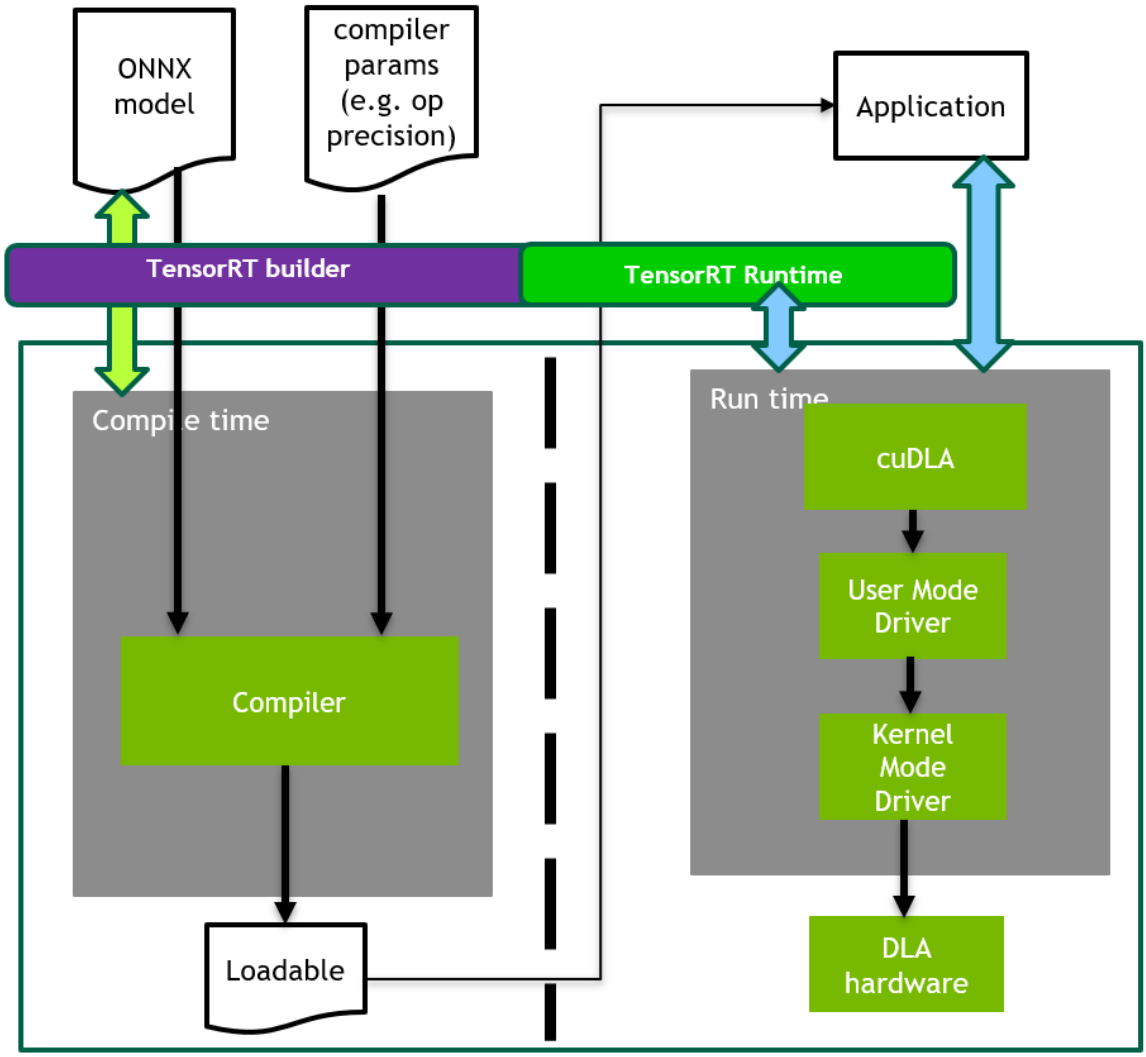
NVIDIA DLA (Deep Learning Accelerator) is a fixed-function accelerator engine targeted for deep learning operations. DLA is designed to do full hardware acceleration of convolutional neural networks. DLA supports various layers such as convolution, deconvolution, fully connected, activation, pooling, batch normalization, and so on. DLA does not support Explicit Quantization. For more information about DLA support in TensorRT layers, refer to DLA Supported Layers and Restrictions.

DLA is useful for offloading CNN processing from the iGPU, and is significantly more power-efficient for these workloads. In addition, it can provide an independent execution pipeline in cases where redundancy is important, for example in mission-critical or safety applications.

For more information about DLA, refer to the DLA developer page and the DLA tutorial Getting started with the Deep Learning Accelerator on NVIDIA Jetson Orin.

When building a model for DLA, the TensorRT builder parses the network and calls the DLA compiler to compile the network into a DLA loadable. Refer to Using trtexec to see how to build and run networks on DLA.

Figure 17. Workflow for the Building and Runtime Phases of DLA



12.1. Building and Launching the Loadable

There are several different ways to build and launch a DLA loadable, either embedded in a TensorRT engine or in standalone form.

For generating a standalone DLA loadable to be used outside TensorRT, refer to DLA Standalone Mode.

12.1.1. Using `trtexec`

To allow `trtexec` to use the DLA, you can use the `–useDLACore` flag. For example, to run the ResNet-50 network on DLA core 0 in FP16 mode, with GPU Fallback Mode for unsupported layers, issue:

```
./trtexec --onnx=data/resnet50/ResNet50.onnx --useDLACore=0 --fp16 --allowGPUFallback
```

The `trtexec` tool has additional arguments to run networks on DLA. For more information, refer to Command-Line Programs.

12.1.2. Using the TensorRT API

You can use the TensorRT API to build and run inference with DLA and to enable DLA at layer level. The relevant APIs and samples are provided in the following sections.

12.1.2.1. Running on DLA during TensorRT Inference

The TensorRT builder can be configured to enable inference on DLA. DLA support is currently limited to networks running in FP16 and INT8 mode. The `DeviceType` enumeration is used to specify the device that the network or layer executes on. The following API functions in the `IBuilderConfig` class can be used to configure the network to use DLA:

`setDeviceType(ILayer* layer, DeviceType deviceType)`
This function can be used to set the `deviceType` that the layer must execute on.

`getDeviceType(const ILayer* layer)`
This function can be used to return the `deviceType` that this layer executes on. If the layer is executing on the GPU, this returns `DeviceType::kGPU`.

`canRunOnDLA(const ILayer* layer)`
This function can be used to check if a layer can run on DLA.

`setDefaultDeviceType(DeviceType deviceType)`
This function sets the default `deviceType` to be used by the builder. It ensures that all the layers that can run on DLA runs on DLA unless `setDeviceType` is used to override the `deviceType` for a layer.

`getDefaultDeviceType()`
This function returns the default `deviceType` which was set by `setDefaultDeviceType`.

`isDeviceTypeSet(const ILayer* layer)`
This function checks whether the `deviceType` has been explicitly set for this layer.

`resetDeviceType(ILayer* layer)`
This function resets the `deviceType` for this layer. The value is reset to the `deviceType` that is specified by `setDefaultDeviceType` or `DeviceType::kGPU` if none is specified.

`allowGPUFallback(bool setFallBackMode)`
This function notifies the builder to use GPU if a layer that was supposed to run on DLA cannot run on DLA. For more information, refer to GPU Fallback Mode.

reset()
This function can be used to reset the `IBuilderConfig` state, which sets the `deviceType` for all layers to be `DeviceType::kGPU`. After reset, the builder can be reused to build another network with a different DLA config.

The following API functions in `IBuilder` class can be used to help configure the network for using the DLA:

getMaxDLABatchSize()
This function returns the maximum batch size DLA can support.

> Note: For any tensor, the total volume of index dimensions combined with the requested batch size must not exceed the value returned by this function.

getNbDLACores()
This function returns the number of DLA cores available to the user.

If the builder is not accessible, such as in the case where a plan file is being loaded online in an inference application, then the DLA to be used can be specified differently by using DLA extensions to the `IRuntime`. The following API functions in the `IRuntime` class can be used to configure the network to use DLA:

getNbDLACores()
This function returns the number of DLA cores that are accessible to the user.

setDLACore(int dlaCore)
The DLA core to execute on. Where `dlaCore` is a value between `0` and `getNbDLACores() - 1`. The default value is `0`.

getDLACore()
The DLA core that the runtime execution is assigned to. The default value is 0.

12.1.2.2. Example: Run Samples with DLA

This section provides details on how to run a TensorRT sample with DLA enabled.

Create the builder:

```
auto builder = SampleUniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(gLogger));
if (!builder) return false;
builder->setMaxBatchSize(batchSize);
config->setMaxWorkspaceSize(16_MB);
```

Then, enable `GPUFallback` mode:

```
config->setFlag(BuilderFlag::kGPU_FALLBACK);
config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
```

Enable execution on DLA, where `dlaCore` specifies the DLA core to execute on:

```
config->setDefaultDeviceType(DeviceType::kDLA);
config->setDLACore(dlaCore);
```

With these additional changes, sampleMNIST is ready to execute on DLA. To run samples with DLA Core 1, append `--useDLACore=0` to the sample command.

12.1.2.3. Example: Enable DLA Mode for a Layer during Network Creation

In this example, let us create a simple network with Input, Convolution, and Output.

1. Create the builder, builder configuration, and the network:

```
IBuilder* builder = createInferBuilder(gLogger);
IBuilderConfig* config = builder.createBuilderConfig();
INetworkDefinition* network = builder->createNetworkV2(0U);
```

2. Add the Input layer to the network, with the input dimensions.

```
auto data = network->addInput(INPUT_BLOB_NAME, dt, Dims3{1, INPUT_H, INPUT_W});
```

3. Add the Convolution layer with hidden layer input nodes, strides, and weights for filter and bias.

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

4. Set the convolution layer to run on DLA:

```
if(canRunOnDLA(conv1))
{
config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
builder->setDeviceType(conv1, DeviceType::kDLA);

}
```

5. Mark the output:

```
network->markOutput(*conv1->getOutput(0));
```

6. Set the DLA core to execute on:

```
config->setDLACore(0)
```

12.1.3. Using the cuDLA API

cuDLA is an extension of the CUDA programming model that integrates DLA runtime software with CUDA. This integration makes it possible to launch DLA loadables using CUDA programming constructs such as streams and graphs.

Managing shared buffers as well as synchronizing the tasks between GPU and DLA is transparently handled by cuDLA. Refer to the NVIDIA cuDLA documentation on how the cuDLA APIs can be used for these use cases while writing a cuDLA application.

Refer to the DLA Standalone Mode section for more information on how to use TensorRT to build a standalone DLA loadable usable with cuDLA.

12.2. DLA Supported Layers and Restrictions

This section lists the layers supported by DLA along with the constraints associated with each layer.

12.2.1. General Restrictions

The following restrictions apply to all layers while running on DLA:

- The maximum supported batch size is 4096.
- The maximum supported size for non-batch dimensions is 8192.

- DLA does not support dynamic dimensions. Thus, for wildcard dimensions, the `min`, `max`, and `opt` values of the profile must be equal.
- The runtime dimensions must be the same as the dimension used for building.
- TensorRT may split a network into multiple DLA loadables if any intermediate layers cannot run on DLA and `GPUFallback` is enabled. Otherwise, TensorRT can emit an error and fallback. For more information, refer to GPU Fallback Mode.
- At most 16 DLA loadables can be loaded concurrently, per core, due to hardware and software memory limitations.
- Within a single DLA loadable, each layer must have the same batch size. If layers have different batch sizes, they will be partitioned into separate DLA graphs.

Note: Batch size for DLA is the product of all index dimensions except the `CHW` dimensions. For example, if input dimensions are `NPQRS`, the effective batch size is `N*P`.

12.2.2. Layer Support and Restrictions

The following list provides layer support and restrictions to the specified layers while running on DLA:

Convolution and Fully Connected layers

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Each dimension of the kernel size must be in the range `[1, 32]`.
- Padding must be in the range `[0, 31]`.
- Dimensions of padding must be less than the corresponding kernel dimension.
- Dimensions of stride must be in the range `[1, 8]`.
- Number of output maps must be in the range `[1, 8192]`.
- Number of input channels `[1, 8192]`.
- Number of groups must be in the range `[1, 8192]` for operations using the formats `TensorFormat::kDLA_LINEAR`, `TensorFormat::kCHW16`, and `TensorFormat::kCHW32`.
- Number of groups must be in the range `[1, 4]` for operations using the formats `TensorFormat::kDLA_HWC4`.
- Dilated convolution must be in the range `[1, 32]`.
- Operations are not supported if the CBUF size requirement `wtBanksForOneKernel + minDataBanks` exceeds the `numConvBufBankAllotted` limitation `16`, where CBUF is the internal convolution cache that stores input weights and activation before operating on them, `wtBanksForOneKernel` is the minimum banks for one kernel to store the minimum weight/kernel elements needed for convolution, and `minDataBanks` is the minimum banks to store the minimum activation data needed for convolution. When a convolution layer fails validation due to CBUF constraints, details are displayed in the logging output.

Deconvolution layer

- Only two spatial dimensions are supported.
- Both FP16 and INT8 are supported.
- The kernel dimensions and strides must be in the range `[1, 32]`, or must be `1x[64, 96, 128]` and `[64, 96, 128]x1`.
- TensorRT has disabled deconvolution square kernels and strides in the range `[23 - 32]` on DLA as they significantly slow down compilation.
- Padding must be `0`
- Grouped deconvolution must be `1`.
- Dilated deconvolutions must be `1`.
- Number of input channels must be in the range `[1, 8192]`.
- Number of output channels must be in the range `[1, 8192]`.

Pooling layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Operations supported: `kMAX`, `kAVERAGE`.
- Dimensions of the window must be in the range `[1, 8]`.
- Dimensions of padding must be in the range `[0, 7]`.
- Dimensions of stride must be in the range `[1, 16]`.
- With INT8 mode, input and output tensor scales must be the same.

Activation layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Functions supported: `ReLU`, `Sigmoid`, `TanH`, `Clipped ReLU`, and `Leaky ReLU`.
  - Negative slope is not supported for `ReLU`.
  - `Clipped ReLU` only supports values in the range `[1, 127]`.
  - `TanH`, `Sigmoid` INT8 support is supported by auto-upgrading to FP16.

Parametric ReLU layer

- Slope input must be a build time constant and have the same rank as the input tensor.

ElementWise layer

- Only two spatial dimension operations are supported.

- Both FP16 and INT8 are supported.
- Operations supported: `Sum`, `Sub`, `Product`, `Max`, `Min`, `Div`, `Pow`, `Equal`, `Greater`, and `Less` (described separately).
- Broadcasting is supported when one of the operands has one of the following shape configurations:
  - NCHW (that is, shapes equal)
  - NC11 (that is, N and C equal, H and W are 1)
  - N111 (that is, N equal, C, H, and W are 1)
- `Div` operation
  - The first input (dividend) can be either INT8 or FP16 or an FP32 constant. The second input (divisor) must be INT8 or an FP32 constant.
  - If one of the inputs is constant, all values of its weights must be the same. Additionally, the other input must be an INT8 non-constant.
- `Pow` operation
  - One input must be a FP32 constant filled with the same value, the other input must be an INT8 non-constant.

Comparison operations (`Equal`, `Greater`, `Less`)

- Only supports INT8 layer precision. Only supports INT8 inputs except when using constants, which should be of FP32 type filled with the same value.
- DLA requires that the comparison operation output be FP16 or INT8 type. Thus, the comparison layer must be immediately followed by a Cast operation (`IIdentityLayer`/`ICastLayer`) to FP16 or INT8 and should have no direct consumers other than this Cast operation.
- For both the `ElementWise` comparison layer and the subsequent `IIdentityLayer`/`ICastLayer` mentioned above, explicitly set your device types to DLA and their precisions to INT8. Otherwise, these layers will run on the GPU.
- Even with GPU fallback allowed, you should expect failures in engine construction in some cases, for example, when DLA loadable compilation fails. If this is the case, unset the device types and/or precisions of both the `ElementWise` comparison layer and `IIdentityLayer`/`ICastLayer` to have both offloaded to GPU.

Scale layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Mode supported: `Uniform`, `Per-Channel`, and `ElementWise`.
- Only `scale` and `shift` operations are supported.

LRN (Local Response Normalization) layer

- Allowed window sizes are `3`, `5`, `7`, or `9`.
- Normalization region supported is `ACROSS_CHANNELS`.
- `LRN` INT8 is supported by auto-upgrading to FP16.

Concatenation layer

- DLA supports concatenation only along the channel axis.
- Concat must have at least two inputs.
- All the inputs must have the same spatial dimensions.
- Both FP16 and INT8 are supported.
- With INT8 mode, the dynamic range of all the inputs must be the same.
- With INT8 mode, the dynamic range of output must be equal to each of the inputs.

Resize layer

- The number of scales must be exactly `4`.
- The first two elements in scales must be exactly `1` (for unchanged batch and channel dimensions).
- The last two elements in scales, representing the scale values along height and width dimensions, respectively, must be integer values in the range of `[1, 32]` in nearest-neighbor mode and `[1, 4]` in bilinear mode.
- Note that for bilinear resize INT8 mode, when the input dynamic range is larger than the output dynamic range, the layer will be upgraded to FP16 to preserve accuracy. This can negatively affect the latency.

Unary layer

- Only the ABS operation is supported.
- DLA supports `ABS`, `SIN`, `COS`, and `ATAN` operation types.
- For `SIN`, `COS`, and `ATAN`, input precision must be INT8.
- All input non-batch dimensions must be in the range `[1, 8192]`.

Slice layer

- Both FP16 and INT8 are supported.
- Supports batch sizes up to general DLA maximum.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Only supports 4-D inputs and slicing at CHW dimensions.

- Only supports static slicing, so slice parameters have to be provided statically either using TensorRT `ISliceLayer` setter APIs or as constant input tensors.

SoftMax layer

- Only supported on NVIDIA Orin™, not Xavier™.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Axis must be one of the non-batch dimensions.
- Supports FP16 and INT8 precision.
- Internally, there are two modes, and the mode is selected based on the given input tensor shape.
  - The accurate mode is triggered when all non-batch, non-axis dimensions are `1`.
  - The optimized mode allows the non-batch, non-axis dimensions to be greater than `1` but restricts the axis dimension to 1024 and involves an approximation that may cause a small error in the output. The magnitude of the error increases as the size of the axis dimension approaches 1024.

Shuffle layer

- Only supports 4-D input tensors.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Note that DLA decomposes the layer into standalone transpose and reshape operations. This means that the above restrictions apply individually to each of the decomposed operations.
- Batch dimensions cannot be involved in either reshapes or transposes.

Reduce layer

- Only supports 4-D input tensors.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Both FP16 and INT8 are supported.
- Only supports MAX operation type where any combination of the CHW axes is reduced.

12.2.3. Inference on NVIDIA Orin

Due to the difference in hardware specifications between NVIDIA Orin and Xavier DLA, an increase up to 2x in latency may be observed for FP16 convolution operations on NVIDIA Orin.

On NVIDIA Orin, DLA stores weights for non-convolution operations (FP16 and INT8) inside a loadable as FP19 values (which use 4 byte containers). The channel dimensions are padded to multiples of either 16 (FP16) or 32 (INT8) for those FP19 values. Especially in the case of large per-element `Scale`, `Add`, or `Sub` operations, this can inflate the size

of the DLA loadable, inflating the engine containing such a loadable. Graph optimization may unintentionally trigger this behavior by changing the type of a layer, for example, when an ElementWise multiplication layer with a constant layer as weights is fused into a scale layer.

12.3. GPU Fallback Mode

The `GPUFallbackMode` sets the builder to use GPU if a layer that was marked to run on DLA could not run on DLA. A layer cannot run on DLA due to the following reasons:

1. The `layer` operation is not supported on DLA.
2. The parameters specified are out of the supported range for DLA.
3. The given batch size exceeds the maximum permissible DLA batch size. For more information, refer to DLA Supported Layers and Restrictions.
4. A combination of layers in the network causes the internal state to exceed what the DLA is capable of supporting.
5. There are no DLA engines available on the platform.

When GPU fallback is disabled, an error is emitted if a layer could not be run on DLA.

12.4. I/O Formats on DLA

DLA supports formats that are unique to the device and have constraints on their layout due to vector width byte requirements.

For DLA input tensors, `kDLA_LINEAR(FP16, INT8)`, `kDLA_HWC4(FP16, INT8)`, `kCHW16(FP16)`, and `kCHW32(INT8)` are supported. For DLA output tensors, only `kDLA_LINEAR(FP16, INT8)`, `kCHW16(FP16)`, and `kCHW32(INT8)` are supported. For `kCHW16` and `kCHW32` formats, if `C` is not an integer multiple, then it must be padded to the next 32-byte boundary.

For `kDLA_LINEAR` format, the stride along the `W` dimension must be padded up to 64 bytes. The memory format is equivalent to a `C` array with dimensions `[N][C][H][roundUp(W, 64/elementSize)]` where `elementSize` is 2 for `FP16` and 1 for `Int8`, with the tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][c][h][w]`.

