NVIDIA TensorRT 11.0.1 Developer Guide

Release 7.2.5 for NVIDIA DriveOS

NVIDIA Corporation

Jun 25, 2026

Contents

This guide explains how to use the C++ and Python APIs to implement common deep learning layers. You can take a model from a deep learning framework, parse it with the provided parsers, and build a TensorRT engine.

Important

This book is part of the **NVIDIA TensorRT 11.0.1 for DriveOS 7.2.5** PDF documentation set. Companion PDFs in the same release include the *Installation Guide*, *Migration Guide*, *Release Notes*, and *Safety Production Guide*.

Note

Scope for DriveOS 7.2.5

- **DLA (Deep Learning Accelerator) is not supported** in this release. Use GPU or safety-runtime execution paths. See *Known Limitations* in the *Release Notes* and *Migrating TensorRT from 10.x to 11.x on NVIDIA DriveOS* in the *Migration Guide*.
- **PDF delivery:** This bundle does not include the general-release online API reference tree or Samples Explorer. Symbol-level API usage is covered in *C++ API Documentation* and *Python API Documentation*.
- **TensorRT 11.x is strongly typed.** If you are upgrading from 10.x, read the *Migration Guide* before changing build or precision code.

Start here

1. Set up TensorRT on your platform — *Installation Guide* (companion PDF).
2. Upgrade from TensorRT 10.x — *Migration Guide* (companion PDF), including the DriveOS platform chapter.
3. Understand runtime objects, memory, and threading — *How TensorRT Works*.
4. Build and run inference — *C++ API Documentation* or *Python API Documentation*.
5. ISO 26262 / safety-runtime workflows — *Safety Production Guide* (companion PDF).
6. What changed in this Developer Guide — *Developer Guide Revision History*.

Chapter 1. Developer Guide Revision History

Table 1.1: Document Revision History

Date	Summary of Change
March 31, 2026	Initial draft for NVIDIA TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5
April 1, 2026	Start of review
June 21, 2026	End of review
June 22, 2026	Approval review

Table 1.2: Document Changes

Date	Summary of Change
May 8, 2026	▶ Multi-Device Inference (Preview) - Scale inference across multiple GPUs with `IDistCollectiveLayer` and multi-device attention using NCCL ▶ MoE (Mixture of Experts) - Built-in `IMoELayer` for transformer MoE blocks on SM110 with NVFP4/FP8 quantization ▶ API Capture and Replay Multi-Network Support - Capture and replay multiple networks within a single process for ensemble models and multi-stage inference pipelines ▶ Breaking ABI Changes - Windows DLL files moved from `lib/` to `bin/` subdirectory; `libonnx_proto.a` merged into `libnvonnxparser_static.a` ▶ Working With Transformers - New chapter for transformer models, including `IRotaryEmbeddingLayer` (RoPE) and related topics.
May 11, 2026	▶ Removed or contextualized DLA references throughout the guide. DLA execution is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5; the `kDLA_LINEAR` and `kDLA_HWC4` tensor-format constants and the **DLA** glossary entry remain documented for completeness, with a callout noting that DLA is unavailable on this platform. ▶ Removed the **Profiling for DLA** subsection from **Performance Benchmarking** and the corresponding row from the profiling-tools comparison table. ▶ Removed the `--useDLACore=N` and `--allowGPUFallback` `trtexec` flags from the build-phase flag table, and trimmed the DLA-only pool types (`dlaSRAM`, `dlaLocalDRAM`, `dlaGlobalDRAM`) from the `--memPoolSize` description.

Chapter 2. Architecture Overview

This section provides an overview of TensorRT's architecture, design principles, and ecosystem. It introduces key concepts and complementary tools that work alongside TensorRT for optimized inference deployment.

2.1. Samples

The TensorRT Sample Support Guide illustrates many of the topics discussed in this section.

2.2. Complementary GPU Features

Multi-instance GPU (MIG) is a feature of NVIDIA GPUs with NVIDIA Ampere Architecture or later architectures. It enables user-directed partitioning of a single GPU into multiple smaller GPUs.

The physical partitions provide dedicated compute and memory slices with quality of service. They also support independent execution of parallel workloads on fractions of the GPU.

For TensorRT applications with low GPU utilization, MIG can produce higher throughput with little or no latency impact. The optimal partitioning scheme is application-specific.

2.3. Complementary Software

Tool	Description
NVIDIA Triton Inference Server	Higher-level library providing optimized inference across CPUs and GPUs with model management, REST, and gRPC endpoints.
NVIDIA DALI	High-performance primitives for preprocessing image, audio, and video data. TensorRT inference can be integrated as a custom operator in a DALI pipeline. See GitHub: DALI for a working example.
Torch-TensorRT	PyTorch-TensorRT compiler that converts PyTorch modules into TensorRT engines. Subgraphs are accelerated through TensorRT while Torch executes the rest natively. See GitHub: Examples.
Model Optimizer	Unified library for quantization, pruning, and distillation. Compresses models for TensorRT-LLM or TensorRT deployment. Replaces the deprecated PyTorch and TensorFlow Quantization Toolkits. To quantize TensorFlow models, export to ONNX first.
NVIDIA Nsight Systems	Profiling tool integrated with TensorRT for performance analysis.
Nsight Deep Learning Designer	IDE for ONNX model editing, performance profiling, and TensorRT engine building.
NVIDIA DRIVE	A restricted subset of TensorRT is certified for NVIDIA DRIVE products. Some APIs are marked for DRIVE use only.

2.4. ONNX

TensorRT's primary means of importing a trained model from a framework is the ONNX interchange format. TensorRT ships with an ONNX parser library to assist in importing models. Where possible, the parser is backward compatible to opset 9. The ONNX Model Opset Version Converter can assist in resolving incompatibilities.

The GitHub version can 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 on GitHub: Supported ONNX Operators.

PyTorch natively supports ONNX export. For TensorFlow, use tf2onnx.

After exporting a model to ONNX, run constant folding using Polygraphy as a good first step. This often solves TensorRT conversion issues in the ONNX parser and simplifies the workflow. For details, refer to this example. In some cases, modifying the ONNX model can be necessary, such as replacing subgraphs with plugins or reimplementing unsupported operations in other operations. To make this process easier, use ONNX-GraphSurgeon.

2.5. Code Analysis Tools

For guidance using the Valgrind and Clang sanitizer tools with TensorRT, refer to the *Troubleshooting* section.

2.6. 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.

Warning

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. Some exceptions exist for the safety runtime as detailed in the NVIDIA DriveOS Developer Guide.

Calibration caches can typically be reused within a major version, but compatibility beyond a specific patch version is not guaranteed.

2.7. Deprecation Policy

Deprecation informs developers that some APIs and tools are no longer recommended. TensorRT has the following deprecation policy, beginning with version 8.0:

- Deprecation notices are communicated in the *Release Notes*.
- When using 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.

2.8. 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 were deprecated in TensorRT 8.6. TensorRT 10.4 was the last release supporting NVIDIA Volta (SM 7.0) devices. Refer to the TensorRT Support Matrix for more information.

2.9. Support

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

You can also 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.