For `kDLA_HWC4` format, the stride along the `W` dimension must be a multiple of 32 bytes on Xavier and 64 bytes on NVIDIA Orin.

- When `C == 1`, TensorRT maps the format to the native grayscale image format.
- When `C == 3` or `C == 4`, it maps to the native color image format. If `C == 3`, the stride for stepping along the `W` axis must be padded to 4 in elements.

  In this case, the padded channel is located at the 4th-index. Ideally, the padding value does not matter because the 4th channel in the weights is padded to zero by the DLA compiler; however, it is safe for the application to allocate a zero-filled buffer of four channels and populate three valid channels.

- When `C` is `{1, 3, 4}`, then padded `C'` is `{1, 4, 4}` respectively, the memory layout is equivalent to a `C` array with dimensions `[N][H][roundUp(W, 32/C'/elementSize)][C']` where `elementSize` is 2 for `FP16` and 1 for `Int8`. The tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][h][w][c]`, `roundUp` calculates the smallest multiple of `64/elementSize` greater than or equal to `W`.

When using `kDLA_HWC4` as DLA input format, it has the following requirements:

- `C` must be `1`, `3`, or `4`
- The first layer must be convolution.
- The convolution parameters must meet DLA requirements. Refer to DLA Supported Layers and Restrictions for more information.

When GPU fallback is enabled, TensorRT may insert reformatting layers to meet the DLA requirements. Otherwise, the input and output formats must be compatible with DLA. In all cases, the strides that TensorRT expects data to be formatted with can be obtained by querying `IExecutionContext::getStrides`.

12.5. DLA Standalone Mode

If you need to run inference outside of TensorRT, you can use `EngineCapability::kDLA_STANDALONE` to generate a DLA loadable instead of a TensorRT engine. This loadable can then be used with an API like Using the cuDLA API.

12.5.1. Building A DLA Loadable Using C++

1. Set the default device type and engine capability to DLA standalone mode.
```
builderConfig->setDefaultDeviceType(DeviceType::kDLA);
builderConfig->setEngineCapability(EngineCapability::kDLA_STANDALONE);
```
2. Specify FP16, INT8, or both. For example:
```
builderConfig->setFlag(BuilderFlag::kFP16);
```
3. DLA standalone mode disallows reformatting, therefore `BuilderFlag::kDIRECT_IO` needs to be set.
```
builderConfig->setFlag(BuilderFlag::kDIRECT_IO);
```
4. You must set the allowed formats for I/O tensors to one or more of those supported by DLA. See the documentation for the TensorFormat enum for details.
5. Finally, build as normal

12.5.1.1. Using `trtexec` To Generate A DLA Loadable

The `trtexec` tool can generate a DLA loadable instead of a TensorRT engine. Specifying both `--useDLACore` and `--safe` parameters sets the builder capability to `EngineCapability::kDLA_STANDALONE`. Additionally, specifying `--inputIOFormats` and `--outputIOFormats` restricts I/O data type and memory layout. The DLA loadable is saved into a file by specifying `--saveEngine` parameter.

For example, to generate an FP16 DLA loadable for an ONNX model using `trtexec`, issue:

```
./trtexec --onnx=model.onnx --saveEngine=model_loadable.bin --useDLACore=0 --fp16 --
inputIOFormats=fp16:chw16 --outputIOFormats=fp16:chw16 --skipInference --safe
```

12.6. Customizing DLA Memory Pools

You can customize the size of the memory pools allocated to each DLA subnetwork in a network using the `IBuilderConfig::setMemoryPoolLimit` C++ API or the `IBuilderConfig.set_memory_pool_limit` Python API. There are three types of DLA memory pools (refer to the `MemoryPoolType` enum for details):

Managed SRAM

- Behaves like a cache and larger values may improve performance.
- If no managed SRAM is available, DLA can still run by falling back to local DRAM.
- On Orin, each DLA core has 1 MiB of dedicated SRAM. On Xavier, 4 MiB of SRAM is shared across multiple cores including the 2 DLA cores.

Local DRAM

- Used to store intermediate tensors in the DLA subnetwork. Larger values may allow larger subnetworks to be offloaded to DLA.

Global DRAM

- Used to store weights in the DLA subnetwork. Larger values may allow larger subnetworks to be offloaded to DLA.

The amount of memory required for each subnetwork may be less than the pool size, in which case the smaller amount will be allocated. The pool size serves only as an upper bound.

Note that all DLA memory pools require sizes that are powers of 2, with a minimum of 4 KiB. Violating this requirement results in a DLA loadable compilation failure.

In multi-subnetwork situations, it is important to keep in mind that the pool sizes apply per DLA subnetwork, not for the whole network, so it is necessary to be aware of the total amount of resources being consumed. In particular, your network can consume at most twice the managed SRAM as the pool size in aggregate.

For NVIDIA Orin, the default managed SRAM pool size is set to 0.5 MiB whereas Xavier has 1 MiB as the default. This is because Orin has a strict per-core limit, whereas Xavier has some flexibility. This Orin default guarantees in all situations that the aggregate managed SRAM consumption of your engine stays below the hardware limit, but if your engine has only a single DLA subnetwork, this would mean your engine only consumes half the hardware limit so you may see a perf boost by increasing the pool size to 1 MiB.

12.6.1. Determining DLA Memory Pool Usage

Upon successfully compiling loadables from the given network, the builder reports the number of subnetwork candidates that were successfully compiled into loadables, as well as the total amount of memory used per pool by those loadables. For each subnetwork candidate that failed due to insufficient memory, a message will be emitted

to point out which memory pool was insufficient. In the verbose log, the builder also reports the memory pool requirements of each loadable.

12.7. Sparsity on DLA

DLA on the NVIDIA Orin platform supports structured sparsity (SS) that offers the opportunity to minimize latency and maximize throughput in production.

12.7.1. Structured Sparsity

Structured sparsity (SS) accelerates a 2:4 sparsity pattern along the C dimension. In each contiguous block of four values, two values must be zero along C. Generally, SS provides the most benefit for INT8 convolutions that are math-bound, have a channel dimension that is a multiple of 128.

Structured Sparsity has several requirements and limitations.

Requirements

- Only available for INT8 convolution for formats other than NHWC.
- Channel size must be larger than 64.

Limitations

- Only convolutions whose quantized INT8 weights are at most 256K can benefit from SS–in practice, the limitation may be more restrictive.
- Only convolutions with `K % 64` in `{0, 1, 2, 4, 8, 16, 32}`, where `K` is the number of kernels (corresponding to the number of output channels), can benefit from SS in this release.

Chapter 13. Performance Best Practices

13.1. Performance Benchmarking using trtexec

This section introduces how to use `trtexec`; a command-line tool designed for TensorRT performance benchmarking, to get the inference performance measurements of your deep learning models.

If you use the TensorRT NGC container, then `trtexec` is already installed at `/opt/tensorrt/bin/trtexec`. If you manually installed TensorRT, `trtexec` is part of the installation. Alternatively, you can also build `trtexec` from source code using the TensorRT OSS repository.

13.1.1. Performance Benchmarking with an ONNX File

If your model is already in the ONNX format, the `trtexec` tool can measure its performance directly. In this example, we will use the ResNet-50 v1 ONNX model from the ONNX model zoo to showcase how to measure its performance using `trtexec`.

For example, the `trtexec` command to measure the performance of ResNet-50 with batch size 4 is:

```
trtexec --onnx=resnet50-v1-12.onnx --shapes=data:4x3x224x224 --fp16 --noDataTransfers --useCudaGraph --useSpinWait
```

- the `--onnx` flag specifies the path to the ONNX file
- the `--shapes` flag specifies the input tensor shapes
- the `--fp16` flag enables FP16 tactics
- the other flags are added to make performance results more stable.

The value for the `--shapes` flag is in the format of `name1:shape1,name2:shape2,...`. If you do not know the input tensor names and shapes, you can get the information

by visualizing the ONNX model using tools like Netron or by running Polygraphy model inspection on the model.

For example, running `polygraphy inspect model resnet50-v1-12.onnx` prints out:

```
[I] Loading model: /home/pohanh/trt/resnet50-v1-12.onnx
[I] ==== ONNX Model ====
    Name: mxnet_converted_model | ONNX Opset: 12
    ---- 1 Graph Input(s) ----
    {data [dtype=float32, shape=('N', 3, 224, 224)]}
    ---- 1 Graph Output(s) ----
    {resnetv17_dense0_fwd [dtype=float32, shape=('N', 1000)]}
    ---- 299 Initializer(s) ----
    ---- 175 Node(s) ----
```

It shows that the ONNX model has a graph input tensor named `data` whose shape is `('N', 3, 224, 224)`, where `'N'` represents that the dimension can be dynamic. Therefore, the `trtexec` flag to specify the input shapes with batch size 4 would be `--shapes=data:4x3x224x224`.

After running the `trtexec` command, `trtexec` will parse your ONNX file, build a TensorRT plan file, measure the performance of this plan file, and then print a performance summary, as follows:

```
[04/25/2024-23:57:45] [I] === Performance summary ===
[04/25/2024-23:57:45] [I] Throughput: 507.399 qps
[04/25/2024-23:57:45] [I] Latency: min = 1.96301 ms, max = 1.97534 ms, mean = 1.96921
 ms, median = 1.96917 ms, percentile(90%) = 1.97122 ms, percentile(95%) = 1.97229 ms,
 percentile(99%) = 1.97424 ms
[04/25/2024-23:57:45] [I] Enqueue Time: min = 0.0032959 ms, max = 0.0340576 ms, mean =
 0.00421173 ms, median = 0.00415039 ms, percentile(90%) = 0.00463867 ms, percentile(95%) =
 0.00476074 ms, percentile(99%) = 0.0057373 ms
[04/25/2024-23:57:45] [I] H2D Latency: min = 0 ms, max = 0 ms, mean = 0 ms, median = 0 ms,
 percentile(90%) = 0 ms, percentile(95%) = 0 ms, percentile(99%) = 0 ms
[04/25/2024-23:57:45] [I] GPU Compute Time: min = 1.96301 ms, max = 1.97534 ms, mean =
 1.96921 ms, median = 1.96917 ms, percentile(90%) = 1.97122 ms, percentile(95%) = 1.97229 ms,
 percentile(99%) = 1.97424 ms
[04/25/2024-23:57:45] [I] D2H Latency: min = 0 ms, max = 0 ms, mean = 0 ms, median = 0 ms,
 percentile(90%) = 0 ms, percentile(95%) = 0 ms, percentile(99%) = 0 ms
[04/25/2024-23:57:45] [I] Total Host Walltime: 3.00355 s
[04/25/2024-23:57:45] [I] Total GPU Compute Time: 3.00108 s
[04/25/2024-23:57:45] [I] Explanations of the performance metrics are printed in the verbose
 logs.
```

It prints a lot of performance metrics, but the two most important metrics are the Throughput and the median Latency. In this case, the ResNet-50 model with batch size 4 can run with a throughput of 507 inferences per second (which is 2028 images per second since the batch size is 4) and median latency of 1.969 ms.

Refer to Advanced Performance Measurement Techniques for explanations about what Throughput and Latency mean to your deep learning inference applications. Refer to trtexec for detailed explanations about other `trtexec` flags and other performance metrics that `trtexec` reports.

13.1.2. Performance Benchmarking with ONNX +Quantization

To enjoy the additional performance benefit from quantizations, Quantize/Dequantize operations need to be inserted into the ONNX model to tell TensorRT where to quantize/dequantize the tensors and what scaling factors to use.

Our recommended tool for ONNX quantization is the `ModelOptimizer` package. You can install it by running:

```
pip3 install --no-cache-dir --extra-index-url https://pypi.nvidia.com
        nvidia-modelopt
```

Using the `ModelOptimizer`, you can get a quantized ONNX model by running:

```
python3 -m modelopt.onnx.quantization --onnx_path resnet50-v1-12.onnx --
quantize_mode int8
        --output_path resnet50-v1-12-quantized.onnx
```

It loads the original ONNX model from `resnet50-v1-12.onnx`, runs calibration using random data, inserts Quantize/Dequantize ops into the graph, and then saves the ONNX model with Quantize/Dequantize ops to `resnet50-v1-12-quantized.onnx`.

Now that the new ONNX model contains the INT8 Quantize/Dequantize ops, we can run `trtexec` again using a similar command:

```
trtexec --onnx=resnet50-v1-12-quantized.onnx --shapes=data:4x3x224x224 --stronglyTyped --
noDataTransfers --useCudaGraph --useSpinWait
```

We are using the `--stronglyTyped` flag instead of the `--fp16` flag to require TensorRT to follow the data types in the quantized ONNX model strictly, including all the INT8 Quantize/Dequantize ops.

Here is an example output after running this `trtexec` command with the quantized ONNX model:

```
[04/26/2024-00:31:43] [I] === Performance summary ===
[04/26/2024-00:31:43] [I] Throughput: 811.74 qps
[04/26/2024-00:31:43] [I] Latency: min = 1.22559 ms, max = 1.23608 ms, mean = 1.2303
 ms, median = 1.22998 ms, percentile(90%) = 1.23193 ms, percentile(95%) = 1.23291 ms,
 percentile(99%) = 1.23395 ms
[04/26/2024-00:31:43] [I] Enqueue Time: min = 0.00354004 ms, max = 0.00997925 ms, mean =
 0.00431524 ms, median = 0.00439453 ms, percentile(90%) = 0.00463867 ms, percentile(95%) =
 0.00476074 ms, percentile(99%) = 0.00512695 ms
[04/26/2024-00:31:43] [I] H2D Latency: min = 0 ms, max = 0 ms, mean = 0 ms, median = 0 ms,
 percentile(90%) = 0 ms, percentile(95%) = 0 ms, percentile(99%) = 0 ms
[04/26/2024-00:31:43] [I] GPU Compute Time: min = 1.22559 ms, max = 1.23608 ms, mean =
 1.2303 ms, median = 1.22998 ms, percentile(90%) = 1.23193 ms, percentile(95%) = 1.23291 ms,
 percentile(99%) = 1.23395 ms
[04/26/2024-00:31:43] [I] D2H Latency: min = 0 ms, max = 0 ms, mean = 0 ms, median = 0 ms,
 percentile(90%) = 0 ms, percentile(95%) = 0 ms, percentile(99%) = 0 ms
[04/26/2024-00:31:43] [I] Total Host Walltime: 3.00219 s
[04/26/2024-00:31:43] [I] Total GPU Compute Time: 2.99824 s
[04/26/2024-00:31:43] [I] Explanations of the performance metrics are printed in the verbose
 logs.
```

As shown here, the Throughput is now 811 inferences per second and the median Latency is 1.23 ms. The Throughput has improved by 60% compared to FP16 performance results in the previous section.

13.1.3. Per-Layer Runtime and Layer Information

In previous sections, we described how to use `trtexec` to measure the end-to-end latency. In this section, we will show an example of per-layer runtime and per-layer information using `trtexec`. This will help you to find out how much latency each layer contributes to the end-to-end latency and in which layers the performance bottlenecks are.

This is an example `trtexec` command to print per-layer runtime and per-layer information using the quantized ResNet-50 ONNX model:

```
trtexec --onnx=resnet50-v1-12-quantized.onnx --shapes=data:4x3x224x224 --stronglyTyped --
noDataTransfers --useCudaGraph --useSpinWait --profilingVerbosity=detailed --dumpLayerInfo --
dumpProfile --separateProfileRun
```

The `--profilingVerbosity=detailed` flag enables detailed layer information capturing, `--dumpLayerInfo` flag shows the per-layer information in the log, and `--dumpProfile --separateProfileRun` flags show the per-layer runtime latencies in the log.

The following code is an example log of the per-layer information for one of the convolution layers in the quantized ResNet-50 model:

```
Name: resnetv17_stage1_conv0_weight + resnetv17_stage1_conv0_weight_QuantizeLinear
 + resnetv17_stage1_conv0_fwd, LayerType: CaskConvolution, Inputs: [ { Name:
 resnetv17_pool0_fwd_QuantizeLinear_Output_1, Location: Device, Dimensions:
 [4,64,56,56], Format/Datatype: Thirty-two wide channel vectorized row major Int8
 format }], Outputs: [ { Name: resnetv17_stage1_relu0_fwd_QuantizeLinear_Output,
 Location: Device, Dimensions: [4,64,56,56], 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: 64, Groups: 1, Weights: {"Type": "Int8", "Count":
 4096}, Bias: {"Type": "Float", "Count": 64}, HasBias: 1, HasReLU: 1, HasSparseWeights:
 0, HasDynamicFilter: 0, HasDynamicBias: 0, HasResidual: 0, ConvXAsActInputIdx:
 -1, BiasAsActInputIdx: -1, ResAsActInputIdx: -1, Activation: RELU, TacticName:
 sm80_xmma_fprop_implicit_gemm_interleaved_i8i8_i8i32_f32_nchw_vect_c_32kcrs_vect_c_32_nchw_vect_c_32_tilesi
 TacticValue: 0x483ad1560c6e5e27, StreamId: 0, Metadata: [ONNX Layer:
 resnetv17_stage1_conv0_fwd]
```

The log shows the layer name, the input and output tensor names, tensor shapes, tensor data types, convolution parameters, tactic name, and the metadata. The `Metadata` field shows which ONNX ops this layer corresponds to. Since TensorRT has graph fusion optimizations, one engine layer may correspond to multiple ONNX ops in the original model.

The following code is an example log of the per-layer runtime latencies for last few layers in the quantized ResNet-50 model:

```
[04/26/2024-00:42:55] [I]    Time(ms)     Avg.(ms)   Median(ms)   Time(%)   Layer
[04/26/2024-00:42:55] [I]       56.57       0.0255       0.0256       1.8
 resnetv17_stage4_conv7_weight + resnetv17_stage4_conv7_weight_QuantizeLinear +
 resnetv17_stage4_conv7_fwd
[04/26/2024-00:42:55] [I]      103.86       0.0468       0.0471       3.3
 resnetv17_stage4_conv8_weight + resnetv17_stage4_conv8_weight_QuantizeLinear +
 resnetv17_stage4_conv8_fwd
[04/26/2024-00:42:55] [I]       46.93       0.0211       0.0215       1.5
 resnetv17_stage4_conv9_weight + resnetv17_stage4_conv9_weight_QuantizeLinear +
 resnetv17_stage4_conv9_fwd + resnetv17_stage4__plus2 + resnetv17_stage4_activation2
[04/26/2024-00:42:55] [I]       34.64       0.0156       0.0154       1.1
 resnetv17_pool1_fwd
[04/26/2024-00:42:55] [I]       63.21       0.0285       0.0287
 2.0   resnetv17_dense0_weight + resnetv17_dense0_weight_QuantizeLinear +
 transpose_before_resnetv17_dense0_fwd + resnetv17_dense0_fwd + resnetv17_dense0_bias
 + ONNXTRT_Broadcast + unsqueeze_node_after_resnetv17_dense0_bias +
 ONNXTRT_Broadcast_ONNXTRT_Broadcast_output + (Unnamed Layer* 851) [ElementWise]
[04/26/2024-00:42:55] [I]     3142.40       1.4149       1.4162     100.0   Total
```

It shows that the median latency of the `resnetv17_pool1_fwd` layer is 0.0156 ms and contributes to 1.1% of the end-to-end latency. With this log, you can identify which layers take the largest portion of the end-to-end latency and is the performance bottleneck.

The `Total` latency reported in the per-layer runtime log is the summation of the per-layer latencies. It is typically slightly longer than the reported end-to-end latency due to the overheads caused by measuring per-layer latencies. For example, the `Total` median

latency is 1.4162 ms but the end-to-end latency shown in the previous section was 1.23 ms.

13.1.4. Performance Benchmarking with TensorRT Plan File

If you construct the TensorRT `INetworkDefinition` using TensorRT APIs and build the plan file in a separate script, you can still use `trtexec` to measure the performance of the plan file.

For example, if the plan file is saved as `resnet50-v1-12-quantized.plan`, then you can run the `trtexec` command to measure the performance using this plan file:

```
trtexec --loadEngine=resnet50-v1-12-quantized.plan --shapes=data:4x3x224x224 --
noDataTransfers --useCudaGraph --useSpinWait
```

The performance summary output is similar to those in the previous sections.

13.1.5. Duration and Number of Iterations

By default, `trtexec` warms up for at least 200 ms, runs inference for at least 10 iterations or at least 3 seconds, whichever is longer. You can modify these parameters by adding the `--warmUp=500`, `--iterations=100`, and `--duration=60` flags, which mean running the warm-up for at least 500 ms and running the inference for at least 100 iterations or at least 60 seconds, whichever is longer.

Refer to trtexec or run `trtexec --help` for a detailed explanation about other `trtexec` flags.

13.2. Advanced Performance Measurement Techniques

Before starting any optimization effort with TensorRT, it is essential to determine what should be measured. Without measurements, it is impossible to make reliable progress or measure whether success has been achieved.

Latency

A performance measurement for network inference is how much time elapses from an input being presented to the network until an output is available. This is the *latency* of the network for a single inference. Lower latencies are better. In some applications, low latency is a critical safety requirement. In other applications, latency is directly visible to users as a quality-of-service issue. For bulk processing, latency may not be important at all.

Throughput

Another performance measurement is how many inferences can be completed in a fixed unit of time. This is the *throughput* of the network. Higher throughput is better. Higher

throughputs indicate a more efficient utilization of fixed compute resources. For bulk processing, the total time taken will be determined by the throughput of the network.

Another way of looking at latency and throughput is to fix the maximum latency and measure throughput at that latency. A quality-of-service measurement like this can be a reasonable compromise between the user experience and system efficiency.

Before measuring latency and throughput, you must choose the exact points at which to start and stop timing. Depending on the network and application, it might make sense to choose different points.

In many applications, there is a processing pipeline, and the overall system performance can be measured by the latency and throughput of the entire processing pipeline. Because the pre- and post-processing steps depend so strongly on the particular application, this section considers the latency and throughput of the network inference only.

13.2.1. Wall-clock Timing

Wall-clock time (the elapsed time between the start of a computation and its end) can be useful for measuring the overall throughput and latency of the application, and for placing inference times in context within a larger system. C++11 provides high precision timers in the `<chrono>` standard library. For example, `std::chrono::system_clock` represents system-wide wall-clock time, and `std::chrono::high_resolution_clock` measures time in the highest precision available.

The following example code snippet shows measuring network inference host time:

C++

```
#include <chrono>

auto startTime = std::chrono::high_resolution_clock::now();
context->enqueueV3(stream);
cudaStreamSynchronize(stream);
auto endTime = std::chrono::high_resolution_clock::now();
float totalTime = std::chrono::duration<float, std::milli>
(endTime - startTime).count()
```

Python

```
import time
from cuda import cudart
err, stream = cudart.cudaStreamCreate()
start_time = time.time()
context.execute_async_v3(stream)
cudart.cudaStreamSynchronize(stream)
total_time = time.time() - start_time
```

If there is only one inference happening on the device at one time, then this can be a simple way of profiling the time-various operations take. Inference is typically asynchronous, so ensure you add an explicit CUDA stream or device synchronization to wait for results to become available.

13.2.2. CUDA Events

One problem with timing on the host exclusively is that it requires host/device synchronization. Optimized applications may have many inferences running in parallel on

the device with overlapping data movement. In addition, the synchronization itself adds some amount of noise to timing measurements.

To help with these issues, CUDA provides an Event API. This API allows you to place events into CUDA streams that will be time-stamped by the GPU as they are encountered. Differences in timestamps can then tell you how long different operations took.

The following example code snippet shows computing the time between two CUDA events:

C++

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

cudaEventRecord(start, stream);
context->(enqueueV3stream);
cudaEventRecord(end, stream);

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

Python

```
from cuda import cudart
err, stream = cudart.cudaStreamCreate()
err, start = cudart.cudaEventCreate()
err, end = cudart.cudaEventCreate()
cudart.cudaEventRecord(start, stream)
context.execute_async_v3(stream)
cudart.cudaEventRecord(end, stream)
cudart.cudaEventSynchronize(end)
err, total_time = cudart.cudaEventElapsedTime(start, end)
```

13.2.3. Built-In TensorRT Profiling

Digging deeper into the performance of inference requires more fine-grained timing measurements within the optimized network.

TensorRT has a *`Profiler`* (C++, Python) interface, which you can implement in order to have TensorRT pass profiling information to your application. When called, the network will run in a profiling mode. After finishing inference, the profiler object of your class is called to report the timing for each layer in the network. These timings can be used to locate bottlenecks, compare different versions of a serialized engine, and debug performance issues.

The profiling information can be collected from a regular inference `enqueueV3()` launch or a CUDA graph launch. Refer to `IExecutionContext::setProfiler()` and `IExecutionContext::reportToProfiler()` (C++, Python) for more information.

Layers inside a loop compile into a single monolithic layer, therefore, separate timings for those layers are not available. Also, some subgraphs (especially with Transformer-like networks) are handled by a next-generation graph optimizer that is not yet integrated with the `Profiler` APIs. For those networks, use CUDA Profiling Tools to profile per-layer performance.

An example showing how to use the `IProfiler` interface is provided in the common sample code (`common.h`).

You can also use `trtexec` to profile a network with TensorRT given an input network or plan file. Refer to the trtexec section for details.

13.2.4. CUDA Profiling Tools

The recommended CUDA profiler is NVIDIA Nsight™ Systems. Some CUDA developers may be more familiar with nvprof and nvvp, however, these are being deprecated. In any case, these profilers can be used on any CUDA program to report timing information about the kernels launched during execution, data movement between host and device, and CUDA API calls used.

Nsight Systems can be configured in various ways to report timing information for only a portion of the execution of the program or to also report traditional CPU sampling profile information together with GPU information.

The basic usage of Nsight Systems is to first run the command `nsys profile -o <OUTPUT> <INFERENCE_COMMAND>`, then, open the generated `<OUTPUT>.nsys-rep` file in the Nsight Systems GUI to visualize the captured profiling results.

Profile Only the Inference Phase

When profiling a TensorRT application, you should enable profiling only after the engine has been built. During the build phase, all possible tactics are tried and timed. Profiling this portion of the execution will not show any meaningful performance measurements and will include all possible kernels, not the ones actually selected for inference. One way to limit the scope of profiling is to:

- First phase: Structure the application to build and then serialize the engines in one phase.
- Second phase: Load the serialized engines and run inference in a second phase and profile this second phase only.

If the application cannot serialize the engines, or if the application must run through the two phases consecutively, you can also add `cudaProfilerStart()`/`cudaProfilerStop()` CUDA APIs around the second phase and add `-c cudaProfilerApi` flag to Nsight Systems command to profile only the part between `cudaProfilerStart()` and `cudaProfilerStop()`.

Understand Nsight Systems Timeline View

In the Nsight Systems Timeline View, the GPU activities are shown at the rows under CUDA HW and the CPU activities are shown at the rows under Threads. By default, the rows under CUDA HW are collapsed, therefore, you must click on it to expand the rows.

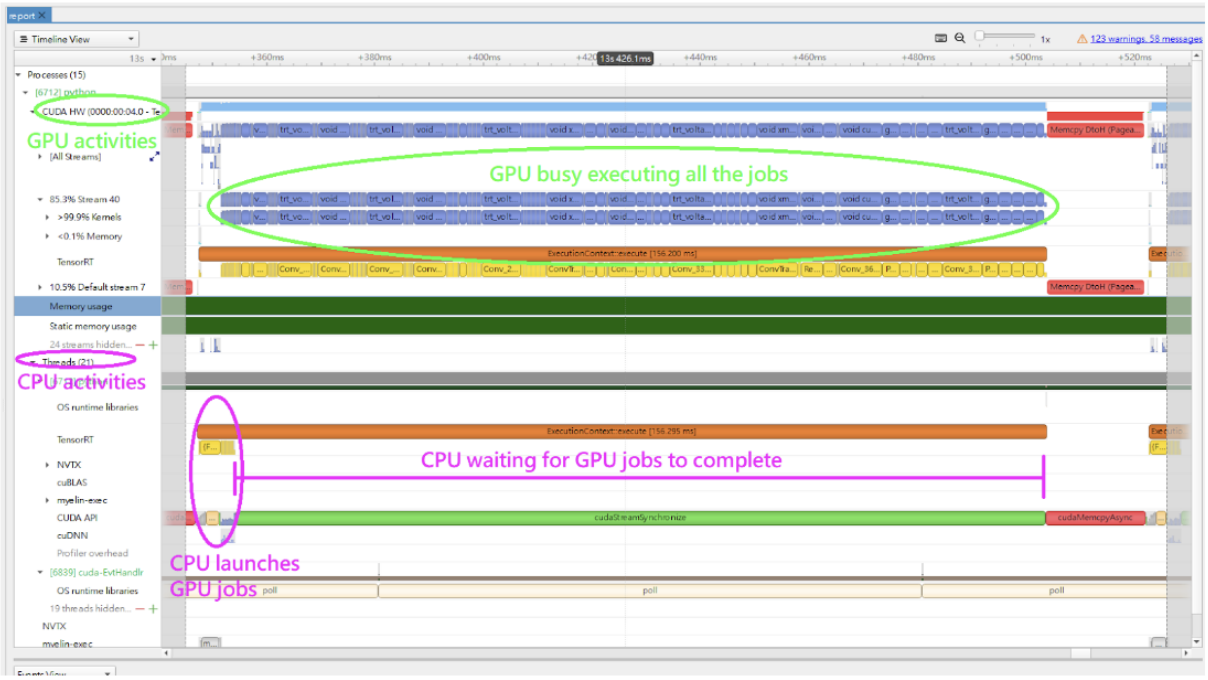
In a typical inference workflow, the application calls the `context->enqueueV3()` or `context->executeV3()` APIs to enqueue the jobs and then synchronize on the stream to wait until the GPU completes the jobs. It may appear as if the system is doing nothing for a while in the `cudaStreamSychronize()` call if you only look at the CPU activities. In

fact, the GPU may be busy executing the enqueued jobs while the CPU is waiting. The following figure shows an example timeline of the inference of a query.

The `trtexec` tool uses a slightly more complicated approach to enqueue the jobs by enqueuing the next query while GPU is still executing the jobs from the previous query. Refer to the trtexec section for more information.

The following image shows a typical view of the normal inference workloads in the Nsight Systems timeline view, showing CPU and GPU activities on different rows.

Figure 18. Normal Inference Workloads in Nsight Systems Timeline View



Use the NVTX Tracing in Nsight Systems

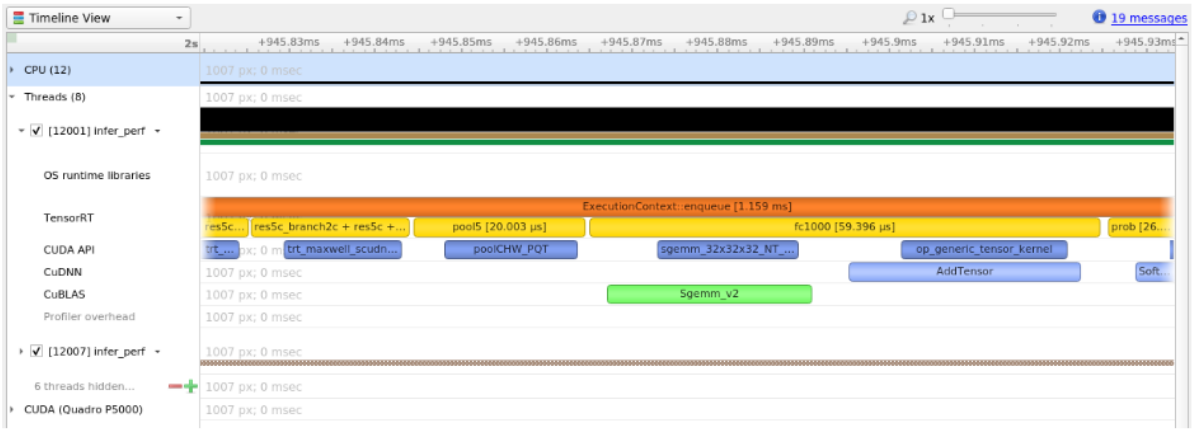
Enabling NVIDIA Tools Extension SDK (NVTX) tracing allows Nsight Compute and Nsight Systems to collect data generated by TensorRT applications. NVTX is a C-based API for marking events and ranges in your applications.

Decoding the kernel names back to layers in the original network can be complicated. Because of this, TensorRT uses NVTX to mark a range for each layer, which then allows the CUDA profilers to correlate each layer with the kernels called to implement it. In TensorRT, NVTX helps to correlate the runtime engine layer execution with CUDA kernel calls. Nsight Systems supports collecting and visualizing these events and ranges on the timeline. Nsight Compute also supports collecting and displaying the state of all active NVTX domains and ranges in a given thread when the application is suspended.

In TensorRT, each layer may launch one or more kernels to perform its operations. The exact kernels launched depend on the optimized network and the hardware present. Depending on the choices of the builder, there may be multiple additional operations that reorder data interspersed with layer computations; these reformat operations may be implemented as either device-to-device memory copies or as custom kernels.

For example, the following screenshots are from Nsight Systems.

Figure 19. The Layer Execution and the Kernel Being Launched on the CPU Side



The kernels actually run on the GPU, in other words, the following image shows the correlation between the layer execution and kernel launch on the CPU side and their execution on the GPU side.

Figure 20. The Kernels Run on the GPU

Control the Level of Details in NVTX Tracing

By default, TensorRT only shows layer names in the NVTX markers, while users can control the level of details by setting the ProfilingVerbosity in the `IBuilderConfig` when the engine is built. For example, to disable NVTX tracing, set the ProfilingVerbosity to `kNONE`:

C++

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

Python

```
builder_config.profilling_verbosity = trt.ProfilingVerbosity.NONE
```

On the other hand, you can choose to allow TensorRT to print more detailed layer information in the NVTX markers, including input and output dimensions, operations, parameters, tactic numbers, and so on, by setting the `ProfilingVerbosity` to `kDETAILED`:

C++

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

Python

```
builder_config.profilling_verbosity = trt.ProfilingVerbosity.DETAILED
```

Note: Enabling detailed NVTX markers increases the latency of `enqueueV3()` calls and could result in a performance drop if the performance depends on the latency of `enqueueV3()` calls.

Run Nsight Systems with `trtexec`

Below is an example of the commands to gather Nsight Systems profiles using trtexec tool:

```
trtexec --onnx=foo.onnx --profilingVerbosity=detailed --saveEngine=foo.plan
nsys profile -o foo_profile --capture-range cudaProfilerApi trtexec --
profilingVerbosity=detailed --loadEngine=foo.plan --warmUp=0 --duration=0 --iterations=50
```

The first command builds and serializes the engine to `foo.plan`, and the second command runs the inference using `foo.plan` and generates a `foo_profile.nsys-rep` file that can then be opened in the Nsight Systems user interface for visualization.

The `--profilingVerbosity=detailed` flag allows TensorRT to show more detailed layer information in the NVTX marking, and the `--warmUp=0 --duration=0 --iterations=50` flags allow you to control how many inference iterations to run. By default, `trtexec` runs inference for three seconds, which may result in a very large output `nsys-rep` file.

If CUDA graph is enabled, add `--cuda-graph-trace=node` flag to the `nsys` command to see the per-kernel runtime information:

```
nsys profile -o foo_profile --capture-range cudaProfilerApi --cuda-graph-trace=node trtexec
 --profilingVerbosity=detailed --loadEngine=foo.plan --warmUp=0 --duration=0 --iterations=50
 --useCudaGraph
```

(Optional) Enable GPU Metrics Sampling in Nsight Systems

On discrete GPU systems, add the `--gpu-metrics-device all` flag to the `nsys` command to sample GPU metrics, including GPU clock frequencies, DRAM bandwidth, Tensor Core utilization, and so on. If the flag is added, these GPU metrics appear in the Nsight Systems web interface.

13.2.4.1. Profiling for DLA

To profile DLA, add the `--accelerator-trace nvmedia` flag when using the NVIDIA Nsight Systems CLI or enable Collect other accelerators trace when using the user interface. For example, the following command can be used with the NVIDIA Nsight Systems CLI:

```
nsys profile -t cuda,nvtx,nvmedia,osrt --accelerator-trace=nvmedia  --show-output=true
 trtexec --loadEngine=alexnet_int8.plan --warmUp=0 --duration=0 --iterations=20
```

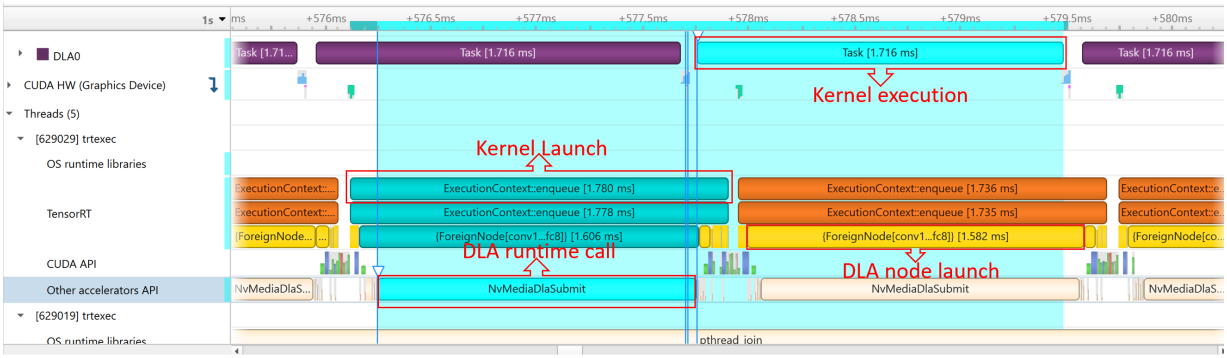
Below is an example report.

- `NvMediaDLASubmit` submits a DLA task for each DLA subgraph. The runtime of the DLA task can be found in the DLA timeline under Other accelerators trace.

- Because GPU fallback was allowed, some CUDA kernels were added by TensorRT automatically, like `permutationKernelPLC3` and `copyPackedKernel`, which are used for data reformatting.
- EGLStream APIs were executed because TensorRT usesEGLStreams for data transfer between GPU memory and DLA.

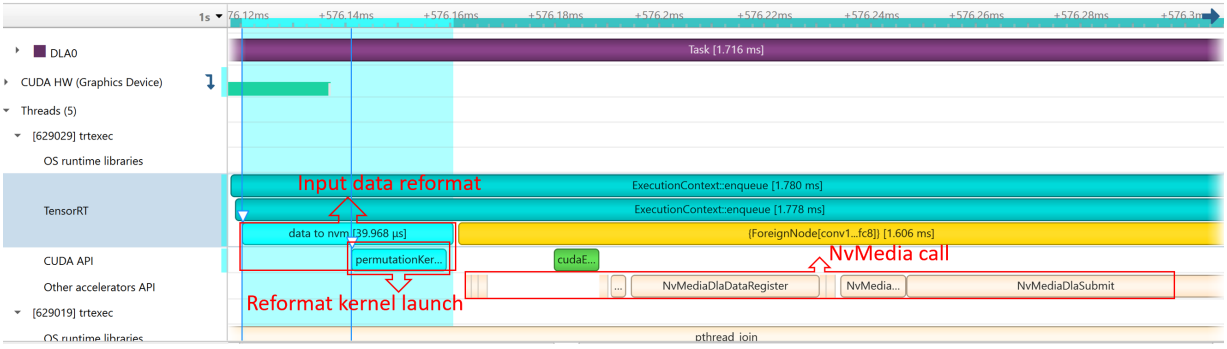
To maximize GPU utilization, `trtexec` enqueues the queries one batch ahead of time.

Figure 21. Sample DLA Profiling Report



The runtime of the DLA task can be found under *Other accelerator API*. Some CUDA kernels and EGLStream API are called for interaction between the GPU and DLA.

Figure 22. Sample DLA Profiling report



13.2.5. Tracking Memory

Tracking memory usage can be as important as execution performance. Usually, the memory will be more constrained on the device than on the host. To keep track of device memory, the recommended mechanism is to create a simple custom GPU allocator that internally keeps some statistics then uses the regular CUDA memory allocation functions `cudaMalloc` and `cudaFree`.

A custom GPU allocator can be set for the builder `IBuilder` for network optimizations, and for `IRuntime` when deserializing engines using the `IGpuAllocator` APIs. One idea for the custom allocator is to keep track of the current amount of memory allocated, and

to push an allocation event with a timestamp and other information onto a global list of allocation events. Looking through the list of allocation events allows profiling memory usage over time.

On mobile platforms, GPU memory and CPU memory share the system memory. On devices with very limited memory size, like Nano, system memory might run out with large networks; even the required GPU memory is smaller than system memory. In this case, increasing the system swap size could solve some problems. An example script is:

```
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
```

13.3. Hardware/Software Environment for Performance Measurements

Performance measurements are influenced by many factors, including hardware environment differences like cooling capability of the machine and software environment differences like GPU clock settings. This section summarizes a few items that may affect performance measurements.

Note that the items involving nvidia-smi are only supported on dGPU systems and not on the mobile systems.

13.3.1. GPU Information Query and GPU Monitoring

While measuring performance, it is recommended that you record and monitor the GPU status in parallel to the inference workload. Having the monitoring data allows you to identify possible root causes when you see unexpected performance measurements results.

Before the inference starts, call the `nvidia-smi -q` command to get the detailed information of the GPU, including the product name, power cap, clock settings, and so on. Then, while the inference workload is running, run the `nvidia-smi dmon -s pcu -f <FILE> -c <COUNT>` command in parallel to print out GPU clock frequencies, power consumption, temperature, and utilization to a file. Call `nvidia-smi dmon --help` for more options about the nvidia-smi device monitoring tool.