2.10. 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 3. How TensorRT Works

NVIDIA TensorRT is an SDK that takes a trained deep learning model and turns it into a fast, GPU-specific program for running *inference* (computing predictions on new inputs after training is complete). It does this in two phases:

- A **build phase** where a *builder* compiles your network and selects the fastest available *kernel* (a low-level GPU function) for each layer on your target GPU. The output is a serialized binary called an *engine* (also referred to as a *plan file*).
- A **runtime phase** where a *runtime* loads the engine into your application and executes it on the GPU.

This page explains the runtime-side concepts you need to build a correct, reliable application on top of TensorRT - what each TensorRT object is for, how long each one needs to live, how memory is allocated and reused, what is and is not thread-safe, what determinism guarantees you get, and which runtime library to ship.

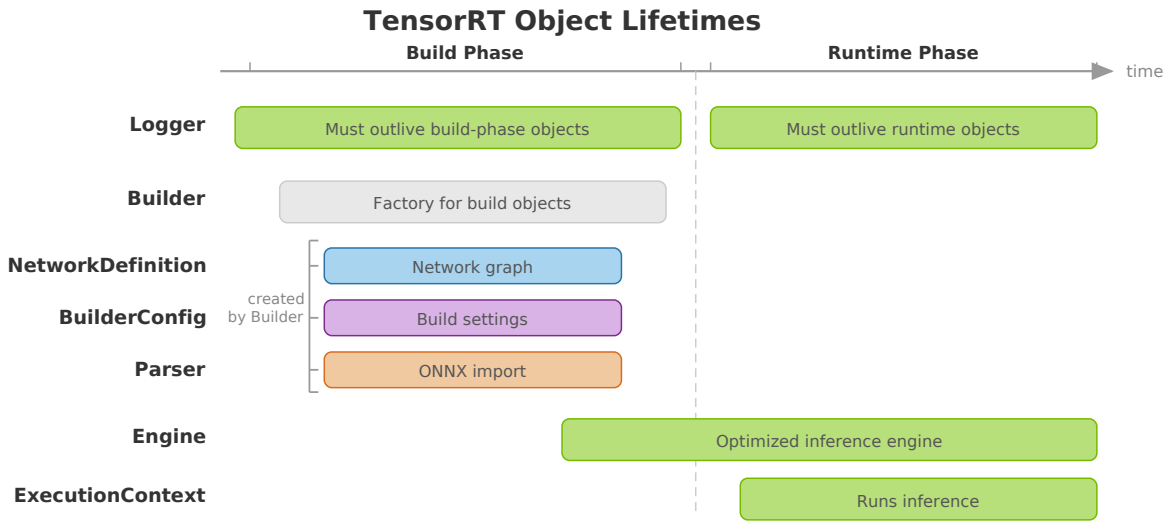
What you will learn

- TensorRT object ownership and lifetime requirements.
- Where TensorRT allocates host versus device memory, and how to control or override those allocations.
- Which operations are thread-safe versus require one execution context per thread.
- What the **Lean Runtime** and **Dispatch Runtime** trade off against the full runtime when you ship to production.

If you have not built your first engine yet, start with the TensorRT Quick Start Guide and come back here when you are ready to embed TensorRT into a real application.

3.1. Object Lifetimes

TensorRT uses a factory pattern where some objects create and own other objects. Correct object lifetime ordering is critical to avoiding crashes and undefined behavior, especially when you destroy a builder or runtime after the build phase completes.



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. 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 creating an engine, you can destroy the builder, network, parser, and build config and continue using the engine.

3.2. Error Handling and Logging

When creating TensorRT top-level interfaces (builder, runtime, or refitter), you must provide an implementation of the *Logger* interface. The logger is used for diagnostics and informational messages. Its verbosity level is configurable. Because the logger can pass back information at any point in TensorRT's lifetime, it must span any use of that interface in your application. The implementation must also be thread-safe because TensorRT can 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 created from a runtime, so TensorRT will use the logger associated with that runtime.

The primary method of error handling is the *ErrorRecorder* interface. You can implement this interface and attach it to an API object to receive errors associated with it. The recorder for an object is also passed to any objects it creates. For example, if you attach an error recorder to an engine and create an execution context from that engine, it uses the same recorder. If you then attach a new error recorder to the execution context, it receives only errors from that context. If an error is generated but no error recorder is found, it is emitted through the associated logger.

Caution

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

3.3. 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. Because device memory is often a constrained resource, it is important to understand how TensorRT uses it.

3.3.1. The Build Phase

During the build, TensorRT allocates device memory for timing layer implementations. Some implementations can consume large amounts 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 the device's global memory but can be restricted when necessary. If the builder finds applicable kernels that could not be run because of insufficient workspace, it emits a logging message indicating this.

Even with relatively little workspace, timing requires creating buffers for input, output, and weights. TensorRT handles the operating system (OS) returning out-of-memory for such allocations. On some platforms, the OS can successfully provide memory, and 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, at least two copies of the weights are typically in host memory: those from the original network and those included as part of the engine as it is built. When TensorRT combines weights, such as convolution with batch normalization, additional temporary weight tensors are created.

3.3.2. The Runtime Phase

TensorRT uses relatively little host memory at runtime but can use considerable device memory.

An engine allocates device memory to store the model weights upon deserialization. Since the serialized engine has almost all the weights, its size approximates the amount of device memory the weights require.

TensorRT provides an API `ICudaEngine::getEngineStat()` to retrieve detailed statistics about the engine, including precise weight sizes. Using the `EngineStat` enum, you can query the following:

- `kTOTAL_WEIGHTS_SIZE`: Returns the total size in bytes of all weights utilized by the engine.
- `kSTRIPPED_WEIGHTS_SIZE`: Returns the size in bytes of stripped weights for engines built with the `BuilderFlag::kSTRIP_PLAN` flag.

Note that when the weight streaming feature (`BuilderFlag::kWEIGHT_STREAMING`) is enabled, weights might not be fully copied to the device. The total weight size returned by `getEngineStat(kTOTAL_WEIGHTS_SIZE)` reflects the sum of all weights used by the engine, which can differ from the actual allocated GPU memory for weights.

If querying `kSTRIPPED_WEIGHTS_SIZE` on a normal engine without stripping, the function returns `-1` to indicate an invalid query.

This API enables programs to accurately monitor and manage weight memory usage within the engine beyond relying on the serialized engine alone.

An `ExecutionContext` uses two kinds of device memory:

- Some layer implementations require persistent memory. 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 can vary across contexts. This memory is allocated to the creation of the execution context and lasts for its lifetime.
- Enqueue memory is used to hold intermediate results while processing the network. This memory is used for intermediate tensors (activation memory) and for temporary storage required by layer implementations (scratch memory). The bound for scratch memory is controlled through `IBuilderConfig::setMemoryPoolLimit()`. TensorRT optimizes memory usage in the following ways:
  - Sharing a block of device memory across activations tensors with disjoint lifetimes
  - Allowing transient (scratch) tensors to occupy unused activation memory where feasible

Therefore, enqueue memory required by TensorRT is in the range of {total activation memory, total activation memory + max scratch memory}.