13.3.2. GPU Clock Locking and Floating Clock

By default, the GPU clock frequency is floating, meaning that the clock frequency sits at the idle frequency when there is no active workload, and it boosts to the boost clock frequency when the workload starts. This is usually the desired behavior in general since it allows the GPU to generate less heat at idle and to run at maximum speed when there is active workload.

Alternatively, you can lock the clock at a specific frequency by calling the `sudo nvidia-smi -lgc <freq>` command (and conversely, you can let the clock float again with the `sudo nvidia-smi -rgc` command). The supported clock frequencies can be found by the `sudo nvidia-smi -q -d SUPPORTED_CLOCKS` command. After the clock frequency is locked, it should stay at that frequency unless power throttling or thermal throttling take place, which will be explained in next sections. When the throttling kicks in, the device behaves as if the clock were floating.

Running TensorRT workloads with floating clocks or with throttling taking place can lead to more non-determinism in tactic selections and unstable performance measurements across inferences because every CUDA kernel may run at slightly different clock frequencies, depending on which frequency the driver boosts or throttles the clock to at that moment. On the other hand, running TensorRT workloads with locked clocks allows more deterministic tactic selections and consistent performance measurements, but the average performance will not be as good as when the clock is floating or is locked at maximum frequency with throttling taking place.

There is no definite recommendation on whether the clock should be locked or which clock frequency to lock the GPU at while running TensorRT workloads. It depends on whether the deterministic and stable performance or the best average performance is desired.

13.3.3. GPU Power Consumption and Power Throttling

Power throttling occurs when the average GPU power consumption reaches the power limit, which can be set by the `sudo nvidia-smi -pl <power_cap>` command. When this happens, the driver has to throttle the clock to a lower frequency to keep the average power consumption below the limit. The constantly changing clock frequencies may lead to unstable performance measurements if the measurements are taken within a short period of time, such as within 20ms.

Power throttling happens by design and is a natural phenomenon when the GPU clock is not locked or is locked at a higher frequency, especially for the GPUs with lower power limits such as NVIDIA T4 and NVIDIA A2 GPUs. To avoid performance variations caused by power throttling, you can lock the GPU clock at a lower frequency so that the performance numbers become more stable. However, the average performance numbers will be lower than the performance numbers with floating clocks or with the clock locked at a higher frequency even though power throttling would happen in this case.

Another issue with power throttling is that it may skew the performance numbers if there are gaps between inferences in your performance benchmarking applications. For example, if the application synchronizes at each inference, there will be periods of time when the GPU is idle between the inferences. The gaps cause the GPU to consume less power on average such that the clock is throttled less and the GPU can run at higher clock frequencies on average. However, the throughput numbers measured in this way are not accurate because when the GPU is fully loaded with no gaps between inferences, the actual clock frequency will be lower and the actual throughput will not reach the throughput numbers measured using the benchmarking application.

To avoid this, the `trtexec` tool is designed to maximize GPU execution by leaving nearly no gaps between GPU kernel executions so that it can measure the true throughput of a TensorRT workload. Therefore, if you see performance gaps between your benchmarking application and what trtexec reports, check if the power throttling and the gaps between inferences are the cause.

Lastly, power consumption can be dependent on the activation values, causing different performance measurements for different inputs. For example, if all the network input values are set to zeros or NaNs, GPU tends to consume less power than if the inputs are normal values because of fewer bit-flips in DRAM and in L2 cache. To avoid this discrepancy, always use the input values that best represent the actual value distribution when measuring the performance. The `trtexec` tool uses random input values by default, but you can specify the input by using the `--loadInputs` flag. Refer to the trtexec section for more information.

13.3.4. GPU Temperature and Thermal Throttling

Thermal throttling happens when the GPU temperature reaches a predefined threshold, which is around 85 degrees Celsius for most GPUs, and the driver has to throttle the clock to a lower frequency to prevent the GPU from overheating. You can tell this by seeing the temperature logged by the `nvidia-smi dmon` command gradually increasing while the inference workload is running, until it reaches ~85C and the clock frequency starts to drop.

If thermal throttling happens on actively cooled GPUs like Quadro A8000, then it is possible that the fans on the GPU are broken, or there are obstacles blocking the airflow.

If thermal throttling happens on passively cooled GPUs like NVIDIA A10, then it is likely that the GPUs are not properly cooled. Passively cooled GPUs require external fans or air conditioning to cool down the GPUs, and the airflow must go through the GPUs for effective cooling. Common cooling problems include installing GPUs in a server that is not designed for the GPUs or installing wrong numbers of GPUs into the server. In some cases, the air flows through the “easy path” (that is, the path with the least friction) around the GPUs instead of going through them. Fixing this requires examination of the airflow in the server and installation of airflow guidance if necessary.

Note that higher GPU temperature also leads to more leakage current in the circuits, which increases the power consumed by the GPU at a specific clock frequency. Therefore, for GPUs that are more likely to be power throttled like NVIDIA T4, poor cooling can lead to lower stabilized clock frequency with power throttling, and thus worse performance, even if the GPU clocks have not been thermally throttled yet.

On the other hand, ambient temperature, that is, the temperature of the environment around the server, does not usually affect GPU performance so long as the GPUs are properly cooled, except for GPUs with lower power limit whose performance may be slightly affected.

13.3.5. H2D/D2H Data Transfers and PCIe Bandwidth

On dGPU systems, often the input data must be copied from the host memory to the device memory (H2D) before an inference starts, and the output data must be copied back from device memory to host memory (D2H) after the inference. These H2D/D2H data transfers go through PCIe buses, and they can sometimes influence the inference performance or even become the performance bottleneck. The H2D/D2H copies can also be seen in the Nsight Systems profiles, appearing as `cudaMemcpy()` or `cudaMemcpyAsync()` CUDA API calls.

To achieve maximum throughput, the H2D/D2H data transfers should run in parallel to the GPU executions of other inferences so that the GPU does not sit idle when the H2D/D2H copies take place. This can be done by running multiple inferences in different streams in parallel, or by launching H2D/D2H copies in a different stream than the stream used for GPU executions and using CUDA events to synchronize between the streams. The trtexec tool shows as an example for the latter implementation.

When the H2D/D2H copies run in parallel to GPU executions, they can interfere with the GPU executions especially if the host memory is pageable, which is the default case. Therefore, it is recommended that you allocate pinned host memory for the input and output data using `cudaHostAlloc()` or `cudaMallocHost()` CUDA APIs.

To check whether the PCIe bandwidth becomes the performance bottleneck, you can check the Nsight Systems profiles and see if the H2D or D2H copies of an inference query have longer latencies than the GPU execution part. If PCIe bandwidth becomes the performance bottleneck, here are a few possible solutions.

First, check whether the PCIe bus configuration of the GPU is correct in terms of which generation (for example, Gen3 or Gen4) and how many lanes (for example, x8 or x16) are used. Next, try reducing the amount of data that must be transferred using the PCIe bus. For example, if the input images have high resolutions and the H2D copies become the bottleneck, then you can consider transmitting JPEG-compressed images over the PCIe bus and decode the image on the GPUs before the inference workflow, instead of transmitting raw pixels. Finally, you can consider using NVIDIA GPUDirect technology to load data directly from/to the network or the filesystems without going through the host memory.

In addition, if your system has AMD x86_64 CPUs, check the NUMA (Non-Uniform Memory Access) configurations of the machine with `numactl --hardware` command. The PCIe bandwidth between a host memory and a device memory located on two different NUMA nodes is much more limited than the bandwidth between the host/device memory located on the same NUMA node. Allocate the host memory on the NUMA node on which the GPU that the data will be copied to resides. Also, pin the CPU threads that will trigger the H2D/D2H copies on that specific NUMA node.

Note that on mobile platforms, the host, and the device share the same memory, so the H2D/D2H data transfers are not required if the host memory is allocated using CUDA APIs and is pinned instead of being pageable.

By default, the `trtexec` tool measures the latencies of the H2D/D2H data transfers that tell the user if the TensorRT workload may be bottlenecked by the H2D/D2H copies. However, if the H2D/D2H copies affect the stability of the GPU Compute Time, you can add the `--noDataTransfers` flag to disable H2D/D2H transfers to measure only the latencies of the GPU execution part.

13.3.6. TCC Mode and WDDM Mode

On Windows machines, there are two driver modes: you can configure the GPU to be in the TCC mode and the WDDM mode. The mode can be specified by calling the `sudo nvidia-smi -dm [0|1]` command, but a GPU connected to a display shall not be configured into TCC mode Refer to the TCC mode documentation for more information and limitations about TCC mode.

In TCC mode, the GPU is configured to focus on computation work and the graphics support like OpenGL or monitor display are disabled. This is the recommended mode for GPUs that run TensorRT inference workloads. On the other hand, the WDDM mode tends to cause GPUs to have worse and unstable performance results when running inference workloads using TensorRT.

This is not applicable to Linux-based OS.

13.3.7. Enqueue-Bound Workloads and CUDA Graphs

The `enqueue()` function of `IExecutionContext` is asynchronous, that is, it returns immediately after all the CUDA kernels are launched without waiting for the completion of CUDA kernel executions. However, in some cases, the `enqueue()` time can take longer than the actual GPU executions, causing the latency of `enqueue()` calls to become the performance bottleneck. We say that this type of workload is "enqueue-bound". There are two reasons that may cause a workload to be enqueue-bound.

First, if the workload is very tiny in terms of the amount of computations, such as containing convolutions with small I/O sizes, matrix multiplications with small GEMM sizes, or mostly element-wise operations throughout the network, then the workload tends to be enqueue-bound. This is because most CUDA kernels take the CPU and the driver around 5-15 microseconds to launch per kernel, so if each CUDA kernel execution time is only several microseconds long on average, the kernel launching time becomes the main performance bottleneck.

To solve this, try to increase the amount of the computation per CUDA kernel, such as by increasing the batch size. Or you can use CUDA Graphs to capture the kernel launches into a graph and launch the graph instead of calling `enqueueV3()`.

Second, if the workload contains operations that require device synchronizations, such as loops or if-else conditions, then the workload is naturally enqueue-bound. In this case, increasing the batch size may help improve the throughput without increasing the latency much.

In `trtexec`, you can tell that a workload is enqueue-bound if the reported `Enqueue Time` is close to or longer than the reported `GPU Compute Time`. In this case, it is recommended that you add the `--useCudaGraph` flag to enable CUDA graphs in `trtexec`,

which will reduce the `Enqueue Time` as long as the workload does not contain any synchronization operations.

13.3.8. `BlockingSync` and `SpinWait` Synchronization Modes

If the performance is measured with `cudaStreamSynchronize()` or `cudaEventSynchronize()`, the variations in synchronization overhead may lead to variations in performance measurements. This section describes the cause of the variations and how to avoid them.

When `cudaStreamSynchronize()` is called, there are two ways in which the driver waits until the completion of the stream. If the `cudaDeviceScheduleBlockingSync` flag has been set with `cudaSetDeviceFlags()` calls, then the `cudaStreamSynchornize()` uses the blocking-sync mechanism. Otherwise, it uses the spin-wait mechanism.

The similar idea applies to CUDA events. If a CUDA event is created with the `cudaEventDefault` flag, then the `cudaEventSynchronize()` call uses the spin-wait mechanism; and if a CUDA event is created with the `cudaEventBlockingSync` flag, then the `cudaEventSynchronize()` call will use the blocking-sync mechanism.

When the blocking-sync mode is used, the host thread yields to another thread until the device work is done. This allows the CPUs to sit idle to save power or to be used by other CPU workloads when the device is still executing. However, the blocking-sync mode tends to result in relatively unstable overheads in stream/event synchronizations in some OS, which in terms lead to variations in latency measurements.

On the other hand, when the spin-wait mode is used, the host thread is constantly polling until the device work is done. Using spin-wait makes the latency measurements more stable due to shorter and more stable overhead in stream/event synchronizations, but it consumes some CPU computation resources and leads to more power consumption by the CPUs.

Therefore, if you want to reduce CPU power consumption, or if you do not want the stream/event synchronizations to consume CPU resources (for example, you are running other heavy CPU workloads in parallel), use the blocking-sync mode. If you care more about stable performance measurements, use the spin-wait mode.

In `trtexec`, the default synchronization mechanism is blocking-sync mode. Add the `--useSpinWait` flag to enable synchronizations using the spin-wait mode for more stable latency measurements, at the cost of more CPU utilizations and power consumptions.

13.4. Optimizing TensorRT Performance

The following sections focus on the general inference flow on GPUs and some of the general strategies to improve performance. These ideas are applicable to most CUDA programmers but may not be as obvious to developers coming from other backgrounds.

13.4.1. Batching

The most important optimization is to compute as many results in parallel as possible using batching. In TensorRT, a *batch* is a collection of inputs that can all be processed uniformly. Each instance in the batch has the same shape and flows through the network in exactly the same way. Each instance can, therefore, be trivially computed in parallel.

Each layer of the network will have some amount of overhead and synchronization required to compute forward inference. By computing more results in parallel, this overhead is paid off more efficiently. In addition, many layers are performance-limited by the smallest dimension in the input. If the batch size is one or small, this size can often be the performance-limiting dimension. For example, the FullyConnected layer with `V` inputs and `K` outputs can be implemented for one batch instance as a matrix multiply of an `1xV` matrix with a `VxK` weight matrix. If `N` instances are batched, this becomes an `NxV` multiplied by the `VxK` matrix. The vector-matrix multiplier becomes a matrix-matrix multiplier, which is much more efficient.

Larger batch sizes are almost always more efficient on the GPU. Extremely large batches, such as `N > 2^16`, can sometimes require extended index computation and so should be avoided if possible. But generally, increasing the batch size improves total throughput. In addition, when the network contains MatrixMultiply layers or FullyConnected layers, batch sizes of multiples of 32 tend to have the best performance for FP16 and INT8 inference because of the utilization of Tensor Cores, if the hardware supports them.

On NVIDIA Ada Lovelace GPUs or later GPUs, it is possible that decreasing the batch size may improve the throughput significantly if the smaller batch sizes happen to help the GPU to cache the input/output values in the L2 cache. Therefore, try various batch sizes to get the batch size for the optimal performance.

Sometimes batching inference work is not possible due to the organization of the application. In some common applications, such as a server that does inference per request, it can be possible to implement opportunistic batching. For each incoming request, wait for a time `T`. If other requests come in during that time, batch them together. Otherwise, continue with a single instance inference. This type of strategy adds fixed latency to each request but can improve the maximum throughput of the system by orders of magnitude.

The NVIDIA Triton Inference Server provides a simple way to enable dynamic batching with TensorRT engines.

Using batching

The batch dimension is part of the tensor dimensions, and you can specify the range of the batch sizes and the batch size to optimize the engine for by adding optimization profiles. Refer to the Working with Dynamic Shapes section for more details.

13.4.2. Within-Inference Multi-Streaming

In general, CUDA programming streams are a way of organizing asynchronous work. Asynchronous commands put into a stream are guaranteed to run in sequence but may execute out of order with respect to other streams. In particular, asynchronous

commands in two streams may be scheduled to run concurrently (subject to hardware limitations).

In the context of TensorRT and inference, each layer of the optimized final network will require work on the GPU. However, not all layers will be able to fully use the computation capabilities of the hardware. Scheduling requests in separate streams allows work to be scheduled immediately as the hardware becomes available without unnecessary synchronization. Even if only some layers can be overlapped, overall performance will improve.

Starting in TensorRT 8.6, you can use the `IBuilderConfig::setMaxAuxStreams()` API to set the maximum number of auxiliary streams that TensorRT is allowed to use to run multiple layers in parallel. The auxiliary streams are in contrast to the “main stream” provided in the `enqueueV3()` call, and if enabled, TensorRT will run some layers on the auxiliary streams in parallel to the layers running on the mainstream.

For example, to run the inference on at most eight streams (that is, seven auxiliary streams and one mainstream) in total:

C++

```
config->setMaxAuxStreams(7)
```

Python

```
config.max_aux_streams = 7
```

Note that this only sets the maximum number of auxiliary streams, however, TensorRT may end up using fewer auxiliary streams than this number if it determines that using more streams does not help.

To get the actual number of auxiliary streams that TensorRT uses for an engine, run:

C++

```
int32_t nbAuxStreams = engine->getNbAuxStreams()
```

Python

```
num_aux_streams = engine.num_aux_streams
```

When an execution context is created from the engine, TensorRT automatically creates the auxiliary streams needed to run the inference. However, you can also specify the auxiliary streams you would like TensorRT to use:

C++

```
int32_t nbAuxStreams = engine->getNbAuxStreams();
std::vector<cudaStream_t> streams(nbAuxStreams);
for (int32_t i = 0; i < nbAuxStreams; ++i)
{
    cudaStreamCreate(&streams[i]);
}
context->setAuxStreams(streams.data(), nbAuxStreams);
```

Python

```
from cuda import cudart
num_aux_streams = engine.num_aux_streams
streams = []
for i in range(num_aux_streams):
    err, stream = cudart.cudaStreamCreate()
    streams.append(stream)
context.set_aux_streams(streams)
```

TensorRT will always insert event synchronizations between the main stream provided using `enqueueV3()` call and the auxiliary streams:

- At the beginning of the `enqueueV3()` call, TensorRT will make sure that all the auxiliary streams wait on the activities on the mainstream.
- At the end of the `enqueueV3()` call, TensorRT will make sure that the main stream waits on the activities on all the auxiliary streams.

Note that enabling auxiliary streams may increase the memory consumption because some activation buffers will no longer be able to be reused.

13.4.3. Cross-Inference Multi-Streaming

In addition to the within-inference streaming, you can also enable streaming between multiple execution contexts. For example, you can build an engine with multiple optimization profiles and create an execution context per profile. Then, call the `enqueueV3()` function of the execution contexts on different streams to allow them to run in parallel.

Running multiple concurrent streams often leads to situations where several streams share compute resources at the same time. This means that the network may have less compute resources available during inference than when the TensorRT engine was being optimized. This difference in resource availability can cause TensorRT to choose a kernel that is suboptimal for the actual runtime conditions. In order to mitigate this effect, you can limit the amount of available compute resources during engine creation to more closely resemble actual runtime conditions. This approach generally promotes throughput at the expense of latency. For more information, refer to Limiting Compute Resources.

It is also possible to use multiple host threads with streams. A common pattern is incoming requests dispatched to a pool of waiting for worker threads. In this case, the pool of worker threads will each have one execution context and CUDA stream. Each thread will request work in its own stream as the work becomes available. Each thread will synchronize with its stream to wait for results without blocking other worker threads.

13.4.4. CUDA Graphs

CUDA graphs are a way to represent a sequence (or more generally a graph) of kernels in a way that allows their scheduling to be optimized by CUDA. This can be particularly useful when your application performance is sensitive to the CPU time taken to enqueue the kernels.

TensorRT's `enqueuev3()` method supports CUDA graph capture for models that require no CPU interaction mid-pipeline. For example:

C++

```
// Call enqueueV3() once after an input shape change to update internal state.
context->enqueueV3(stream);

// Capture a CUDA graph instance
cudaGraph_t graph;
cudaGraphExec_t instance;
cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
context->enqueueV3(stream);
cudaStreamEndCapture(stream, &graph);
cudaGraphInstantiate(&instance, graph, 0);
```

```
// To run inferences, launch the graph instead of calling enqueueV3().
for (int i = 0; i < iterations; ++i) {
    cudaGraphLaunch(instance, stream);
    cudaStreamSynchronize(stream);
}
```

Python

```
from cuda import cudart
err, stream = cudart.cudaStreamCreate()

# Call execute_async_v3() once after an input shape change to update internal state.
context.execute_async_v3(stream);

# Capture a CUDA graph instance
cudaStreamBeginCapture(stream, cudart.cudaStreamCaptureModeGlobal)
context.execute_async_v3(stream)
err, graph = cudart.cudaStreamEndCapture(stream)
err, instance = cudart.cudaGraphInstantiate(graph, 0)

# To run inferences, launch the graph instead of calling execute_async_v3().
for i in range(iterations):
    cudart.cudaGraphLaunch(instance, stream)
    cudart.cudaStreamSynchronize(stream)
```

Models for which graphs are not supported include those with loops or conditionals. In this case, `cudaStreamEndCapture()` will return `cudaErrorStreamCapture*` errors, indicating that the graph capturing has failed, but the context can continue to be used for normal inference without CUDA graphs.

When capturing a graph, it is important to account for the two-phase execution strategy used in the presence of dynamic shapes.

1. Update internal state of the model to account for any changes in input size.
2. Stream work to the GPU

For models where input size is fixed at build time, the first phase requires no per-invocation work. Otherwise, if the input sizes have changed since the last invocation, some work may be required to update derived properties.

The first phase of work is not designed to be captured, and even if the capture is successful may increase model execution time. Therefore, after changing the shapes of inputs or the values of shape tensors, call `enqueueV3()` once to flush deferred updates before capturing the graph.