You can optionally create an execution context without enqueue memory using `ICudaEngine::createExecutionContextWithoutDeviceMemory()` and provide that memory 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 enqueue memory required is returned through `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] Max Scratch Memory: 9970688
```

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

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.

TensorRT measures the memory used before and after critical operations in the 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 that memory use changes with 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 through `cudaGetMemInfo` and TensorRT is prone to race conditions, where a new allocation or free is done through a different process or thread. Because CUDA does not control memory on unified-memory devices, the results returned through `cudaGetMemInfo` can be inaccurate on these platforms.

3.3.3. CUDA Lazy Loading

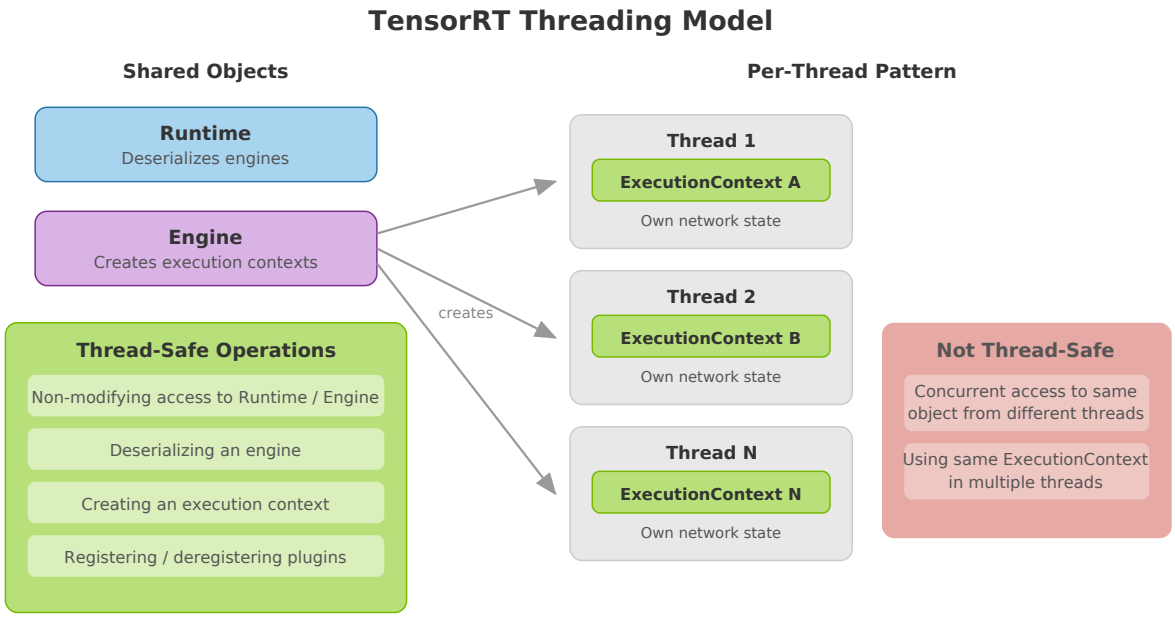
CUDA lazy loading is a CUDA feature that can significantly reduce the peak GPU and host memory usage of TensorRT and speed up TensorRT initialization with negligible (<1%) performance impact. The memory usage and time savings for initialization depend on the model, software stack, and GPU platform. Enable it using the environment variable `CUDA_MODULE_LOADING=LAZY`. For more information, refer to the NVIDIA CUDA documentation.

3.3.4. L2 Persistent Cache Management

NVIDIA Ampere and later architectures support L2 cache persistence, a feature that allows prioritization of L2 cache lines for retention when a line is chosen for eviction. TensorRT uses this to retain cache activations, 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`. For more information, refer to the NVIDIA CUDA Best Practices Guide.

3.4. Threading



TensorRT objects are generally not thread-safe; the client must serialize access to objects from different threads.

The expected runtime concurrency model is that different threads operate in different execution contexts. The context contains the network state (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.

3.5. Determinism

The TensorRT builder uses timing to find the fastest kernel to implement a given layer. Timing kernels are subject to noise, such as other work running on the GPU and GPU clock speed fluctuations. Timing noise means the same implementation can be selected differently on successive builder runs.

Different implementations use different orders 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 through 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 different kernel selection include different input sizes, such as batch size, or a different optimization point for an input profile. Refer to the *Working with Dynamic Shapes* section for more information.

The Editable Timing Cache mechanism allows you to force the builder to pick a particular implementation for a given layer. Use this to ensure the builder picks the same kernels 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 produces the same output.

While TensorRT can maintain determinism for identical inputs across runs, determinism can be compromised when feeding the same input at different slots within a batch or when that input is batched with different neighbors, potentially leading to different outputs.

3.5.1. `IFillLayer` Determinism

When `IFillLayer` is added to a network using 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. TensorRT supports distributive independence feature at build time (`BuilderFlag::kDISTRIBUTIVE_INDEPENDENCE`) to eliminate this difference with some constraints, refer to the *Distributive Independence Determinism* section.

3.5.2. `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 `ScatterMode::kELEMENT` and `ScatterMode::kND` modes, and one of the values from the input updates tensor will be picked arbitrarily.

3.5.3. Distributive Independence Determinism

When `BuilderFlag::kDISTRIBUTIVE_INDEPENDENCE` is set and a layer documents axis `i` of an output as a distributive axis, TensorRT guarantees that the layer behaves exactly as if each evaluation across axis `i` was done using identical operations. That is, if some inputs are identical across the distributive axis, the corresponding outputs of these inputs are guaranteed to be identical.

TensorRT defines the concept of distributive axis as follows:

- For `IMatrixMultiplyLayer`: All axes that are not one of the vector or matrix dimensions are distributive axes.
- For layers that perform reduction: All non-reduction axes are distributive axes.
- For layers that perform einsum: Let `n` be the leftmost reduction axis. The axes to the left of `n` are distributive axes.

Note that the distributive independence feature has the following constraints:

- With `BuilderFlag::kDISTRIBUTIVE_INDEPENDENCE` enabled, multiple profiles are disabled because kernels in different profiles can be different, so the results can be slightly different.
- With `BuilderFlag::kDISTRIBUTIVE_INDEPENDENCE` enabled, there might be a performance drop, especially for dynamic shapes cases.

3.6. 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 smaller than the default library and contains only the code necessary to run a version-compatible engine. It has some restrictions; primarily, some operator implementations are not supported.
- The *dispatch runtime* (`libnvinfer_dispatch.so/.dll`) is a small shim library that can load a lean runtime and redirect calls. The dispatch runtime can load older versions of the lean runtime and, together with the 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. Still, 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 must specify that the engine should be built for the lean runtime by enabling version compatibility. It can be 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.

For details on ensuring engines work across TensorRT versions and GPU models, refer to *Version Compatibility* and *Hardware Compatibility*.

3.7. 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 to relax the constraints on TensorRT versions and GPU models.

Chapter 4. TensorRT's Capabilities

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

4.1. C++ and Python APIs

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

Note

The Python API is not available for all platforms, such as QNX. For more information, refer to the TensorRT Support Matrix.

4.2. 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.

4.2.1. The Build Phase

The highest-level interface for the build phase of TensorRT is the *Builder*. The builder is responsible for optimizing a model and producing an *Engine*.

To build an engine, you must:

1. Create a network definition.
2. Specify a configuration for the builder.
3. Call the builder to create the engine.

The *NetworkDefinition* interface defines the model. The most common way 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, other approaches can be more suitable in some cases:

- PyTorch models: Use Torch-TensorRT. For better use of the latest quantization schemes, consider using ModelOpt to add quantization and export the model as ONNX.
- Python API (step-by-step network construction): TriPy.
- For all other frameworks, such as TensorFlow: Convert to ONNX and use TensorRT's ONNX parser.

You can also populate `NetworkDefinition` by adding layers one at a time using its `add...` methods and the interfaces for `Layer` and `Tensor`.

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

The *BuilderConfig* interface specifies 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. Because 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, reorders, and combines operations to run more efficiently on the GPU.

It can reduce the precision of floating-point computations by 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 has multiple implementations for each layer with varying data formats. Then, it 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, TensorRT engines are specific to both the TensorRT version and the GPU that they were created on. To configure an engine for forward compatibility, refer to the *Version Compatibility* and *Hardware Compatibility* sections.
- TensorRT's network definition does not deep-copy parameter arrays, such as the weights for a convolution. Do 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 and 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 can perturb the timings, resulting in poor optimization.

4.2.2. The Runtime Phase

The highest-level interface for the execution phase of TensorRT is the *Runtime*.

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

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

Chapter 5. C++ API Documentation

Attention

- The TensorRT C++ API enables developers to import, calibrate, generate, and deploy networks using C++. Networks can be imported directly from ONNX. They can also be created programmatically by instantiating individual layers and setting parameters and weights directly.
- To view API changes between releases, refer to the TensorRT GitHub repository and use the compare tool.

This section illustrates the basic usage of the C++ API, assuming you start with an ONNX model. The sampleOnnxMNIST sample 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`, such as `ILogger` and `IBuilder`.

A CUDA context is automatically created the first time TensorRT calls CUDA if none exists before that point. However, creating and configuring the CUDA context yourself is generally preferable before the first call to TensorRT.

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

5.1. 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;
```