Graphs captured with TensorRT are specific to the input size for which they were captured, and also to the state of the execution context. Modifying the context from which the graph was captured will result in undefined behavior when executing the graph - in particular, if the application is providing its own memory for activations using `createExecutionContextWithoutDeviceMemory()`, the memory address is also captured as part of the graph. Binding locations are also captured as part of the graph.

Therefore, the best practice is to use one execution context per captured graph, and to share memory across the contexts with `createExecutionContextWithoutDeviceMemory()`.

`trtexec` allows you to check whether the built TensorRT engine is compatible with CUDA graph capture. Refer to the trtexec section for more information.

13.4.5. Enabling Fusion

13.4.5.1. Layer Fusion

TensorRT attempts to perform many different types of optimizations in a network during the build phase. In the first phase, layers are fused together whenever possible. Fusions transform the network into a simpler form but preserve the same overall behavior. Internally, many layer implementations have extra parameters and options that are not directly accessible when creating the network. Instead, the fusion optimization step detects supported patterns of operations and fuses multiple layers into one layer with internal options set.

Consider the common case of a convolution followed by ReLU activation. To create a network with these operations, it involves adding a Convolution layer with `addConvolution`, following it with an Activation layer using `addActivation` with an `ActivationType` of `kRELU`. The unoptimized graph will contain separate layers for convolution and activation. The internal implementation of convolution supports computing the ReLU function on the output in one step directly from the convolution kernel without requiring a second kernel call. The fusion optimization step will detect the convolution followed by ReLU. Verify that the operations are supported by the implementation, then fuse them into one layer.

To investigate which fusions have happened, or have not happened, the builder logs its operations to the logger object provided during construction. Optimization steps are at the `kINFO` log level. To see these messages, ensure you log them in the `ILogger` callback.

Fusions are normally handled by creating a new layer with a name containing the names of both of the layers, which were fused. For example, in MNIST, a FullyConnected layer (InnerProduct) named `ip1` is fused with a ReLU Activation layer named `relu1` to create a new layer named `ip1 + relu1`.

13.4.5.2. Types of Fusions

The following list describes the types of supported fusions.

Supported Layer Fusions

ReLU Activation

An Activation layer performing ReLU followed by an activation performing ReLU will be replaced by a single activation layer.

Convolution and ReLU Activation

The Convolution layer can be of any type and there are no restrictions on values. The Activation layer must be ReLU type.

Convolution and GELU Activation

The precision of input and output should be the same; with both of them FP16 or INT8. The Activation layer must be GELU type. TensorRT should be running on an NVIDIA Turing or later device with CUDA version 10.0 or later.

Convolution and Clip Activation
The Convolution layer can be any type and there are no restrictions on values. The Activation layer must be Clip type.

Scale and Activation
The Scale layer followed by an Activation layer can be fused into a single Activation layer.

Convolution and ElementWise Operation
A Convolution layer followed by a simple sum, min, or max in an ElementWise layer can be fused into the Convolution layer. The sum must not use broadcasting, unless the broadcasting is across the batch size.

Padding and Convolution/Deconvolution
Padding followed by a Convolution or Deconvolution can be fused into a single Convolution/Deconvolution layer if all the padding sizes are non-negative.

Shuffle and Reduce
A Shuffle layer without reshape, followed by a Reduce layer can be fused into a single Reduce layer. The Shuffle layer can perform permutations but cannot perform any reshape operation. The Reduce layer must have a `keepDimensions` set of dimensions.

Shuffle and Shuffle
Each Shuffle layer consists of a transpose, a reshape, and a second transpose. A Shuffle layer followed by another Shuffle layer can be replaced by a single Shuffle (or nothing). If both Shuffle layers perform reshape operations, this fusion is only allowed if the second transpose of the first shuffle is the inverse of the first transpose of the second shuffle.

Scale
A Scale layer that adds `0`, multiplied by `1`, or computes powers to the `1` can be erased.

Convolution and Scale
A Convolution layer followed by a Scale layer that is `kUNIFORM` or `kCHANNEL` can be fused into a single convolution by adjusting the convolution weights. This fusion is disabled if the scale has a non-constant `power` parameter.

Convolution and Generic Activation
This fusion happens after the pointwise fusion mentioned below. A pointwise with one input and one output can be called as a generic activation layer. A convolution layer followed by a generic activation layer can be fused into a single convolution layer.

Reduce
A Reduce layer that performs average pooling will be replaced by a Pooling layer. The Reduce layer must have a `keepDimensions` set, reduced across `H` and `W` dimensions from `CHW` input format before batching, using the `kAVG` operation.

Convolution and Pooling
The Convolution and Pooling layers must have the same precision. The Convolution layer may already have a fused activation operation from a previous fusion.

Depthwise Separable Convolution
A depthwise convolution with activation followed by a convolution with activation may sometimes be fused into a single optimized DepSepConvolution layer. The precision of

both convolutions must be INT8 and the device's computes capability must be 7.2 or later.

SoftMax and Log
It can be fused into a single SoftMax layer if the SoftMax has not already been fused with a previous log operation.

SoftMax and TopK
Can be fused into a single layer. The SoftMax may or may not include a Log operation.

FullyConnected
The FullyConnected layer will be converted into the Convolution layer. All fusions for convolution will take effect.

Supported Reduction Operation Fusions

GELU
A group of Unary layer and ElementWise layer that represents the following equations can be fused into a single GELU reduction operation.

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

Or the alternative representation:

$$0.5x \times \left(1+\operatorname{erf}\left(x/\sqrt{2}\right)\right)$$

L1Norm
A Unary layer `kABS` operation followed by a Reduce layer `kSUM` operation can be fused into a single L1Norm reduction operation.

Sum of Squares
A product ElementWise layer with the same input (square operation) followed by a `kSUM` reduction can be fused into a single square Sum reduction operation.

L2Norm
A sum of squares operation followed by a `kSQRT` UnaryOperation can be fused into a single L2Norm reduction operation.

LogSum
A Reduce layer `kSUM` followed by a `kLOG` UnaryOperation can be fused into a single LogSum reduction operation.

LogSumExp
A Unary `kEXP` ElementWise operation followed by a LogSum fusion can be fused into a single LogSumExp reduction.

13.4.5.3. PointWise Fusion

Multiple adjacent PointWise layers can be fused into a single PointWise layer, to improve performance.

The following types of PointWise layers are supported, with some limitations:

`Activation`
Every `ActivationType` is supported.

Constant
Only constant with a single value (size == 1).

ElementWise
Every `ElementWiseOperation` is supported.

PointWise
`PointWise` itself is also a PointWise layer.

Scale
Only support `ScaleMode::kUNIFORM`.

Unary
Every `UnaryOperation` is supported.

The size of the fused PointWise layer is not unlimited, therefore, some PointWise layers may not be fused.

Fusion creates a new layer with a name consisting of both of the layers, which were fused. For example, an ElementWise layer named `add1` is fused with a ReLU Activation layer named `relu1` with a new layer name: `fusedPointwiseNode(add1, relu1)`.

13.4.5.4. Q/DQ Fusion

For an explanation and suggestions on optimizations of INT8 and FP8 networks containing `QuantizeLinear` and `DequantizeLinear` layers, refer to Explicit Quantization.

13.4.5.5. Multi-Head Attention Fusion

Multi-head attention (MHA) computes `softmax(Q * K^T / scale + mask) * V` where `Q` is query embeddings, `K` is key embeddings, and `V` is value embeddings. The shape of `Q` is `[B, N, S_q, H]` and the shapes of `K` and `V` are `[B, N, S_kv, H]` where `B` is batch size, `N` is number of attention heads, `H` is the head/hidden size, `S_q` and `S_kv` are the sequence lengths of query and key/value respectively.

The following is a list of restrictions for MHA to be fused into a single kernel.

- SM version must be >= 75.
- The input types of the two batched matrix multiplications must be the same and must be FP16, INT8 (refer to the following regarding quantize and dequantize layer placement), or BF16.
- Head size `H` must satisfy the constraints 16 <= H <= 256 and H % 8 == 0 for FP16 and BF16.
- Head size must be 16, 32, or 64 and sequence lengths (`S_q`, `S_kv`) must be <= 512 for INT8.
- INT8 fused MHA will be generated only if quantize and dequantize layers are placed before the first batched matrix multiplication, after softmax, and after the second batched matrix multiplication.
- TensorRT may decide not to fuse an MHA graph into a single kernel based on performance evaluation or other constraints.

13.4.6. Limiting Compute Resources

Limiting the number of compute resources available to TensorRT during engine creation is beneficial when the reduced amount better represents the expected conditions during runtime. For example, when the GPU is expected to be performing additional work in parallel to the TensorRT engine or when the engine is expected to be run on a different GPU with less resources (note that the recommended approach is to build the engine on the GPU that will be used for inference, but this may not always be feasible).

You can limit the number of available compute resources with the following steps:

1. Start the CUDA MPS control daemon.
   ```
   nvidia-cuda-mps-control -d
   ```
2. Set the number of compute resources to use with the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable. For example, `export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=50`.
3. Build the network engine.
4. Stop the CUDA MPS control daemon.
   ```
   echo quit | nvidia-cuda-mps-control
   ```

The resulting engine is optimized to the reduced number of compute cores (50% in this example) and provides better throughput when using similar conditions during inference. You are encouraged to experiment with different amounts of streams and different MPS values to determine the best performance for your network.

For more details about `nvidia-cuda-mps-control`, refer to the nvidia-cuda-mps-control documentation and the relevant GPU requirements here.

13.4.7. Deterministic Tactic Selection

In the engine building phase, TensorRT runs through all the possible tactics and selects the fastest ones. Since the selection is based on the latency measurements of the tactics, TensorRT may end up selecting different tactics across different runs if some tactics have very similar latencies. Therefore, different engines built from the same `INetworkDefinition` may behave slightly differently in terms of output values and performance. You can inspect the selected tactics of an engine by using the Engine Inspector or by turning on verbose logging while building the engine.

If deterministic tactic selection is desired, the following lists a few suggestions that may help improve the determinism of tactic selection.

Locking GPU Clock Frequency

By default, the clock frequency of the GPU is not locked, meaning that the GPU normally sits at the idle clock frequency and only boosts to the max clock frequency when there are active GPU workloads. However, there is a latency for the clock to be boosted from the idle frequency and that may cause performance variations while TensorRT is running through the tactics and selecting the best ones, resulting in non-deterministic tactic selections.

Therefore, locking the GPU clock frequency before starting to build a TensorRT engine may improve the determinism of tactic selection. You can lock the GPU clock frequency by calling the `sudo nvidia-smi -lgc <freq>` command, where `<freq>` is the desired frequency to lock at. You can call `nvidia-smi -q -d SUPPORTED_CLOCKS` to find the supported clock frequencies by the GPU.

Therefore, locking the GPU clock frequency before starting to build a TensorRT engine may improve the determinism of tactic selection. Refer to the Hardware/Software Environment for Performance Measurements section for more information about how to lock and monitor the GPU clock and the factors that may affect GPU clock frequencies.

Increasing Average Timing Iterations

By default, TensorRT runs each tactic for at least four iterations and takes the average latency. You can increase the number of iterations by calling the `setAvgTimingIterations()` API:

C++

```
builderConfig->setAvgTimingIterations(8);
```

Python

```
Builder_config.avg_timing_iterations = 8
```

Increasing the number of average timing iterations may improve the determinism of tactic selections, but the required engine building time will become longer.

Using Timing Cache

Timing Cache records the latencies of each tactic for a specific layer configuration. The tactic latencies are reused if TensorRT encounters another layer with an identical configuration. Therefore, by reusing the same timing cache across multiple engine buildings runs with the same `INetworkDefinition` and builder config, you can make TensorRT select an identical set of tactics in the resulting engines.

Refer to the Timing Cache section for more information.

13.4.8. Overhead of Shape Change and Optimization Profile Switching

After the `IExecutionContext` switches to a new optimization profile, or the shapes of the input bindings change, TensorRT must recompute the tensor shapes throughout the network and recompute the resources needed by some tactics for the new shapes before the next inference can start. That means, the first `enqueue()` call after a shape/profile change may be longer than the subsequent `enqueue()` calls.

Optimizing the cost of shape/profile switching is an active area of development. However, there are still a few cases where the overhead can influence the performance of the inference applications. For example, some convolution tactics for NVIDIA Volta GPUs or older GPUs have much longer shape/profile switching overhead, even if their inference performance is the best among all the available tactics. In this case, disabling

`kEDGE_MASK_CONVOLUTIONS` tactics from tactic sources when building the engine may help reduce the overhead of shape/profile switching.

13.5. Optimizing Layer Performance

The following descriptions detail how you can optimize the listed layers.

Gather Layer

To get the maximum performance out of a Gather layer, use an `axis` of `0`. There are no fusions available for a Gather layer.

Reduce Layer

To get the maximum performance out of a Reduce layer, perform the reduction across the last dimensions (tail reduce). This allows optimal memory to read/write patterns through sequential memory locations. If doing common reduction operations, express the reduction in a way that will be fused to a single operation if possible.

RNN Layer

If possible, opt to use the newer RNNv2 interface in preference to the legacy RNN interface. The newer interface supports variable sequence lengths and variable batch sizes, as well as having a more consistent interface. To get maximum performance, larger batch sizes are better. In general, sizes that are multiples of 64 achieve highest performance. Bidirectional RNN-mode prevents wavefront propagation because of the added dependency, therefore, it tends to be slower.

In addition, the newly introduced Loop-based API provides a much more flexible mechanism to use general layers within recurrence without being limited to a small set of predefined RNNv2 interface. The `ILoopLayer` recurrence enables a rich set of automatic loop optimizations, including loop fusion, unrolling, and loop-invariant code motion, to name a few. For example, significant performance gains are often obtained when multiple instances of the same MatrixMultiply or FullyConnected layer are properly combined to maximize machine utilization after loop unrolling along the sequence dimension. This works best if you can avoid a MatrixMultiply or FullyConnected layer with a recurrent data dependence along the sequence dimension.

Shuffle

Shuffle operations that are equivalent to identity operations on the underlying data are omitted if the input tensor is only used in the shuffle layer and the input and output tensors of this layer are not input and output tensors of the network. TensorRT does not execute additional kernels or memory copies for such operations.

TopK

To get the maximum performance out of a TopK layer, use small values of `K` reducing the last dimension of data to allow optimal sequential memory accesses. Reductions along multiple dimensions at once can be simulated by using a Shuffle layer to reshape the data, then reinterpreting the index values appropriately.

For more information about layers, refer to the NVIDIA TensorRT Operator's Reference.

13.6. Optimizing for Tensor Cores

Tensor Core is a key technology to deliver high-performance inference on NVIDIA GPUs. In TensorRT, Tensor Core operations are supported by all compute-intensive layers - MatrixMultiply, FullyConnected, Convolution, and Deconvolution.

Tensor Core layers tend to achieve better performance if the I/O tensor dimensions are aligned to a certain minimum granularity:

- In Convolution and Deconvolution layers the alignment requirement is on I/O channel dimension.
- In MatrixMultiply and FullyConnected layers the alignment requirement is on matrix dimensions `K` and `N` in a MatrixMultiply that is `M x K` times `K x N`.

The following table captures the suggested tensor dimension alignment for better Tensor Core performance.

Table 3. 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

When using Tensor Core implementations in cases where these requirements are not met, TensorRT implicitly pads the tensors to the nearest multiple of alignment rounding up the dimensions in the model definition instead to allow for extra capacity in the model without increasing computation or memory traffic.

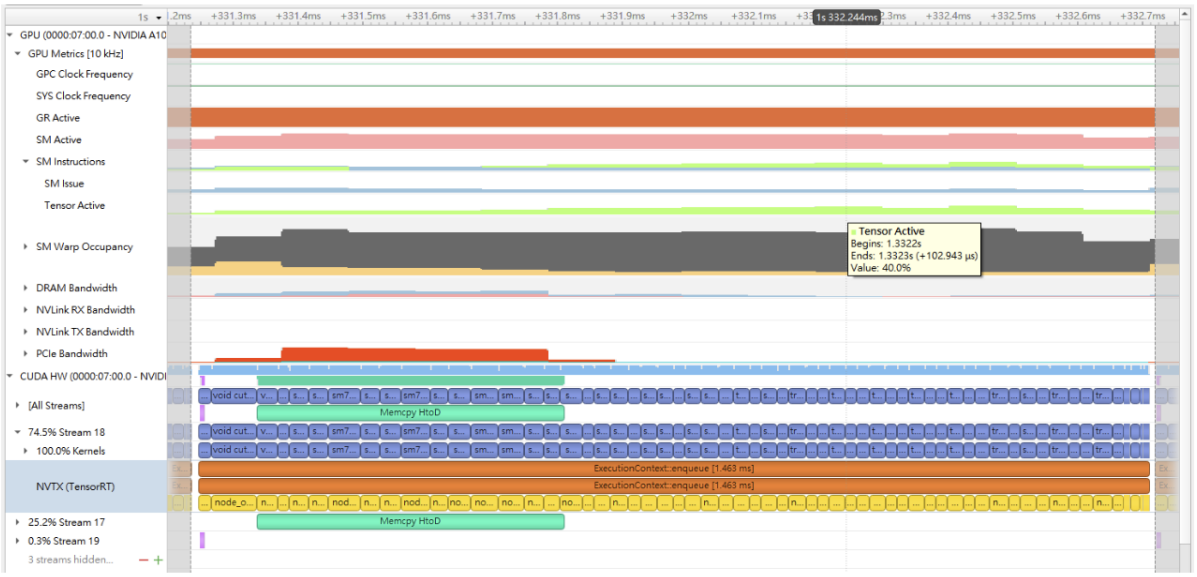
TensorRT always uses the fastest implementation for a layer, and thus in some cases may not use a Tensor Core implementation even if available.

To check if Tensor Core is used for a layer, run Nsight Systems with the `--gpu-metrics-device all` flag while profiling the TensorRT application. The Tensor Core usage rate can be found in the profiling result in the Nsight Systems user interface under the SM instructions/Tensor Active row. Refer to the CUDA Profiling Tools for more information about how to use Nsight Systems to profile TensorRT applications.

Note that it is not practical to expect a CUDA kernel to reach 100% Tensor Core usage since there are other overheads such as DRAM reads/writes, instruction stalls, other computation units, and so on. In general, the more computation-intensive an operation is, the higher the Tensor Core usage rate the CUDA kernel can achieve.

The following image is an example of Nsight Systems profiling.

Figure 23. Tensor Core Activities on an A100 GPU Running ResNet-50 with FP16 Enabled



13.7. Optimizing Plugins

TensorRT provides a mechanism for registering custom plugins that perform layer operations. After a plugin creator is registered, you can look up the registry to find the creator and add the corresponding plugin object to the network during serialization/deserialization.

All TensorRT plugins are automatically registered once the plugin library is loaded. For more information about custom plugins, refer to Extending TensorRT with Custom Layers.

The performance of plugins depends on the CUDA code performing the plugin operation. Standard CUDA best practices apply. When developing plugins, it can be helpful to start with simple standalone CUDA applications that perform the plugin operation and verify correctness. The plugin program can then be extended with performance measurements, more unit testing, and alternate implementations. After the code is working and optimized, it can be integrated as a plugin into TensorRT.

To get the best performance possible, it is important to support as many formats as possible in the plugin. This removes the need for internal reformat operations during the execution of the network. Refer to the Extending TensorRT with Custom Layers section for examples.

13.8. Optimizing Python Performance

When using the Python API, most of the same performance considerations apply. When building engines, the builder optimization phase will normally be the performance

bottleneck; not API calls to construct the network. Inference time should be nearly identical between the Python API and C++ API.

Setting up the input buffers in the Python API involves using `pycuda` or another CUDA Python library, like `cupy`, to transfer the data from the host to device memory. The details of how this works will depend on where the host data is coming from. Internally, `pycuda` supports the Python Buffer Protocol which allows efficient access to memory regions. This means that if the input data is available in a suitable format in `numpy` arrays or another type that also has support for the buffer protocol, this allows efficient access and transfer to the GPU. For even better performance, ensure that you allocate a page-locked buffer using `pycuda` and write your final preprocessed input there.

For more information about using the Python API, refer to The Python API.

13.9. Improving Model Accuracy

TensorRT can execute a layer in FP32, FP16, BF16, FP8 or INT8 precision depending on the builder configuration. By default, TensorRT chooses to run a layer in a precision that results in optimal performance. Sometimes this can result in poor accuracy. Generally, running a layer in higher precision helps improve accuracy with some performance hit.