(continues on next page)

(continued from previous page)

```
    }
} logger;
```

You can then create an instance of the builder:

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

5.1.1. 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.

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

Finally, create a network:

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

5.1.1.1 Creating a Network Definition from Scratch (Advanced)

Instead of using a parser, you can define the network directly to TensorRT through 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.

This example creates a simple network with Input, Convolution, Pooling, MatrixMultiply, Shuffle, Activation, and Softmax layers. It also loads the weights into a weightMap data structure, which is used in the following code.

First, create the builder and network objects. Note that 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()));
auto network = SampleUniquePtr<nvinfer1::INetworkDefinition>(builder->
↪createNetworkV2(0));
```

Add the Input layer to the network by specifying the input tensor's name, datatype, and full dimensions. 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 matrix multiplication:

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

Now, add a MatrixMultiply layer. The model exporter provided transposed weights, so the kTRANSPOSE option is specified.

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

Add the ReLU Activation layer:

```
auto relu1 = network->addActivation(*biasAdd->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. For instructions on how to build an engine and run an inference with this network, refer to the *Building an Engine* and *Performing Inference* sections.

5.1.2. 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 the builder copies into the optimized engine. Since the network was created using the parser, the parser owns the memory occupied by the weights, so the parser object should not be deleted until after the builder has run.

5.1.2.1 Importing a Model Using the ONNX Parser with Custom Weights

The ONNX parser API allows users to provide their own weights, also known as ONNX initializers, in host memory to override any weights found in the model. Instead of parsing the model immediately, the following APIs can be used to load custom weights:

```
#include "NvOnnxParser.h"
using namespace nvonnxparser;
IParser* parser = createParser(*network, logger);
```

Load the model into the parser.

```
parser->loadModelProto(modelData, modelSize);
```

Provide the name, pointer to data, and size of the data for the parser to use instead of the one found in the model. This command can be called multiple times to override multiple weights. Note that these pointers must remain in scope until the parser is destroyed.

```
parser->loadInitializer(name, data, dataSize);
```

Begin parsing with user-defined weights.

```
parser->parseModelProto();
```

The same idea extends to the `IParserRefitter` class, and similar APIs can be used to provide custom weights when refitting an engine built from an ONNX model. For more information, refer to the *Refitting a Weight-Stripped Engine Directly from ONNX* section.

5.1.3. 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 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, TensorRT might 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, such as 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 both TensorRT and DirectX concurrently utilize the GPU.

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

After 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 can be safely deleted:

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

The engine can then be saved to disk, and the buffer that holds the serialized data can be deleted.

```
delete serializedModel;
```

Note

Serialized engines are not cross-platform portable. They are specific to the exact GPU model that they were built for (in addition to the platform).

Building engines is intended as an offline process, so it can take significant time. The *Reducing Engine Build Time* section has tips on making the builder run faster.

5.2. Deserializing a Plan

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

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

TensorRT provides three main methods to deserialize an engine, each with its use case and benefits.

In-memory Deserialization

This method is straightforward and suitable for smaller models or when memory is not a constraint.

```
std::vector<char> modelData = readModelFromFile("model.plan");
ICudaEngine* engine = runtime->deserializeCudaEngine(modelData.data(),
↪modelData.size());
```

IStreamReaderV2 Deserialization

This method allows for a more controlled reading of the engine file and is useful for custom file handling or weight streaming. It supports reading from both host and device pointers and enables potential performance improvements. With this approach, reading the entire plan file into a buffer to be deserialized is unnecessary, as `IStreamReaderV2` allows reading the file in chunks as needed, and possibly bypassing the CPU, thus reducing peak CPU memory usage.

```
class MyStreamReaderV2 : public IStreamReaderV2 {
    // Custom implementation with support for device memory reading
};
MyStreamReaderV2 readerV2("model.plan");
ICudaEngine* engine = runtime->deserializeCudaEngine(readerV2);
```

The `IStreamReaderV2` approach is particularly beneficial for large models or when using advanced features like GPUDirect or weight streaming. It can significantly reduce engine load time and memory usage.

When choosing a deserialization method, consider your specific requirements:

- For small models or simple use cases, in-memory deserialization is often sufficient.
- For large models or when memory efficiency is crucial, consider using `IStreamReaderV2`.
- If you need custom file handling or weight streaming capabilities, `IStreamReaderV2` provides the necessary flexibility.

5.3. Performing Inference

The engine holds the optimized model, but you must manage additional states for intermediate activations to perform inference. 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.

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 tensor's name and the buffer's address. 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, loops, and synchronous plugins. 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.

5.4. Complete End-to-End Example

The snippets above introduce each C++ API class in isolation. The program below stitches them into a single copy-paste-ready file that goes from an ONNX model to a benchmarked engine to a live inference call. Save it as `trt_end_to_end.cpp`, build with the command line at the bottom, and run it against any ONNX model - for example, the ResNet-50 v1 model referenced earlier in this section.