There are several steps that we can take to improve model accuracy:

1. Validate layer outputs:
   a). Use Polygraphy to dump layer outputs and verify that there are no NaNs or Infs. The `--validate` option can check for NaNs and Infs. Also, we can compare layer outputs with golden values from, for example, ONNX runtime.
   b). For FP16 and BF16, it is possible that a model might require retraining to ensure that intermediate layer output can be represented in FP16/BF16 precision without overflow/underflow.
   c). For INT8, consider recalibrating with a more representative calibration data set. If your model comes from PyTorch, we also provide NVIDIA's Quantization Toolkit for PyTorch for QAT in the framework besides PTQ in TensorRT. You can try both approaches and choose the one with more accuracy.
2. Manipulate layer precision:
   a). Sometimes running a layer in certain precision results in incorrect output. This can be due to inherent layer constraints (for example, LayerNorm output should not be INT8), model constraints (output gets diverged resulting in poor accuracy), or report a TensorRT bug.
   b). You can control layer execution precision and output precision.
   c). An experimental debug precision tool can help automatically find layers to run in high precision.
3. Use an Algorithm Selection and Reproducible Builds to disable flaky tactics:
   a). When accuracy changes between build+run to build+run, it might be due to a selection of a bad tactic for a layer.

b). Use an algorithm selector to dump tactics from both good and bad runs. Configure the algorithm selector to allow only a subset of tactics (that is, just allow tactics from a good run, and so on).

c). You can use Polygraphy to automate this process.

Accuracy from run-to-run variation should not change; once the engine is built for a specific GPU, it should result in bit accurate outputs in multiple runs. If not, file a TensorRT bug.

13.10. Optimizing Builder Performance

For each layer, the TensorRT builder profiles all the available tactics to search for the fastest inference engine plan. The builder time can be long if the model has a large number of layers or complicated topology. The following sections provide options to reduce builder time.

13.10.1. Timing Cache

To reduce builder time, TensorRT creates a layer timing cache to keep the layer-profiling information. The information it contains is specific to the targeted device, CUDA, TensorRT versions, and `BuilderConfig` parameters that can change the layer implementation such as `BuilderFlag::kTF32` or `BuilderFlag::kREFIT`.

If there are other layers with the same IO tensor configuration and layer parameters, the TensorRT builder skips profiling and reuses the cached result for the repeated layers. If a timing query misses in the cache, the builder times the layer and updates the cache.

The timing cache can be serialized and deserialized. You can load a serialized cache from a buffer using `IBuilderConfig::createTimingCache`:.

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

Setting the buffer size to `0` creates a new empty timing cache.

You then attach the cache to a builder configuration before building.

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

During the build, the timing cache can be augmented with more information as a result of cache misses. After the build, it can be serialized for use with another builder.

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

If there is no timing cache attached to a builder, the builder creates its own temporary local cache and destroys it when it is done.

The compilation cache is part of the timing cache, which caches JIT-compiled code and will be serialized as part of the timing cache by default. It can be disabled by setting the `BuildFlag`.

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

The cache is incompatible with algorithm selection (refer to the Algorithm Selection and Reproducible Builds section). It can be disabled by setting the `BuilderFlag`.

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

Note: The timing cache supports the most frequently used layer types: Convolution, Deconvolution, Pooling, SoftMax, MatrixMultiply, ElementWise, Shuffle, and tensor memory layout conversion. More layer types will be added in future releases.

13.10.2. Tactic Selection Heuristic

TensorRT allows heuristic-based tactic selection to minimize the builder time in the layer profiling stage. The builder predicts the tactic timing for the given problem size and prunes the tactics that are not likely to be fast prior to the layer profiling stage. In cases where the prediction is wrong, the engine will not be as performant as when built with a profiling-based builder. This feature can be enabled by setting the `BuilderFlag`.

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

Note: The tactic selection heuristic feature is only supported by the NVIDIA Ampere architecture and newer GPUs.

13.11. Builder Optimization Level

Set the optimization level in builder config to adjust how long TensorRT should spend searching for tactics with potentially better performance. By default, the optimization level is 3. Setting it to a smaller value results in much faster engine building time, but the performance of the engine may be worse. On the other hand, setting it to a larger value will increase the engine building time, but the resulting engine may perform better if TensorRT is able to find better tactics.

For example, to set the optimization level to 0 (the fastest):

C++

```
config->setOptimizationLevel(0);
```

Python

```
config.optimization_level = 0
```

Chapter 14. Troubleshooting

The following sections help answer the most commonly asked questions regarding typical use cases.

14.1. FAQs

This section is to help troubleshoot the problem and answer our most asked questions.

Q: How do I create an engine that is optimized for several different batch sizes?

A: While TensorRT allows an engine optimized for a given batch size to run at any smaller size, the performance for those smaller sizes cannot be as well optimized. To optimize for multiple different batch sizes, create optimization profiles at the dimensions that are assigned to `OptProfilerSelector::kOPT`.

Q: Are calibration tables portable across TensorRT versions?

A: No. Internal implementations are continually optimized and can change between versions. For this reason, calibration tables are not guaranteed to be binary compatible with different versions of TensorRT. Applications must build new INT8 calibration tables when using a new version of TensorRT.

Q: Are engines portable across TensorRT versions?

A: By default, no. Refer to Version Compatibility for how to configure engines for forward compatibility.

Q: How do I choose the optimal workspace size?

A: Some TensorRT algorithms require additional workspace on the GPU. The method `IBuilderConfig::setMemoryPoolLimit()` controls the maximum amount of workspace that can be allocated and prevents algorithms that require more workspace from being considered by the builder. At runtime, the space is allocated automatically when creating an `IExecutionContext`. The amount allocated is no more than is required, even if the amount set in `IBuilderConfig::setMemoryPoolLimit()` is much higher. Applications

should therefore allow the TensorRT builder as much workspace as they can afford; at runtime, TensorRT allocates no more than this and typically less.

Q: How do I use TensorRT on multiple GPUs?

A: Each `ICudaEngine` object is bound to a specific GPU when it is instantiated, either by the builder or on deserialization. To select the GPU, use `cudaSetDevice()` before calling the builder or deserializing the engine. Each `IExecutionContext` is bound to the same GPU as the engine from which it was created. When calling `execute()` or `enqueue()`, ensure that the thread is associated with the correct device by calling `cudaSetDevice()` if necessary.

Q: How do I get the version of TensorRT from the library file?

A: There is a symbol in the symbol table named `tensorrt_version_#_#_#_#` which contains the TensorRT version number. One possible way to read this symbol on Linux is to use the `nm` command like in the following example:

```
$ nm -D libnvinfer.so.* | grep tensorrt_version
00000000abcd1234 B tensorrt_version_#_#_#_#
```

Q: What can I do if my network is producing the wrong answer?

A: There are several reasons why your network can be generating incorrect answers. Here are some troubleshooting approaches that can help diagnose the problem:

- Turn on `VERBOSE` level messages from the log stream and check what TensorRT is reporting.
- Check that your input preprocessing is generating exactly the input format required by the network.
- If you are using reduced precision, run the network in FP32. If it produces the correct result, it is possible that lower precision has an insufficient dynamic range for the network.
- Try marking intermediate tensors in the network as outputs, and verify if they match what you are expecting.

> Note: Marking tensors as outputs can inhibit optimizations, and therefore, can change the results.

You can use NVIDIA Polygraphy to assist you with debugging and diagnosis.

Q: How do I implement batch normalization in TensorRT?

A: Batch normalization can be implemented using a sequence of `IElementWiseLayer` in TensorRT. More specifically:

```
adjustedScale = scale / sqrt(variance + epsilon)
batchNorm = (input + bias - (adjustedScale * mean)) * adjustedScale
```

Q: Why does my network run slower when using DLA compared to without DLA?

A: DLA was designed to maximize energy efficiency. Depending on the features supported by DLA and the features supported by the GPU, either implementation can be more performant. Which implementation to use depends on your latency or throughput requirements and your power budget. Since all DLA engines are independent of the GPU and each other, you could also use both implementations at the same time to further increase the throughput of your network.

Q: Is INT4 quantization or INT16 quantization supported by TensorRT?

A: TensorRT supports INT4 quantization for GEMM weight-only quantization. TensorRT does not support INT16 quantization.

Q: When will TensorRT support layer XYZ required by my network in the UFF parser?

A: UFF is deprecated. We recommend users switch their workflows to ONNX. The TensorRT ONNX parser is an open source project.

Q: Can I use multiple TensorRT builders to compile on different targets?

A: TensorRT assumes that all resources for the device it is building on are available for optimization purposes. Concurrent use of multiple TensorRT builders (for example, multiple `trtexec` instances) to compile on different targets (DLA0, DLA1 and GPU) can oversubscribe system resources causing undefined behavior (meaning, inefficient plans, builder failure, or system instability).

It is recommended to use `trtexec` with the `--saveEngine` argument to compile for different targets (DLA and GPU) separately and save their plan files. Such plan files can then be reused for loading (using `trtexec` with the `--loadEngine` argument) and submitting multiple inference jobs on the respective targets (DLA0, DLA1, GPU). This two-step process alleviates over-subscription of system resources during the build phase while also allowing execution of the plan file to proceed without interference by the builder.

Q: Which layers are accelerated by Tensor Cores?

A: Most math-bound operations will be accelerated with tensor cores - convolution, deconvolution, fully connected, and matrix multiply. In some cases, particularly for small channel counts or small group sizes, another implementation may be faster and be selected instead of a tensor core implementation.

Q: Why are reformatting layers observed although there is no warning message `no implementation obeys reformatting-free rules ...`?

A: Reformat-free network I/O does not mean that there are no reformatting layers inserted in the entire network. Only that the input and output network tensors have a possibility not to require reformatting layers. In other words, reformatting layers can be inserted by TensorRT for internal tensors to improve performance.

14.2. Understanding Error Messages

If an error is encountered during execution, TensorRT reports an error message that is intended to help in debugging the problem. Some common error messages that can be encountered by developers are discussed in the following sections.

ONNX Parser Error Messages

The following table captures the common ONNX parser error messages. For more information on specific ONNX node support, refer to the operators support document.

Error Message	Description
`<X> must be an initializer!` `!inputs.at(X).is_weights()`	These error messages signify that an ONNX node input tensor is expected to be an initializer in TensorRT. A possible fix is to run constant folding on the model using TensorRT's Polygraphy tool: `polygraphy surgeon sanitize model.onnx --fold-constants --output model_folded.onnx`
`getPluginCreator() could not find Plugin <operator name> version 1`	This is an error stating that the ONNX parser does not have an import function defined for a particular operator, and did not find a corresponding plugin in the loaded registry for the operator.

TensorRT Core Library Error Messages

The following table captures the common TensorRT core library error messages.

	Error Message	Description
Installation Errors	`Cuda initialization failure with error <code>. Please check cuda installation: http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.`	This error message can occur if the CUDA or NVIDIA driver installation is corrupt. Refer to the URL for instructions

	Error Message	Description
		on installing CUDA and the NVIDIA driver on your OS.
Builder Errors	`Internal error: could not find any implementation for node <name>. Try increasing the workspace size with IBuilderConfig::setMemoryPoolLimit()`	This error message occurs because there is no layer implementation for the given node in the network that can operate with the given workspace size. This usually occurs because the workspace size is insufficient but could also indicate a bug. If increasing the workspace size as suggested does not help, report a bug (refer to Reporting TensorRT Issues).
	`<layer-name>: (kernel|bias) weights has non-zero count but null values` `<layer-name>: (kernel|bias) weights has zero count but non-null values`	This error message occurs when there is a mismatch between the values and count fields in a Weights data structure passed to the builder. If the count is `0`, then the values field must contain a null pointer; otherwise, the count must be non-zero, and values must contain a non-null pointer.
	`Builder was created on device different from current device.`	This error message can show up if you: 1. Created an `IBuilder` targeting one GPU, then 2. Called `cudaSetDevice()` to target a different GPU, then 3. Attempted to use the `IBuilder` to create an engine. Ensure you only use the `IBuilder` when targeting the GPU that was used to create the `IBuilder`.

	Error Message	Description
	You can encounter error messages indicating that the tensor dimensions do not match the semantics of the given layer. Carefully read the documentation on `NvInfer.h` on the usage of each layer and the expected dimensions of the tensor inputs and outputs to the layer.	
INT8 Calibration Errors	`Tensor <X> is uniformly zero.`	This warning occurs and should be treated as an error when data distribution for a tensor is uniformly zero. In a network, the output tensor distribution can be uniformly zero under the following scenarios: 1. Constant tensor with all zero values; not an error. 2. Activation (ReLU) output with all negative inputs: not an error. 3. Data distribution is forced to all zero due to computation error in the previous layer; emit a warning here.[1] 4. User does not provide any calibration images; emit a warning here.[2]
	`Could not find scales for tensor <X>.`	This error message indicates that a calibration failure occurred with no scaling factors detected. This could be due to no INT8 calibrator or insufficient custom scales for network layers.
Engine Compatibility Errors	`The engine plan file is not compatible with this version of TensorRT, expecting (format|library) version <X> got <Y>, please rebuild.`	This error message can occur if you are running TensorRT using an engine PLAN file that is incompatible with the

[1] It is recommended to evaluate the calibration input or validate the previous layer outputs.
[2] It is recommended to evaluate the calibration input or validate the previous layer outputs.

	Error Message	Description
		current version of TensorRT. Ensure you use the same version of TensorRT when generating the engine and running it.
	`The engine plan file is generated on an incompatible device, expecting compute <X> got compute <Y>, please rebuild.`	This error message can occur if you build an engine on a device of a different compute capability than the device that is used to run the engine.
	`Using an engine plan file across different models of devices is not recommended and is likely to affect performance or even cause errors.`	This warning message can occur if you build an engine on a device with the same compute capability but is not identical to the device that is used to run the engine. As indicated by the warning, it is highly recommended to use a device of the same model when generating the engine and deploying it to avoid compatibility issues.

Out Of Memory Errors	`GPU memory allocation failed during initialization of (tensor|layer): <name> GPU memory`
	`Allocation failed during deserialization of weights.`
	`GPU does not meet the minimum memory requirements to run this engine …`
		These error messages can occur if there is insufficient GPU memory available to instantiate a given TensorRT engine. Verify that the GPU has sufficient available memory to contain the required layer weights and activation tensors.

FP16 Errors	`Network needs native FP16 and platform does not have native FP16`	This error message can occur if you attempt to deserialize an engine that uses FP16 arithmetic on a GPU that does not support FP16 arithmetic. You either must rebuild the engine without FP16 precision inference or upgrade your GPU

	Error Message	Description
		to a model that supports FP16 precision inference.
Plugin Errors	`Custom layer <name> returned non-zero initialization`	This error message can occur if the `initialize()` method of a given plugin layer returns a non-zero value. Refer to the implementation of that layer to debug this error further. For more information, refer to the NVIDIA TensorRT Operator's Reference.

14.3. Code Analysis Tools

14.3.1. Compiler Sanitizers

Google sanitizers are a set of code analysis tools.

14.3.1.1. Issues with `dlopen` and Address Sanitizer

There is a known issue with sanitizers, documented here. When using `dlopen` on TensorRT under a sanitizer, there will be reports of memory leaks unless one of two solutions is adopted:

1. Do not call `dlclose` when running under the sanitizers.
2. Pass the flag `RTLD_NODELETE` to `dlopen` when running under sanitizers.

14.3.1.2. Issues with `dlopen` and Thread Sanitizer

The thread sanitizer can list errors when using `dlopen` from multiple threads. In order to suppress this warning, create a file called `tsan.supp` and add the following to the file:

```
race::dlopen
```

When running applications under thread sanitizer, set the environment variable using:

```
export TSAN_OPTIONS="suppressions=tsan.supp"
```

14.3.1.3. Issues with CUDA and Address Sanitizer

The address sanitizer has a known issue with CUDA applications documented here. In order to successfully run CUDA libraries such as TensorRT under the address sanitizer, add the option `protect_shadow_gap=0` to the `ASAN_OPTIONS` environment variable.

On CUDA 11.4, there is a known bug that can trigger mismatched allocation-and-free errors in the address sanitizer. Add `alloc_dealloc_mismatch=0` to `ASAN_OPTIONS` to disable these errors.

14.3.1.4. Issues with Undefined Behavior Sanitizer

UndefinedBehaviorSanitizer (UBSan) reports false positives with the `-fvisibility=hidden` option as documented here. Add the `-fno-sanitize=vptr` option to avoid UBSan reporting such false positives.

14.3.2. Valgrind

Valgrind is a framework for dynamic analysis tools that can be used to automatically detect memory management and threading bugs in applications.

Some versions of valgrind and glibc are affected by a bug, which causes false memory leaks to be reported when `dlopen` is used, which can generate spurious errors when running a TensorRT application under valgrind's `memcheck` tool. To work around this, add the following to a valgrind suppressions file as documented here:

```
{
   Memory leak errors with dlopen
   Memcheck:Leak
   match-leak-kinds: definite
   ...
   fun:*dlopen*
   ...
}
```

On CUDA 11.4, there is a known bug that can trigger mismatched allocation-and-free errors in valgrind. Add the option `--show-mismatched-frees=no` to the valgrind command line to suppress these errors.

14.3.3. Compute Sanitizer

When running a TensorRT application under compute-sanitizer, `cuGetProcAddress` can fail with error code 500 due to missing functions. This error can be ignored or suppressed with `--report-api-errors no` option. This is due to CUDA backward compatibility checking if a function is usable on the CUDA toolkit/driver combination. The functions are introduced in a later version of CUDA but are not available on the current platform.

14.4. Understanding Formats Printed in Logs

In logs from TensorRT, formats are printed as a type followed by stride and vectorization information. For example:

```
Half(60,1:8,12,3)
```

The `Half` indicates that the element type is `DataType::kHALF`, that is, 16-bit floating point. The `:8` indicates the format packs eight elements per vector, and that vectorization is along the second axis. The rest of the numbers are strides in units of vectors. For this tensor, the mapping of a coordinate `(n,c,h,w)` to an address is:

```
((half*)base_address) + (60*n + 1*floor(c/8) + 12*h + 3*w) * 8 + (c mod 8)
```

The `1:` is common to NHWC formats. For example, here is another example for an NCHW format:

```
Int8(105,15:4,3,1)
```

The INT8 indicates that the element type is `DataType::kINT8`, and the `:4` indicates a vector size of 4. For this tensor, the mapping of a coordinate `(n,c,h,w)` to an address is:

```
(int8_t*)base_address + (105*n + 15*floor(c/4) + 3*h + w) * 4 + (c mod 4)
```

Scalar formats have a vector size of 1. For brevity, printing omits the *:1*.

In general, the coordinates to address mappings have the following form:

```
(type*)base_address + (vec_coordinate · strides) * vec_size + vec_mod
```

Where:

- the dot denotes an inner product
- strides are the printed strides, that is, strides in units of vectors
- `vec_size` is the number of elements per vectors
- `vec_coordinate` is the original coordinate with the coordinate along the vectorized axis divided by `vec_size`
- `vec_mod` is the original coordinate along the vectorized axis modulo `vec_size`

14.5. Reporting TensorRT Issues

If you encounter issues when using TensorRT, first confirm that you have followed the instructions in this Developer Guide. Also, check the FAQs and the Understanding Error Messages sections to look for similar failing patterns. For example, many engine building failures can be solved by sanitizing and constant-folding the ONNX model using Polygraphy with the following command:

```
polygraphy surgeon sanitize model.onnx --fold-constants --output
        model_folded.onnx
```

In addition, it is highly recommended that you first try our latest TensorRT release before filing an issue if you have not done so, since your issue may have been fixed in the latest release.

14.5.1. Channels for TensorRT Issue Reporting

If none of the FAQs or the Understanding Error Messages work, there are two main channels where you can report the issue: NVIDIA Developer Forum or TensorRT GitHub Issue page. These channels are constantly monitored to provide feedback to the issues you encountered.

Here are the steps to report an issue on the NVIDIA Developer Forum:

1. Register for the NVIDIA Developer website.
2. Log in to the developer site.
3. Click on your name in the upper right corner.
4. Click My account > My Bugs and select Submit a New Bug.

5. Fill out the bug reporting page. Be descriptive and if possible, provide the steps to reproduce the problem.
6. Click Submit a bug.

When reporting an issue, provide setup details and include the following information:

- Environment information:
  - OS or Linux distro and version
  - GPU type
  - NVIDIA driver version
  - CUDA version
  - cuDNN version
  - Python version (if Python is used).
  - TensorFlow, PyTorch, and ONNX version (if any of them is used).
  - TensorRT version
  - NGC TensorRT container version (if TensorRT container is used).
  - Jetson (if used), include OS and hardware versions
- Thorough description of the issue.
- Steps to reproduce the issue:
  - ONNX file (if ONNX is used).
  - Minimal commands or scripts to trigger the issue
  - Verbose logs by enabling `kVERBOSE` in `ILogger`

Depending on the type of the issue, providing more information listed below can expedite the response and debugging process.

14.5.2. Reporting a Functional Issue

When reporting functional issues, such as linker errors, segmentation faults, engine building failures, inference failures, and so on, provide the scripts and the commands to reproduce the issue as well as the detailed description of the environment. Having more details helps us in debugging the functional issue faster.

Since the TensorRT engine is specific to a specific TensorRT version and a specific GPU type, do not build the engine in one environment and use it to run it in another environment with different GPUs or dependency software stack, such as TensorRT version, CUDA version, cuDNN version, and so on. Also, ensure that the application is linked to the correct TensorRT and cuDNN shared object files by checking the environment variable `LD_LIBRARY_PATH` (or `%PATH%` on Windows).