Listing 5.1: trt_end_to_end.cpp — build an engine from ONNX and run inference

```
#include <cassert>
#include <fstream>
#include <iostream>
#include <memory>
#include <random>
#include <vector>

#include <cuda_runtime.h>

#include "NvInfer.h"
#include "NvOnnxParser.h"

using namespace nvinfer1;

class Logger : public ILogger
{
    void log(Severity severity, char const* msg) noexcept override
    {
        if (severity <= Severity::kWARNING)
        {
            std::cout << msg << std::endl;
        }
    }
} gLogger;

// Build a strongly typed TensorRT engine from an ONNX file and save it to disk.
std::vector<char> buildEngine(char const* onnxPath, char const* enginePath)
```

(continues on next page)

(continued from previous page)

```
{
    std::unique_ptr<IBuilder> builder{createInferBuilder(gLogger)};
    uint32_t const flag = 1U << static_cast<uint32_t>(
        NetworkDefinitionCreationFlag::kSTRONGLY_TYPED);
    std::unique_ptr<INetworkDefinition> network{builder->
↪createNetworkV2(flag)};
    std::unique_ptr<nvonnxparser::IParser> parser{
        nvonnxparser::createParser(*network, gLogger)};
    if (!parser->parseFromFile(
            onnxPath, static_cast<int32_t>(ILogger::Severity::kWARNING)))
    {
        for (int32_t i = 0; i < parser->getNbErrors(); ++i)
        {
            std::cerr << parser->getError(i)->desc() << std::endl;
        }
        throw std::runtime_error("Failed to parse ONNX file");
    }

    std::unique_ptr<IBuilderConfig> config{builder->createBuilderConfig()};
    config->setMemoryPoolLimit(MemoryPoolType::kWORKSPACE, 1ULL << 30); // 1
↪GiB

    std::unique_ptr<IHostMemory> serialized{
        builder->buildSerializedNetwork(*network, *config)};
    if (!serialized)
    {
        throw std::runtime_error("Engine build failed");
    }
    std::ofstream out(enginePath, std::ios::binary);
    out.write(static_cast<char const*>(serialized->data()), serialized->
↪size());
    return std::vector<char>(
        static_cast<char const*>(serialized->data()),
        static_cast<char const*>(serialized->data()) + serialized->size());
}

// Deserialize the engine and run a single inference on the supplied input.
std::vector<float> runInference(
    std::vector<char> const& engineBytes, std::vector<float> const& hostInput,
    Dims const& inputDims)
{
    std::unique_ptr<IRuntime> runtime{createInferRuntime(gLogger)};
    std::unique_ptr<ICudaEngine> engine{
        runtime->deserializeCudaEngine(engineBytes.data(), engineBytes.
↪size())};
    std::unique_ptr<IExecutionContext> context{engine->
↪createExecutionContext()};

    char const* const inputName = engine->getIOTensorName(0);
    char const* const outputName = engine->getIOTensorName(1);
    context->setInputShape(inputName, inputDims);
    Dims const outputDims = context->getTensorShape(outputName);
```

(continues on next page)

(continued from previous page)

```
    size_t outputCount = 1;
    for (int32_t i = 0; i < outputDims.nbDims; ++i)
    {
        outputCount *= outputDims.d[i];
    }
    std::vector<float> hostOutput(outputCount);

    void* dInput{nullptr};
    void* dOutput{nullptr};
    cudaMalloc(&dInput, hostInput.size() * sizeof(float));
    cudaMalloc(&dOutput, hostOutput.size() * sizeof(float));
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    cudaMemcpyAsync(dInput, hostInput.data(), hostInput.size() *
↪sizeof(float),
        cudaMemcpyHostToDevice, stream);
    context->setTensorAddress(inputName, dInput);
    context->setTensorAddress(outputName, dOutput);
    if (!context->enqueueV3(stream))
    {
        throw std::runtime_error("enqueueV3 failed");
    }
    cudaMemcpyAsync(hostOutput.data(), dOutput,
        hostOutput.size() * sizeof(float), cudaMemcpyDeviceToHost, stream);
    cudaStreamSynchronize(stream);

    cudaFree(dInput);
    cudaFree(dOutput);
    cudaStreamDestroy(stream);
    return hostOutput;
}

int main(int argc, char** argv)
{
    char const* const onnxPath = (argc > 1) ? argv[1] : "resnet50-v1-12.onnx";
    char const* const enginePath = "model.engine";

    auto engineBytes = buildEngine(onnxPath, enginePath);

    // Replace this with a preprocessed image batch in your own application.
    std::vector<float> hostInput(1 * 3 * 224 * 224);
    std::mt19937 rng{0};
    std::uniform_real_distribution<float> dist{0.0f, 1.0f};
    for (auto& v : hostInput) v = dist(rng);

    auto hostOutput = runInference(
        engineBytes, hostInput, Dims4{1, 3, 224, 224});

    size_t topClass = 0;
    for (size_t i = 1; i < hostOutput.size(); ++i)
```

(continues on next page)

(continued from previous page)

```
    {
        if (hostOutput[i] > hostOutput[topClass]) topClass = i;
    }
    std::cout << "Output size: " << hostOutput.size() << "\n";
    std::cout << "Top-1 class index: " << topClass << "\n";
    return 0;
}
```

Build and run, replacing the include and library paths with your TensorRT install location:

```
g++ -std=c++17 trt_end_to_end.cpp \
    -I/usr/include/x86_64-linux-gnu -I/usr/local/cuda/include \
    -L/usr/lib/x86_64-linux-gnu -L/usr/local/cuda/lib64 \
    -lnvinfer -lnvonnxparser -lcudart \
    -o trt_end_to_end
./trt_end_to_end resnet50-v1-12.onnx
```

On first invocation the program builds and serializes `model.engine`, then runs one inference on a random tensor sized for ResNet-50. Once you have a working baseline, swap the random input for a preprocessed image batch, plug the program into your application, and use the focused sections above when you need finer control over network construction, custom weights, alternate deserialization paths, or asynchronous inference scheduling.

5.5. Next Steps

See also

Optimizing Performance
Benchmarking methodology and best practices for maximizing inference throughput and latency.

Working with Dynamic Shapes
Building engines that handle variable input dimensions at runtime.

Accuracy Considerations
Understanding precision trade-offs and mitigating accuracy loss with reduced precision.

Working with Quantized Types
INT8, FP8, and FP4 quantization workflows including PTQ and QAT.

Advanced Features
Strongly typed networks, layer precision control, and custom plugins.

Chapter 6. Python API Documentation

Attention

- The TensorRT Python API enables developers in Python based development environments and those looking to experiment with TensorRT to easily parse models (such as from ONNX) and generate and run PLAN files.
- To view API changes between releases, refer to the TensorRT GitHub repository and use the compare tool.

This section illustrates the 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
```

6.1. 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`. It can be used as follows:

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

Alternatively, it is possible to define your 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)
```

Building engines is intended as an offline process, so it can take significant time. The *Reducing Engine Build Time* section has tips on making the builder run faster.

6.1.1. 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 definition options are specified using a combination of flags OR-d together.

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

Finally, create a network:

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

6.1.1.1 Creating a Network Definition from Scratch (Advanced)

Instead of using a parser, you can define the network directly to TensorRT through 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 code corresponding to this section can be found in network_api_pytorch_mnist.

This example creates a simple network with Input, Convolution, Pooling, MatrixMultiply, Shuffle, Activation, and Softmax layers. This example uses a helper class to hold some of the 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)
network = builder.create_network(0)
```

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()
```

(continues on next page)

(continued from previous page)

```
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 matrix multiplication:

```
# Get the output from the previous pooling layer
pool2_output = pool2.get_output(0)
batch = pool2_output.shape[0]
mm_inputs = np.prod(pool2_output.shape[1:])
input_reshape = network.add_shuffle(pool2_output)
input_reshape.reshape_dims = trt.Dims2(batch, mm_inputs)
```

Now, add a MatrixMultiply layer. The model exporter provided transposed weights, so the kTRANSPOSE option is specified.