14.5.3. Reporting an Accuracy Issue

When reporting an accuracy issue, provide the scripts and the commands used to calculate the accuracy metrics. Describe what the expected accuracy level is and, if

possible, share the steps to get the expected results using other frameworks like ONNX-Runtime.

The Polygraphy tool can be used to debug the accuracy issue and produce a minimal failing case. Refer to the Debugging TensorRT Accuracy Issues documentation for the instructions. Having a Polygraphy command that shows the accuracy issue or having the minimal failing case expedites the time it takes for us to debug your accuracy issue.

Note that it is not practical to expect bitwise identical results between TensorRT and other frameworks like PyTorch, TensorFlow, or ONNX-Runtime even in FP32 precision since the order of the computations on the floating-point numbers can result in slight differences in output values. In practice, small numeric differences should not significantly affect the accuracy metric of the application, such as the mAP score for object-detection networks or the BLEU score for translation networks. If you *do* see a significant drop in the accuracy metric between using TensorRT and using other frameworks such as PyTorch, TensorFlow, or ONNX-Runtime, it may be a genuine TensorRT bug.

If you are seeing NaNs or infinite values in TensorRT engine output when FP16/BF16 precision is enabled, it is possible that intermediate layer outputs in the network overflow in FP16/BF16. Some approaches to help mitigate this include:

- Ensuring that network weights and inputs are restricted to a reasonably narrow range (such as [-1, 1] instead of [-100, 100]). This may require making changes to the network and retraining.
  - Consider pre-processing input by scaling or clipping it to the restricted range before passing it to the network for inference.
- Overriding precision for individual layers vulnerable to overflows (for example, Reduce and Element-Wise Power ops) to FP32.

Polygraphy can help you diagnose common problems with using reduced precision. Refer to Polygraphy's Working with Reduced Precision how-to guide for more details.

Refer to the Improving Model Accuracy section for some possible solutions to accuracy issues, and the Working with Quantized Types section for instructions about using INT8/FP8 precision.

14.5.4. Reporting a Performance Issue

If you are reporting a performance issue, share the full `trtexec` logs using this command:

```
trtexec --onnx=<onnx_file> <precision_and_shape_flags> --verbose --
profilingVerbosity=detailed --dumpLayerInfo --dumpProfile --separateProfileRun --useCudaGraph
 --noDataTransfers --useSpinWait --duration=60
```

The verbose logs help us to identify the performance issue. If possible, also share the Nsight Systems profiling files using these commands:

```
trtexec --onnx=<onnx_file> <precision_and_shape_flags> --verbose --
profilingVerbosity=detailed --dumpLayerInfo --saveEngine=<engine_path>
nsys profile --cuda-graph-trace=node -o <output_profile> trtexec --loadEngine=<engine_path>
 <precision_and_shape_flags> --useCudaGraph --noDataTransfers --useSpinWait --warmUp=0 --
duration=0 --iterations=20
```

Refer to the trtexec section for more instructions about how to use the `trtexec` tool and the meaning of these flags.

If you do not use `trtexec` to measure performance, provide the scripts and the commands that you use to measure the performance. If possible, compare the performance measurement from your script with that from the `trtexec` tool. If the two numbers differ, there may be some issues about the performance measurement methodology in your scripts.

Refer to the Hardware/Software Environment for Performance Measurements section for some environment factors that may affect the performance.

Appendix A. Appendix

A.1. Data Format Descriptions

TensorRT supports different data formats. There are two aspects to consider: data type and layout.

Data Type Format

The data type is the representation of each individual value. Its size determines the range of values and the precision of the representation, which are:

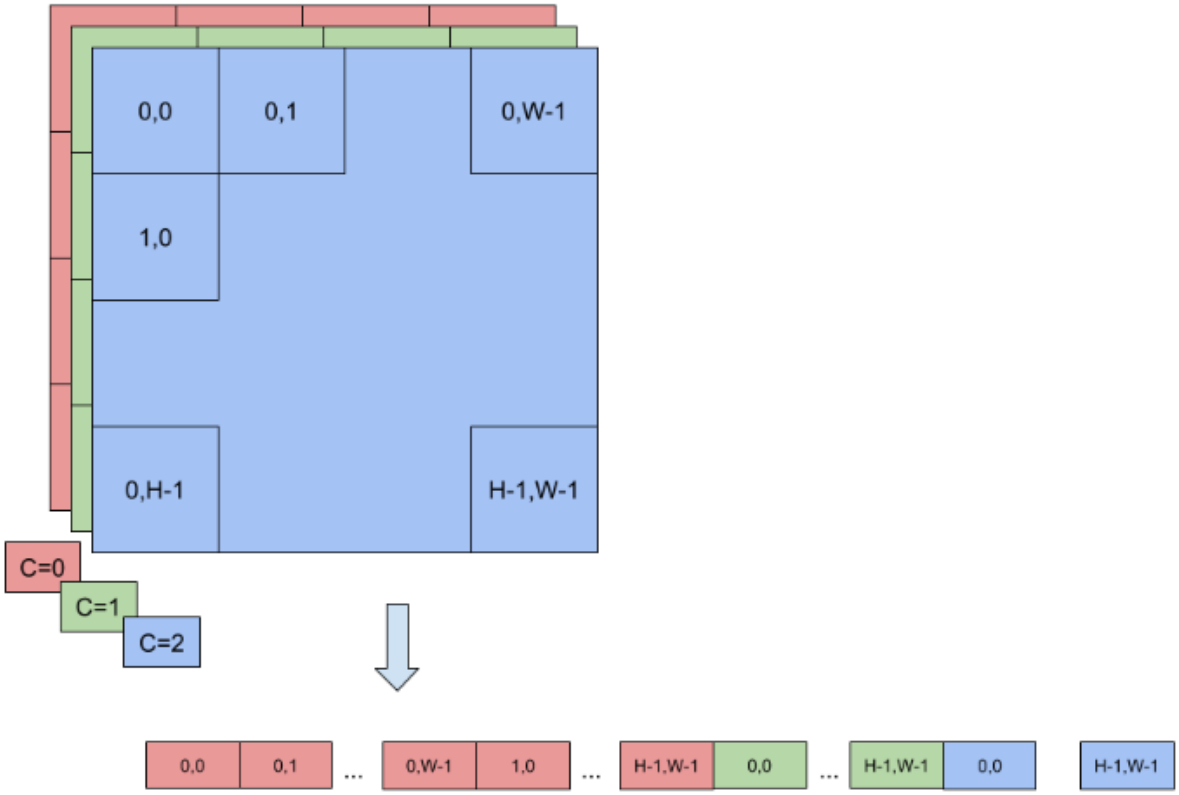
- FP32 (32-bit floating point, or single precision)
- FP16 (16-bit floating point, or half precision)
- BF16 (1-bit sign, 8-bit exponent, 7-bit mantissa)
- FP8 (1-bit sign, 4-bit exponent, 3-bit mantissa)
- INT64 (64-bit integer)
- INT32 (32-bit integer)
- INT8 (8-bit integer)
- UINT8 (unsigned 8-bit integer)
- INT4 (4-bit integer)

Layout Format

The layout format determines the ordering in which values are stored. Typically, batch dimensions are the leftmost dimensions, and the other dimensions refer to aspects of each data item, such as `C` is channel, `H` is height, and `W` is width, in images. Ignoring batch sizes, which are always preceding these, `C`, `H`, and `W` are typically sorted as `CHW` (refer to Figure 24) or `HWC` (refer to Figure 25).

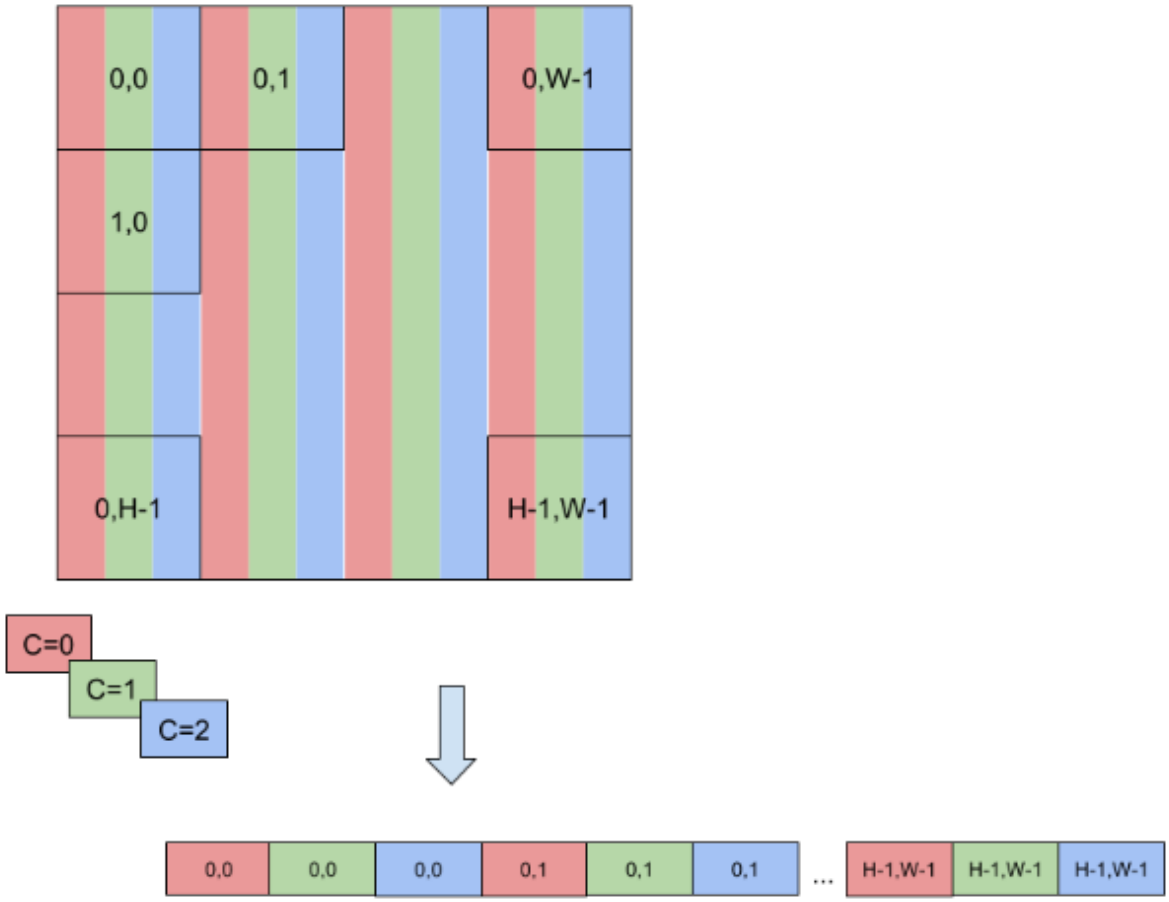
The following image is divided into `HxW` matrices, one per channel, and the matrices are stored in sequence; all the values of a channel are stored contiguously.

Figure 24. Layout Format for `CHW`



The image is stored as a single `HxW` matrix, whose value is actually C-tuple, with a value per channel; all the values of a point (pixel) are stored contiguously.

Figure 25. Layout format for `HWC`

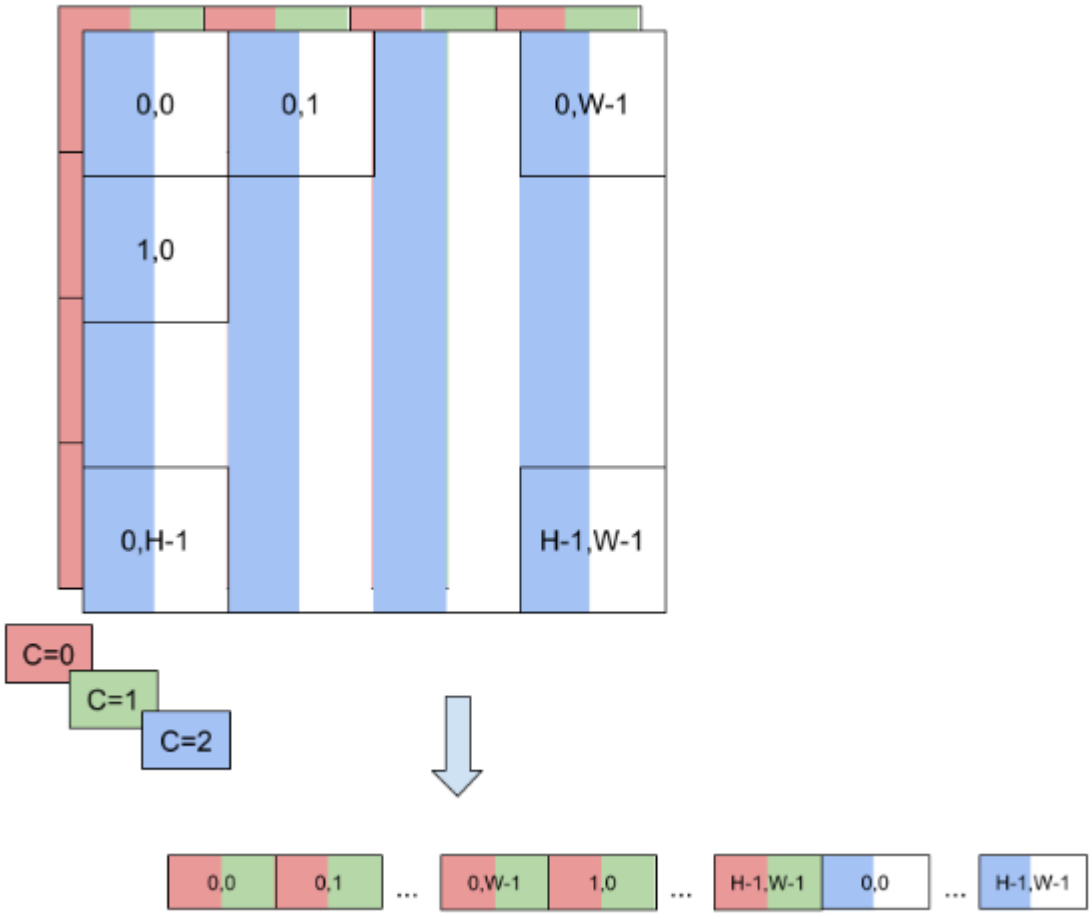


To enable faster computations, more formats are defined to pack together channel values and use reduced precision. For this reason, TensorRT also supports formats like `NC`, `2HW2` and `NHWC8`.

In `NC`, `2HW2 (TensorFormat::kCHW2)`, pairs of channel values are packed together in each `HxW` matrix (with an empty value in the case of an odd number of channels). The result is a format in which the values of `#C/2# HxW` matrices are pairs of values of two consecutive channels (refer to Figure 26); notice that this ordering interleaves dimension as values of channels that have stride `1` if they are in the same pair and stride `2xHxW` otherwise.

A pair of channel values is packed together in each `HxW` matrix. The result is a format in which the values of `#C/2# HxW` matrices are pairs of values of two consecutive channels.

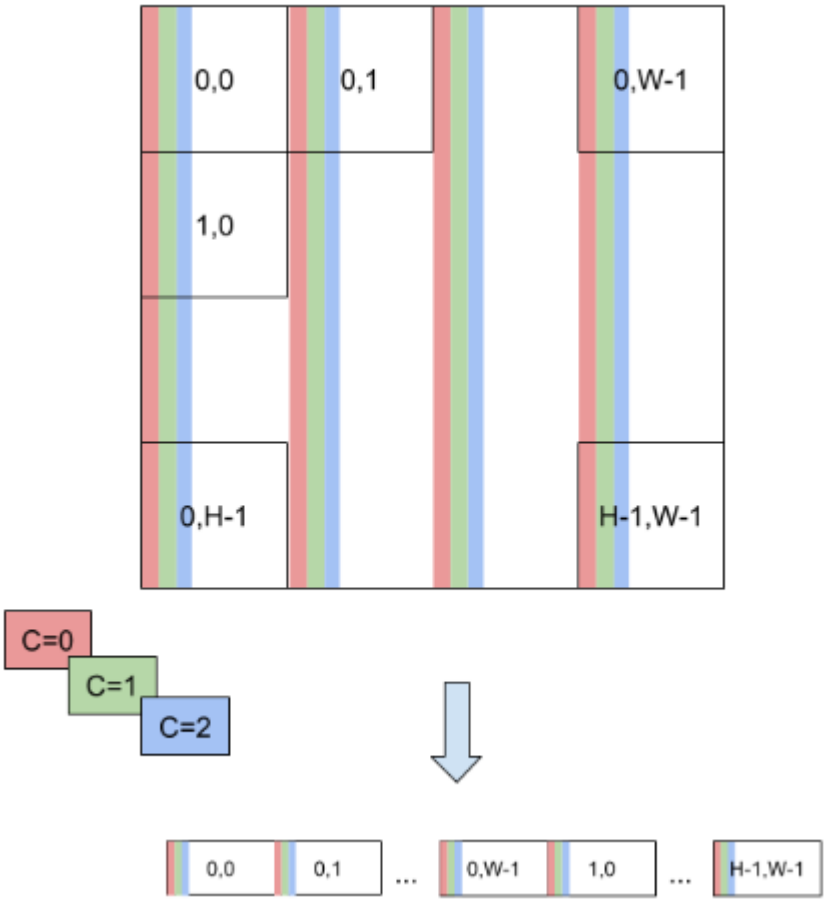
Figure 26. Values of `#C/2# HxW` Matrices are Pairs of Values of Two Consecutive Channels



In `NHWC8 (TensorFormat::kHWC8)`, the entries of an `HxW` matrix include the values of all the channels (refer to Figure 27). In addition, these values are packed together in `#C/8#` 8-tuples, and `C` is rounded up to the nearest multiple of 8.

In this `NHWC8` format, the entries of an `HxW` matrix include the values of all the channels.

Figure 27. In `NHWC8` Format, the Entries of an `HxW` Matrix Include the Values of all the Channels



Other `TensorFormat` follow similar rules to `TensorFormat::kCHW2` and `TensorFormat::kHWC8` mentioned previously.

A.2. Command-Line Programs

A.2.1. `trtexec`

Included in the `samples` directory is a command-line wrapper tool called `trtexec`. `trtexec` is a tool to quickly utilize TensorRT without having to develop your own application. The `trtexec` tool has three main purposes:

- It is useful for *benchmarking networks* on random or user-provided input data.
- It is useful for *generating serialized engines* from models.
- It is useful for *generating serialized timing cache* from the builder.

A.2.1.1. Benchmarking Network

If you have a model saved as an ONNX file, you can use the `trtexec` tool to test the performance of running inference on your network using TensorRT. The `trtexec` tool has many options for specifying inputs and outputs, iterations for performance timing, precision allowed, and other options.

To maximize GPU utilization, `trtexec` enqueues the inferences one batch ahead of time. In other words, it does the following:

```
enqueue batch 0 -> enqueue batch 1 -> wait until batch 0 is done -> enqueue batch 2 -> wait
 until batch 1 is done -> enqueue batch 3 -> wait until batch 2 is done -> enqueue batch 4 -
> ...
```

If cross-inference multi-stream (`--infStreams=N` flag) is used, then `trtexec` follows this pattern on each stream separately.

The `trtexec` tool prints the following performance metrics. The following figure shows an example Nsight System profile of a `trtexec` run with markers showing what each performance metric means.

Throughput
The observed throughput is computed by dividing the number of inferences by the Total Host Walltime. If this is significantly lower than the reciprocal of GPU Compute Time, the GPU may be underutilized because of host-side overheads or data transfers. Using CUDA graphs (with `--useCudaGraph`) or disabling H2D/D2H transfers (with `--noDataTransfer`) may improve GPU utilization. The output log provides guidance on which flag to use when `trtexec` detects that the GPU is underutilized.

Host Latency
The summation of H2D Latency, GPU Compute Time, and D2H Latency. This is the latency to infer a single inference.

Enqueue Time
The host latency to enqueue an inference, including calling H2D/D2H CUDA APIs, running host-side heuristics, and launching CUDA kernels. If this is longer than GPU Compute Time, the GPU may be underutilized and the throughput may be dominated by host-side overhead. Using CUDA graphs (with `--useCudaGraph`) may reduce enqueue time.

H2D Latency
The latency for host-to-device data transfers for input tensors of a single inference. Add `--noDataTransfer` to disable H2D/D2H data transfers.

D2H Latency
The latency for device-to-host data transfers for output tensors of a single inference. Add `--noDataTransfer` to disable H2D/D2H data transfers.

GPU Compute Time
The GPU latency to execute the CUDA kernels for an inference.