```
fc1_w = weights['fc1.weight'].numpy()
fc1_const = network.add_constant(trt.Dims2(nbOutputs, mm_inputs), fc1_w)
mm = network.add_matrix_multiply(input_reshape.get_output(0), trt.
↪MatrixOperation.NONE,
                                 fc1_const.get_output(0), trt.MatrixOperation.
↪TRANSPOSE)
```

Add bias, which will broadcast across the batch dimension:

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

Add the ReLU activation layer:

```
relu1 = network.add_activation(input=bias_add.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. For instructions on how to build an engine and run an inference with this network, refer to the *Building an Engine* and *Performing Inference* sections.

6.1.2. 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
```

6.1.2.1 Importing a Model Using the ONNX Parser with Custom Weights

The ONNX parser API allows users to provide their own weights, also known as ONNX initializers, in host memory to override any weights found in the model. Instead of parsing the model immediately, the following APIs can be used to load custom weights:

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

Load the model into the parser.

```
status = parser.load_model_proto(model)
assert status
```

Provide the name, pointer to data, and size of the data for the parser to use instead of the one found in the model. This command can be called multiple times to override multiple weights. Note that these pointers must remain in scope until the parser is destroyed.

```
status = parser.load_initializer(name, data, dataSize)
assert status
```

Begin parsing with user-defined weights.

```
status = parser.parse_model_proto()
assert status
```

6.1.3. 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 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, TensorRT might 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, such as 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 can be useful to save the engine to a file for future use. You can do that as follows:

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

Note

Serialized engines are not cross-platform portable. They are specific to the exact GPU model that they were built for (in addition to the platform).

6.2. Deserializing a Plan

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

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

An engine can be deserialized using the following methods:[1]

In-memory Deserialization

This method is straightforward and suitable for smaller models or when memory is not a constraint. Read the plan file into a memory buffer.

[1] The Python API also supports `IStreamReader` for deserialization, which is now deprecated.

```
with open("model.plan", "rb") as f:
    model_data = f.read()
engine = runtime.deserialize_cuda_engine(model_data)
```

You can deserialize the engine from a serialized engine object:

```
serialized_engine = builder.build_serialized_network(network, config)
engine = runtime.deserialize_cuda_engine(serialized_engine)
```

IStreamReaderV2 Deserialization

This is the most advanced method, supporting reading to both host and device pointers and enabling potential performance improvements. With this approach, reading the entire plan file into a buffer to be deserialized is unnecessary, as `IStreamReaderV2` allows reading the file in chunks as needed.

```
class MyStreamReaderV2(trt.IStreamReaderV2):
    def __init__(self, bytes):
        trt.IStreamReaderV2.__init__(self)
        self.bytes = bytes
        self.len = len(bytes)
        self.index = 0

    def read(self, size, cudaStreamPtr):
        assert self.index + size <= self.len
        data = self.bytes[self.index:self.index + size]
        self.index += size
        return data

    def seek(self, offset, where):
        if where == SeekPosition.SET:
            self.index = offset
        elif where == SeekPosition.CUR:
            self.index += offset
        elif where == SeekPosition.END:
            self.index = self.len - offset
        else:
            raise ValueError(f"Invalid seek position: {where}")

with open("model.plan", "rb") as f:
    plan_data = f.read()
reader_v2 = MyStreamReaderV2(plan_data)
engine = runtime.deserialize_cuda_engine(reader_v2)
```

The `trt.IStreamReaderV2` method is particularly beneficial for large models or when using advanced features like GPUDirect or weight streaming. It can significantly reduce engine load time and memory usage.

When choosing a deserialization method, consider your specific requirements:

- For small models or simple use cases, in-memory deserialization is often sufficient.
- For large models or when memory efficiency is crucial, consider using `trt.IStreamReaderV2`.
- If you need custom file handling or streaming capabilities, `trt.IStreamReaderV2` provides the necessary flexibility.

6.3. Performing Inference

The engine holds the optimized model, but inference requires an additional state for intermediate activations. This is done through 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.

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, loops, and synchronous plugins.

First, create the CUDA stream. If you already have one, use a pointer to it, such as for PyTorch CUDA streams, `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(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`. To `synchronize()` with streams created with CUDA Python binding, issue `cudaStreamSynchronize(stream)`.

6.4. Complete End-to-End Example

The snippets above introduce each Python API class in isolation. The script below stitches them into a single copy-paste-ready program that goes from an ONNX file to a benchmarked engine to a live inference call. Save it as `trt_end_to_end.py` and run it against any ONNX model - for example, the ResNet-50 v1 model referenced earlier in this section.

Listing 6.1: trt_end_to_end.py — build an engine from ONNX and run inference

```
import sys
import numpy as np

import tensorrt as trt
import pycuda.driver as cuda
```

(continues on next page)

(continued from previous page)

```
import pycuda.autoinit  # creates a CUDA context for the current thread


def build_engine(onnx_path: str, engine_path: str) -> bytes:
    """Build a strongly typed TensorRT engine from an ONNX file and save it."""
    logger = trt.Logger(trt.Logger.WARNING)
    builder = trt.Builder(logger)
    network = builder.create_network(
        1 << int(trt.NetworkDefinitionCreationFlag.STRONGLY_TYPED)
    )
    parser = trt.OnnxParser(network, logger)
    if not parser.parse_from_file(onnx_path):
        for i in range(parser.num_errors):
            print(parser.get_error(i))
        raise RuntimeError(f"Failed to parse {onnx_path}")

    config = builder.create_builder_config()
    config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 30)  # 1
↪GiB

    serialized = builder.build_serialized_network(network, config)
    if serialized is None:
        raise RuntimeError("Engine build failed")
    with open(engine_path, "wb") as f:
        f.write(serialized)
    return bytes(serialized)


def run_inference(engine_bytes: bytes, input_array: np.ndarray) -> np.ndarray:
    """Deserialize the engine and run a single inference on the supplied input.
↪"""
    logger = trt.Logger(trt.Logger.WARNING)
    runtime = trt.Runtime(logger)
    engine = runtime.deserialize_cuda_engine(engine_bytes)
    context = engine.create_execution_context()

    input_name = engine.get_tensor_name(0)
    output_name = engine.get_tensor_name(1)
    context.set_input_shape(input_name, input_array.shape)

    output_shape = tuple(context.get_tensor_shape(output_name))
    host_input = np.ascontiguousarray(input_array, dtype=np.float32)
    host_output = np.empty(output_shape, dtype=np.float32)

    d_input = cuda.mem_alloc(host_input.nbytes)
    d_output = cuda.mem_alloc(host_output.nbytes)
    stream = cuda.Stream()

    cuda.memcpy_htod_async(d_input, host_input, stream)
    context.set_tensor_address(input_name, int(d_input))
    context.set_tensor_address(output_name, int(d_output))
    context.execute_async_v3(stream.handle)
```

(continues on next page)

(continued from previous page)

```
    cuda.memcpy_dtoh_async(host_output, d_output, stream)
    stream.synchronize()
    return host_output


if __name__ == "__main__":
    onnx_path = sys.argv[1] if len(sys.argv) > 1 else "resnet50-v1-12.onnx"
    engine_path = "model.engine"
    engine_bytes = build_engine(onnx_path, engine_path)

    # Replace this with a preprocessed image batch in your own application.
    dummy_input = np.random.rand(1, 3, 224, 224).astype(np.float32)
    output = run_inference(engine_bytes, dummy_input)
    print("Output shape:", output.shape)
    print("Top-1 class index:", int(np.argmax(output[0])))
```

Run the script with the ONNX file you want to benchmark; on first invocation it builds and serializes `model.engine`, then runs one inference on a random tensor sized for ResNet-50.