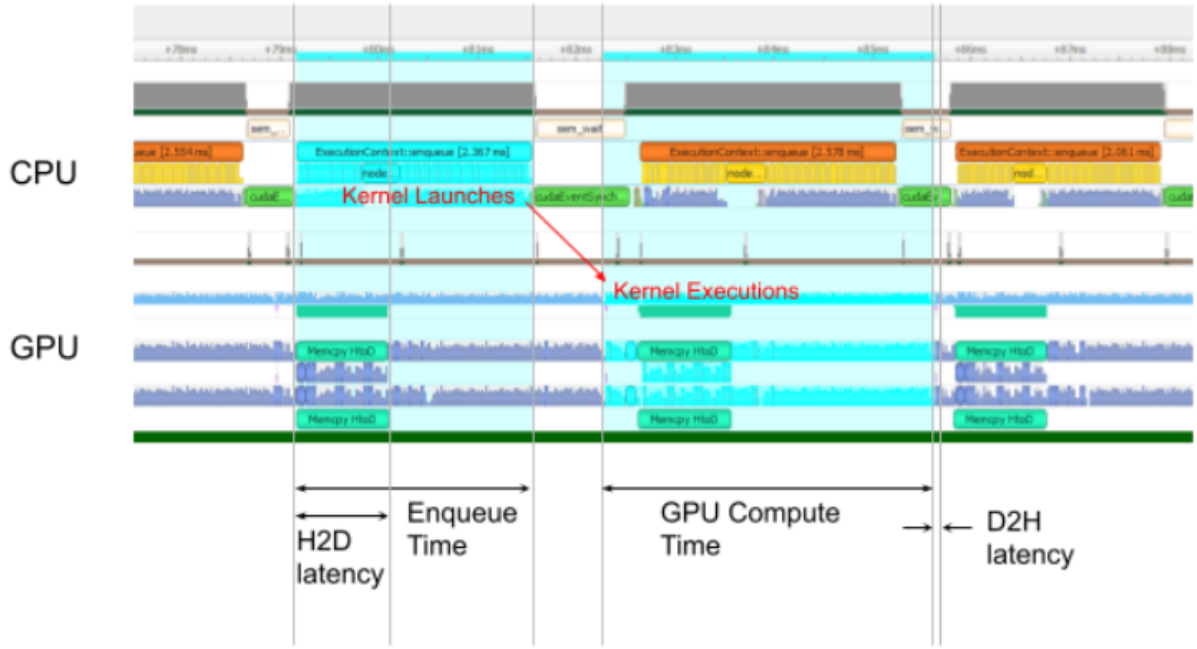
Total Host Walltime
The host walltime from when the first inference (after warm-ups) is enqueued to when the last inference was completed.

Total GPU Compute Time
The summation of the GPU Compute Time of all the inferences. If this is significantly shorter than Total Host Walltime, the GPU may be under utilized because of host-side overheads or data transfers.

Performance metrics in a normal `trtexec` run under Nsight Systems (ShuffleNet, BS=16, best, TitanRTX at 1200 MHz).

Note: In the latest Nsight Systems, the GPU rows appear above the CPU rows rather than beneath the CPU rows.

Figure 28. Performance Metrics in a Normal `trtexec` Run under Nsight Systems



Add the `--dumpProfile` flag to `trtexec` to show per-layer performance profiles, which allows users to understand which layers in the network take the most time in GPU execution. The per-layer performance profiling works with launching inference as a CUDA graph as well. In addition, build the engine with the `--profilingVerbosity=detailed` flag and add the `--dumpLayerInfo` flag to show detailed engine information, including per-layer detail and binding information. This allows you to understand which operation each layer in the engine corresponds to and their parameters.

A.2.1.2. Serialized Engine Generation

If you generate a saved serialized engine file, you can pull it into another application that runs inference. For example, you can use the NVIDIA Triton Inference Server to run the engine with multiple execution contexts from multiple threads in a fully pipelined asynchronous way to test parallel inference performance. There are some caveats; for

example, in INT8 mode, `trtexec` sets random dynamic ranges for tensors unless the calibration cache file is provided with the `--calib=<file>` flag, so the resulting accuracy will not be as expected.

A.2.1.3. Serialized Timing Cache Generation

If you provide a timing cache file to the `--timingCacheFile` option, the builder can load existing profiling data from it and add new profiling data entries during layer profiling. The timing cache file can be reused in other builder instances to improve the builder execution time. It is suggested to reuse this cache only in the same hardware/software configurations (for example, CUDA/cuDNN/TensorRT versions, device model, and clock frequency); otherwise, functional or performance issues may occur.

A.2.1.4. Commonly Used Command-line Flags

The section lists the commonly used `trtexec` command-line flags.

Flags for the Build Phase

- `--onnx=<model>`: Specify the input ONNX model.
- If the input model is in ONNX format, use the `--minShapes`, `--optShapes`, and `--maxShapes` flags to control the range of input shapes including batch size.
- `--minShapes=<shapes>`, `--optShapes=<shapes>`, and `--maxShapes=<shapes>`: Specify the range of the input shapes to build the engine with. Only required if the input model is in ONNX format.
- `--memPoolSize=<pool_spec>`: Specify the maximum size of the workspace that tactics are allowed to use, as well as the sizes of the memory pools that DLA will allocate per loadable. Supported pool types include `workspace`, `dlaSRAM`, `dlaLocalDRAM`, `dlaGlobalDRAM`, and `tacticSharedMem`.
- `--saveEngine=<file>`: Specify the path to save the engine to.
- `--fp16`, `--bf16`,`--int8`, `--fp8`,`--noTF32`, and `--best`: Specify network-level precision.
- `--stronglyTyped`: Create a strongly typed network.
- `--sparsity=[disable|enable|force]`: Specify whether to use tactics that support structured sparsity.
  - `disable`: Disable all tactics using structured sparsity. This is the default.
  - `enable`: Enable tactics using structured sparsity. Tactics will only be used if the weights in the ONNX file meet the requirements for structured sparsity.
  - `force`: Enable tactics using structured sparsity and allow trtexec to overwrite the weights in the ONNX file to enforce them to have structured sparsity patterns.

Note that the accuracy is not preserved, so this is to get inference performance only.

> Note: This has been deprecated. Use Polygraphy (`polygraphy surgeon prune`) to rewrite the weights of ONNX models to structured-sparsity pattern and then run with `--sparsity=enable`.

- `--timingCacheFile=<file>`: Specify the timing cache to load from and save to.
- `--noCompilationCache`: Disable compilation cache in builder, and the cache is part of timing cache (default is to enable compilation cache).
- `--verbose`: Turn on verbose logging.
- `--skipInference`: Build and save the engine without running inference.
- `--profilingVerbosity=[layer_names_only|detailed|none]`: Specify the profiling verbosity to build the engine with.
- `--dumpLayerInfo`, `--exportLayerInfo=<file>`: Print/Save the layer information of the engine.
- `--precisionConstraints=spec`: Control precision constraint setting.
  - `none`: No constraints.
  - `prefer`: Meet precision constraints set by `--layerPrecisions`/`--layerOutputTypes` if possible.
  - `obey`: Meet precision constraints set by `--layerPrecisions`/`--layerOutputTypes` or fail otherwise.
- `--layerPrecisions=spec`: Control per-layer precision constraints. Effective only when `precisionConstraints` is set to `obey` or `prefer`. The specs are read left to right, and later ones override earlier ones. "`*`" can be used as a `layerName` to specify the default precision for all the unspecified layers.
  - For example: `--layerPrecisions=*:fp16,layer_1:fp32` sets the precision of all layers to FP16 except for `layer_1`, which will be set to FP32.
- `--layerOutputTypes=spec`: Control per-layer output type constraints. Effective only when `precisionConstraints` is set to `obey` or `prefer`. The specs are read left to right, and later ones override earlier ones. "`*`" can be used as a `layerName` to specify the default precision for all the unspecified layers. If a layer has more than one output, then multiple types separated by "`+`" can be provided for this layer.
  - For example: `--layerOutputTypes=*:fp16,layer_1:fp32+fp16` sets the precision of all layer outputs to FP16 except for `layer_1`, whose first output will be set to FP32 and whose second output will be set to FP16.
- `--layerDeviceTypes=spec`: Explicitly set per-layer device type to either `GPU` or `DLA`. The specs are read left to right, and later ones override earlier ones.
- `--useDLACore=N`: Use the specified DLA core for layers that support DLA.
- `--allowGPUFallback`: Allow layers unsupported on DLA to run on GPU instead.

- `--versionCompatible`, `--vc`: Enable version compatible mode for engine build and inference. Any engine built with this flag enabled is compatible with newer versions of TensorRT on the same host OS when run with TensorRT's dispatch and lean runtimes. Only supported with explicit batch mode.
- `--excludeLeanRuntime`: When `--versionCompatible` is enabled, this flag indicates that the generated engine should not include an embedded lean runtime. If this is set, you must explicitly specify a valid lean runtime to use when loading the engine. Only supported with explicit batch and weights within the engine.
- `--tempdir=<dir>`: Overrides the default temporary directory TensorRT will use when creating temporary files. Refer to the `IRuntime::setTemporaryDirectory` API documentation for more information.
- `--tempfileControls=controls`: Controls what TensorRT is allowed to use when creating temporary executable files. Should be a comma-separated list with entries in the format `[in_memory|temporary]:[allow|deny]`.
  - Options include:
    - `in_memory`: Controls whether TensorRT is allowed to create temporary in-memory executable files.
    - `temporary`: Controls whether TensorRT is allowed to create temporary executable files in the filesystem (in the directory given by `--tempdir`).
  - Example usage: `--tempfileControls=in_memory:allow,temporary:deny`
- `--dynamicPlugins=<file>`: Load the plugin library dynamically and serialize it with the engine when it is included in `--setPluginsToSerialize` (can be specified multiple times).
- `--setPluginsToSerialize=<file>`: Set the plugin library to be serialized with the engine (can be specified multiple times).
- `--builderOptimizationLevel=N`: Set the builder optimization level to build the engine with. Higher level allows TensorRT to spend more building time for more optimization options.
- `--maxAuxStreams=N`: Set maximum number of auxiliary streams per inference stream that TRT is allowed to use to run kernels in parallel if the network contains ops that can run in parallel, with the cost of more memory usage. Set this to 0 for optimal memory usage. Refer to the Within-Inference Multi-Streaming section for more information.
- `--stripWeights`: Strip weights from plan. This flag works with either refit or refit with identical weights. Defaults to refit with identical weights, however, you can switch to refit by enabling both `--stripWeights` and `--refit` at the same time.
- `--markDebug`: Specify a list of tensor names to be marked as debug tensors. Separate names with a comma.
- `--allowWeightStreaming`: Enables an engine that can stream its weights. Must be specified with `--stronglyTyped`. TensorRT will automatically choose the appropriate

weight streaming budget at runtime to ensure model execution. A specific amount can be set with `--weightStreamingBudget`.

Flags for the Inference Phase

- `--loadEngine=<file>`: Load the engine from a serialized plan file instead of building it from the input ONNX model.
- If the input model is in ONNX format or if the engine is built with explicit batch dimension, use `--shapes` instead.
- `--shapes=<shapes>`: Specify the input shapes to run the inference with.
- `--loadInputs=<specs>`: Load input values from files. Default is to generate random inputs.
- `--warmUp=<duration in ms>`, `--duration=<duration in seconds>`, `--iterations=<N>`: Specify the minimum duration of the warm-up runs, the minimum duration for the inference runs, and the minimum iterations of the inference runs. For example, setting `--warmUp=0 --duration=0 --iterations=N` allows you to control exactly how many iterations to run the inference for.
- `--useCudaGraph`: Capture the inference to a CUDA graph and run inference by launching the graph. This argument may be ignored when the built TensorRT engine contains operations that are not permitted under CUDA graph capture mode.
- `--noDataTransfers`: Turn off host to device and device-to-host data transfers.
- `--useSpinWait`: Actively synchronize on GPU events. This option makes latency measurement more stable but increases CPU usage and power.
- `--infStreams=<N>`: Run inference with multiple cross-inference streams in parallel. Refer to the Cross-Inference Multi-Streaming section for more information.
- `--verbose`: Turn on verbose logging.
- `--dumpProfile, --exportProfile=<file>`: Print/Save the per-layer performance profile.
- `--dumpLayerInfo, --exportLayerInfo=<file>`: Print layer information of the engine.
- `--profilingVerbosity=[layer_names_only|detailed|none]`: Specify the profiling verbosity to run the inference with.
- `--useRuntime=[full|lean|dispatch]`: TensorRT runtime to execute engine. `lean` and `dispatch` require `--versionCompatible` to be enabled and are used to load a VC engine. All engines (VC or not) must be built with full runtime.
- `--leanDLLPath=<file>`: External lean runtime DLL to use in version compatible mode. Requires `--useRuntime=[lean|dispatch]`.
- `--dynamicPlugins=<file>`: Load the plugin library dynamically when the library is not included in the engine plan file (can be specified multiple times).

- `--getPlanVersionOnly`: Print TensorRT version when loaded plan was created. Works without deserialization of the plan. Use together with `--loadEngine`. Supported only for engines created with 8.6 and later.
- `--saveDebugTensors`: Specify list of tensor names to turn on the debug state and filename to save raw outputs to. These tensors must be specified as debug tensors during build time.
- `--allocationStrategy`: Specify how the internal device memory for inference is allocated. You can choose from `static`, `profile`, and `runtime`. The first option is the default behavior that pre-allocates enough size for all profiles and input shapes. The second option enables `trtexec` to only allocate what's required for the profile to use. The third option enables `trtexec` to only allocate what's required for the actual input shapes.
- `--weightStreamingBudget`: Manually set the weight streaming budget. Base-2 unit suffixes are supported: B (Bytes), G (Gibibytes), K (Kibibytes), M (Mebibytes). A value of 0 will choose the minimum possible budget if the weights don't fit on the device. A value of -1 will disable weight streaming at runtime.

Refer to `trtexec --help` for all the supported flags and detailed explanations.

Refer to the GitHub: trtexec/README.md file for detailed information about how to build this tool and examples of its usage.

A.3. Glossary

Data-Dependent Shape
A tensor shape with a dynamic dimension that is not calculated solely from network input dimensions and network input shape tensors.

Device
A specific GPU. Two GPUs are considered identical devices if they have the same model name and same configuration.

Explicitly Data-Dependent Shape
A tensor shape that depends on the dimensions of an output of `INonZeroLayer` or `INMSLayer`.

Implicitly Data-Dependent Shape
A tensor shape with a dynamic dimension that is calculated from data other than network input dimensions, network input shape tensors, and `INonZeroLayer` or `INMSLayer`. For example, a shape with a dimension calculated from data output by a convolution.

Platform
A combination of architecture and OS. Example platforms are Linux on x86 and QNX Standard on Aarch64. Platforms with different architectures or different OS are considered different platforms.

A.4. ACKNOWLEDGEMENTS

TensorRT uses elements from the following software, whose licenses are reproduced below.

Google Protobuf

This license applies to all parts of Protocol Buffers except the following:

- Atomicops support for generic gcc, located in `src/google/protobuf/stubs/atomicops_internals_generic_gcc.h`. This file is copyrighted by Red Hat Inc.
- Atomicops support for AIX/POWER, located in `src/google/protobuf/stubs/atomicops_internals_power.h`. This file is copyrighted by Bloomberg Finance LP.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Code generated by the Protocol Buffer compiler is owned by the owner of the input file used when generating it. This code is not standalone and requires a support library to be linked with it. This support library is itself covered by the above license.

Google Flatbuffers

Apache License Version 2.0, January 2004 http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing

lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a). You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b). You must cause any modified files to carry prominent notices stating that You changed the files; and

   c). You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   d). If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the

Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf

and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright 2014 Google Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at: http://www.apache.org/licenses/LICENSE-2.0.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

BVLC caffe

COPYRIGHT

All contributions by the University of California:

Copyright (c) 2014, 2015, The Regents of the University of California (Regents)

All rights reserved.

All other contributions:

Copyright (c) 2014, 2015, the respective contributors

All rights reserved.

Caffe uses a shared copyright model: each contributor holds copyright over their contributions to Caffe. The project versioning records all such contribution and copyright details. If a contributor wants to further mark their specific copyright on a particular contribution, they should indicate their copyright solely in the commit message of the change when it is committed.

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTION AGREEMENT

By contributing to the BVLC/caffe repository through pull-request, comment, or otherwise, the contributor releases their content to the license and copyright terms herein.

half.h

Copyright (c) 2012-2017 Christian Rau <rauy@users.sourceforge.net>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

jQuery.js

jQuery.js is generated automatically under doxygen.

In all cases TensorRT uses the functions under the MIT license.

CRC

TensorRT includes CRC routines from FreeBSD.

$FreeBSD: head/COPYRIGHT 260125 2013-12-31 12:18:10Z gjb $

@(#)COPYRIGHT 8.2 (Berkeley) 3/21/94

The compilation of software known as FreeBSD is distributed under the following terms:

Copyright (c) 1992-2014 The FreeBSD Project. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The 4.4BSD and 4.4BSD-Lite software is distributed under the following terms:

All of the documentation and software included in the 4.4BSD and 4.4BSD-Lite Releases is copyrighted by The Regents of the University of California.

Copyright 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994 The Regents of the University of California. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement: This product includes software developed by the University of California, Berkeley and its contributors.
4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The Institute of Electrical and Electronics Engineers and the American National Standards Committee X3, on Information Processing Systems have given us permission to reprint portions of their documentation.

In the following statement, the phrase ``this text'' refers to portions of the system documentation.

Portions of this text are reprinted and reproduced in electronic form in the second BSD Networking Software Release, from IEEE Std 1003.1-1988, IEEE Standard Portable Operating System Interface for Computer Environments (POSIX), copyright C 1988 by the Institute of Electrical and Electronics Engineers, Inc. In the event of any discrepancy between these versions and the original IEEE Standard, the original IEEE Standard is the referee document.

In the following statement, the phrase ``This material'' refers to portions of the system documentation.

This material is reproduced with permission from American National Standards Committee X3, on Information Processing Systems. Computer and Business Equipment Manufacturers Association (CBEMA), 311 First St., NW, Suite 500, Washington, DC 20001-2178. The developmental work of Programming Language C was completed by the X3J11 Technical Committee.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Regents of the University of California.

Note: The copyright of UC Berkeley's Berkeley Software Distribution ("BSD") source has been updated. The copyright addendum may be found at ftp://ftp.cs.berkeley.edu/pub/4bsd/README.Impt.License.Change and is included below.

July 22, 1999

To All Licensees, Distributors of Any Version of BSD:

As you know, certain of the Berkeley Software Distribution ("BSD") source code files require that further distributions of products containing all or portions of the software, acknowledge within their advertising materials that such products contain software developed by UC Berkeley and its contributors.

Specifically, the provision reads:

" * 3. All advertising materials mentioning features or use of this software

* must display the following acknowledgement:

* This product includes software developed by the University of

* California, Berkeley and its contributors."

Effective immediately, licensees and distributors are no longer required to include the acknowledgement within advertising materials. Accordingly, the foregoing paragraph of those BSD Unix files containing it is hereby deleted in its entirety.

William Hoskins

Director, Office of Technology Licensing

University of California, Berkeley

getopt.c

$OpenBSD: getopt_long.c,v 1.23 2007/10/31 12:34:57 chl Exp $

$NetBSD: getopt_long.c,v 1.15 2002/01/31 22:43:40 tv Exp $

Copyright (c) 2002 Todd C. Miller <Todd.Miller@courtesan.com>

Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES

WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F39502-99-1-0512.

Copyright (c) 2000 The NetBSD Foundation, Inc.

All rights reserved.

This code is derived from software contributed to The NetBSD Foundation by Dieter Baron and Thomas Klausner.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ONNX Model Zoo

MIT License

Copyright (c) ONNX Project Contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE

RESNET-50 Caffe models

The MIT License (MIT)

Copyright (c) 2016 Shaoqing Ren

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

OpenSSL

Apache License Version 2.0

Copyright (c) OpenSSL Project Contributors

Apache License

Version 2.0, January 2004

https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works

of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a). You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b). You must cause any modified files to carry prominent notices stating that You changed the files; and

   c). You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   d). If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

Boost Beast

Copyright (c) 2016-2017 Vinnie Falco (vinnie dot falco at gmail dot com)

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Arm

Arm, AMBA and Arm Powered are registered trademarks of Arm Limited. Cortex, MPCore and Mali are trademarks of Arm Limited. "Arm" is used to represent Arm Holdings plc; its operating company Arm Limited; and the regional subsidiaries Arm Inc.; Arm KK; Arm Korea Limited.; Arm Taiwan Limited; Arm France SAS; Arm Consulting (Shanghai) Co. Ltd.; Arm Germany GmbH; Arm Embedded Technologies Pvt. Ltd.; Arm Norway, AS and Arm Sweden AB.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

Blackberry/QNX

Copyright © 2020 BlackBerry Limited. All rights reserved.

Trademarks, including but not limited to BLACKBERRY, EMBLEM Design, QNX, AVIAGE, MOMENTICS, NEUTRINO and QNX CAR are the trademarks or registered trademarks of BlackBerry Limited, used under license, and the exclusive rights to such trademarks are expressly reserved.

Google

Android, Android TV, Google Play and the Google Play logo are trademarks of Google, Inc.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com