```
python3 trt_end_to_end.py resnet50-v1-12.onnx
```

Once you have a working baseline, swap `dummy_input` for a preprocessed image, plug the script into your application, and use the focused sections above when you need finer control over network construction, custom weights, alternate deserialization paths, or asynchronous inference scheduling.

6.5. Next Steps

See also

Optimizing Performance
: Benchmarking methodology and best practices for maximizing inference throughput and latency.

Working with Dynamic Shapes
: Building engines that handle variable input dimensions at runtime.

Accuracy Considerations
: Understanding precision trade-offs and mitigating accuracy loss with reduced precision.

Working with Quantized Types
: INT8, FP8, and FP4 quantization workflows including PTQ and QAT.

Advanced Features
: Strongly typed networks, layer precision control, and custom plugins.

Chapter 7. Advanced Topics

This section covers advanced TensorRT features and configuration options.

See also

Architecture Overview
Foundational context for the TensorRT developer experience - complementary GPU/software ecosystem (MIG, Triton, DALI, Model Optimizer, Polygraphy, Nsight tools), ONNX import, API versioning, and the deprecation policy.

7.1. Engine Compatibility

This page covers engine compatibility across TensorRT versions and hardware targets, including version compatibility, hardware compatibility, compatibility checks, and cross-platform compatibility.

7.1.1. Version Compatibility

By default, TensorRT engines are compatible only with the version of TensorRT used to build them. With appropriate build-time configuration, you can build engines that are compatible with later TensorRT versions.

For example, engines built with TensorRT 8 work with TensorRT 9 and TensorRT 10 runtimes. However, engines built with newer versions (TensorRT 9 or 10) do not work with older runtimes (TensorRT 8).

Note that version-compatible engines can be slower than engines built for a specific runtime version.

Version compatibility is supported from version 8.6; 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 used to build it 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, if you want to use TensorRT 8 or TensorRT 9 engines with TensorRT 10, you must migrate away from removed APIs. We recommend avoiding 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)
```

The request for a version-compatible engine causes a copy of the lean runtime to be added to the plan. When you deserialize the plan, TensorRT will recognize that it contains a runtime copy. It loads the runtime to deserialize and execute the rest of the plan.

> **Warning**
>
> Because version-compatible plans contain executable code that runs 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. For more information, refer to the *Plugin Shared Libraries* section.

7.1.1.1 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 many 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 that it was built with. 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.

7.1.1.2 Loading from Storage

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

7.1.1.3 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 can require plugins 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 can need to be included (either serialized with the engine or provided externally and explicitly loaded).

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 the *Plugin Shared Libraries* section for instructions on using the resulting library list to serialize the plugins or package them externally.

7.1.2. Hardware Compatibility

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

7.1.2.1 NVIDIA Ampere GPU Architecture (and Later) Compatibility Level

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. As a result, a hardware-compatible engine can 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.

7.1.2.2 Same Compute Capability Compatibility Level

TensorRT now supports a new `kSAME_COMPUTE_CAPABILITY` hardware compatibility level. This option allows you to build engines that are compatible with GPUs having the same Compute Capability as the GPU that the engine was built on.

To configure an engine for compatibility with GPUs of the same compute capability, use the following code:

```
config->
↪setHardwareCompatibilityLevel(nvinfer1::HardwareCompatibilityLevel::kSAME_
↪COMPUTE_CAPABILITY);
```

This option ensures the engine is compatible with GPUs that have the same compute capability as the build device. While it can result in lower performance compared to an engine with no compatibility restrictions, it enables broader deployment across devices. Generally, `kSAME_COMPUTE_CAPABILITY` can provide better performance than `kAMPERE_PLUS`. It also allows usage of new hardware features like FP8 and FP4 precision.

7.1.3. Compatibility Checks

TensorRT records the major, minor, patch, and build versions of the library used to create the plan in a plan. If these do not match the runtime version 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 that the plan is being loaded on. 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 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 tactics chosen by the serialization system cannot be optimal for the runtime system and can incur some performance degradation.

If it is impossible to build a TensorRT engine for each 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 4080 and an engine with RTX 4060. At runtime, you can use the RTX 4080 engine on an RTX 4090 GPU and the 4060 engine on an RTX 4070 GPU. In most cases, the engine will run without functional issues and with only a small performance drop compared to running the engine built with the same GPU.

However, deserialization can only succeed if the engine requires a large amount of device memory and the memory available is smaller than when the engine was built. In this case, it is recommended to build the engine on a smaller GPU or on a larger device with limited compute resources.

The safety runtime can deserialize engines generated in an environment where the major, minor, patch, and build versions of TensorRT do not match exactly in some cases. For more information, refer to the NVIDIA DRIVE OS Developer Guide and the NVIDIA TensorRT Safety Production Guide for DriveOS for any safety-related activities.

7.1.4. Cross-Platform Compatibility

By default, TensorRT engines can only be executed on the same platform (operating system and CPU architecture) where they were built. With build-time configuration, engines can be built to be compatible with other types of platforms. For example, to build an engine on Linux x86_64 platforms and expect the engine to run on Windows x86_64 platforms, configure the `IBuilderConfig` as follows:

```
config->setRuntimePlatform(nvinfer1::RuntimePlatform::kWINDOWS_AMD64);
```

The cross-platform engine might have performance differences from the natively built engine on the target platform. It also cannot run on the host platform it was built on.

When building a cross-platform engine that also requires version forward compatibility, `kEXCLUDE_LEAN_RUNTIME` must be set to exclude the target platform lean runtime.

Cross-platform building requires installing a separate Windows builder resource library. Refer to the *TensorRT Installation Guide* for more information.

7.2. 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 goes 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 weight roles.

You can also pass GPU weights to the refitter using:

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

Because of how 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 because supplying only the missing weights does not require more.

After 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 engine behaves like it was built from a network updated with the new weights. After refitting the engine, the previously created execution context can continue to be used.

To view all refittable weights in an engine, use `refitter->getAllWeights(...)`, which is similar to how `getMissingWeights` was used above.

7.2.1. Weight-Stripping

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

This cost is unavoidable when you do not know the refit weights at build time. However, in some scenarios, you know the weights in advance. For example, you might use TensorRT as one of multiple back ends to execute an ONNX model and want to avoid storing 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-fittable 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 for building the weight-stripped engine. For more information, refer to the *trtexec* section.

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. For more information, refer to the *Refitting a Weight-Stripped Engine Directly from ONNX* section.

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 plan file and distribute it to the installer.
3. On the client side, when you launch the network for the first time, update all the weights in the engine. Since all the weights in the engine plan were removed, use the `getAllWeights` API.

   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. Note that when serializing a weight-stripping engine without `kEXCLUDE_WEIGHTS` flag, the resulting serialized engine is not refittable by default. Setting the `kINCLUDE_REFIT` flag can ensure that the serialized engine remains refittable.

7.2.2. 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 is identical to the one used to create the weight-stripped engine. This function will return `true` if all the stripped weights are 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()
```

7.2.3. Weight-Stripping Work with Lean Runtime

You can also use 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)
```

(continues on next page)

(continued from previous page)

```

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.

7.2.4. Fine-Grained Refit Build

When using the kREFIT builder configuration, all weights are marked as refittable. This is useful when you cannot easily distinguish between trainable and untrainable weights. However, marking all weights as refittable can reduce performance because certain optimizations break when weights are marked as refittable.

For example, with the GELU expression, TensorRT can encode all GELU coefficients in a single CUDA kernel. However, if all coefficients are marked as refittable, TensorRT can no longer fuse the Conv-GELU operations into a single kernel.

To address this, TensorRT provides the fine-grained refit API. This API gives you precise control over the weights that 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 to 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.

7.2.5. Stripping Weights with Fine-Grained Refit Build

The fine-grained refit build also works with the weights stripping flag. To run this, we must enable both builder flags in the code after marking the 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 and inference codes are the same as the *Weight-Stripping* sections.

7.3. Precision Control

This page covers how TensorRT selects algorithms and how you control precision. It includes algorithm selection and reproducible builds, strongly typed networks, reduced precision in weakly-typed networks, and control of computational precision.

7.3.1. 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, when implementations have similar timings, system noise can determine the one that is chosen on any particular run of the builder. Different implementations will typically use different orders of accumulation of floating point values, and two implementations can use different algorithms or even run at different

precisions. As a result, different invocations of the builder will typically not result in engines that return bit-identical results.

When the engine is being built for the first time, you supply the `BuilderFlag::kEDITABLE_TIMING_CACHE` flag to TensorRT to enable the editable cache. At the same time, you enable and retain the logs and cache files. The logs will provide the name, key, available tactics, and the selected tactic for each model layer. The cache file will record the decisions made by TensorRT.

Next time the same engine is being built, you supply the same flags to TensorRT and use the interface `ITimingCache::update` to update the cache. Specifically, select tactics for some layers. Then, pass the cache to TensorRT. In the building process, TensorRT will use the newly assigned tactic. Unlike before, in the new version, only one tactic can be assigned to each layer.

7.3.2. 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 with 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 figuring out the layers that must be run at high precision to get the best accuracy can be challenging.

An alternative approach is to specify low-precision use in the model, such as AutoCast and Quantization from Model-Optimizer, 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 that a network is strongly typed, TensorRT infers a type for each intermediate and output tensor using the rules in the operator type specification. TensorRT adheres to these inferred types while building the engine.

Because types are not autotuned, an engine built from a strongly typed network can be slower than one where TensorRT chooses tensor types. However, the build time can improve because fewer kernel alternatives are evaluated.

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`, `kINT8`, `kINT4`, and `kFP4`. The builder flag `kTF32` is permitted as it controls TF32 Tensor Core usage for FP32 types rather than controlling the use of TF32 data types.

7.3.3. Reduced Precision in Weakly-Typed Networks

7.3.3.1 Network-Level Control of Precision

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

Reduced precision support depends on your hardware (refer to GPU Architecture and Precision Support in the TensorRT Support Matrix). 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 can 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 can be enabled independently. TensorRT will still choose a higher-precision kernel if it results in a 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, working with INT8 adds additional complexity. For more information, refer to the *Working with Quantized Types* section.

Note that even if the precision flags are enabled, the engine's input/output bindings default to FP32. Refer to the *I/O Formats* section for information on how to set the data types and formats of the input/output bindings.

7.3.3.2 Layer-Level Control of Precision

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

C++

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

Python

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

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

You can also 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 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 will use it and issue 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);
```

Precision constraints are optional. You can query 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, 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 can choose from 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 constraining the implementation 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. If you call `setOutputType()` for a layer that produces a network output, you should generally configure the corresponding network output to have the same type.

7.3.3.3 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 provides better numerical stability than FP16 for models that require an HDR (high dynamic range) for weights or activations.

There is no guarantee that TF32 Tensor Cores are 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`. When set to 0, this environment variable 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 running the engine 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.

7.3.3.4 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.

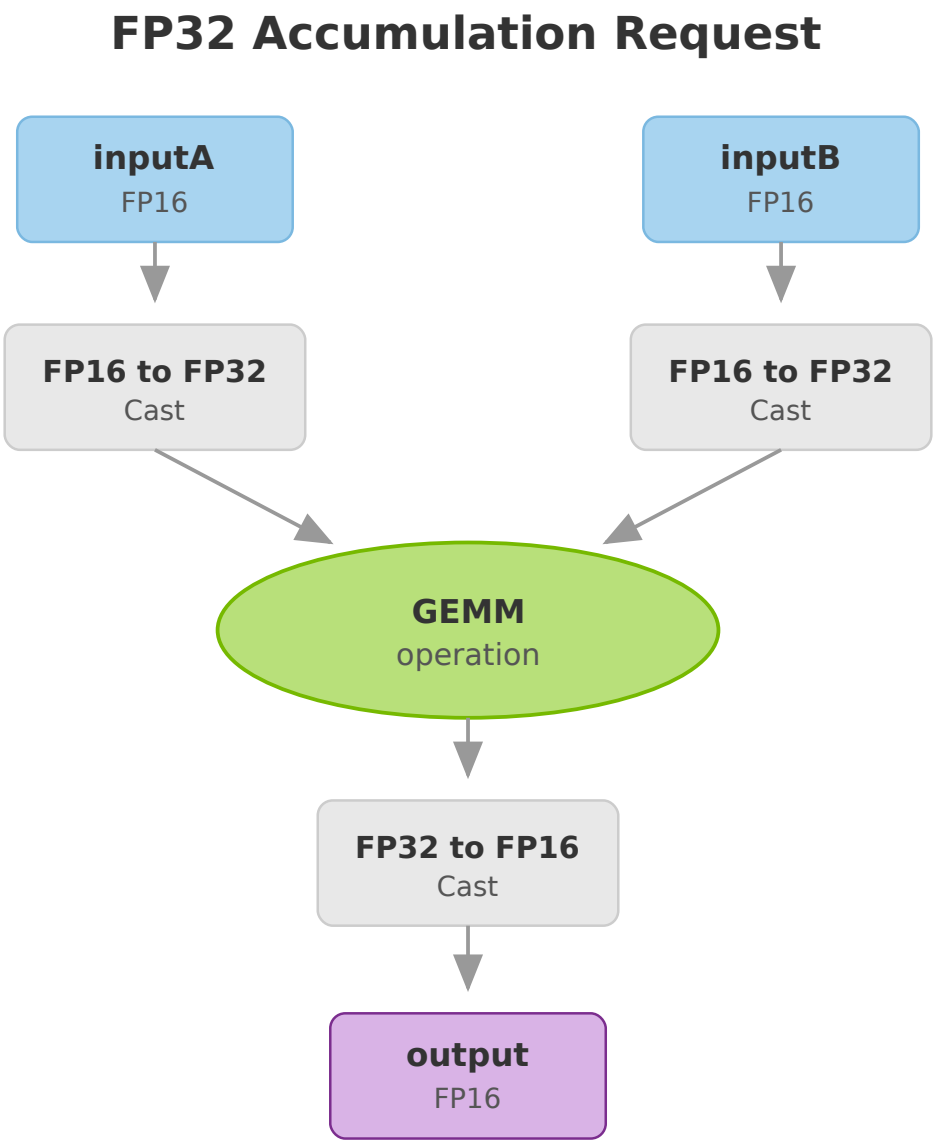
7.3.4. Control of Computational Precision

Sometimes, it is desirable to control the internal precision of the computation in addition to setting the input and output precisions for an operator. TensorRT selects the computational precision by default 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 the 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. However, 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 FP32 precision for FP16 GEMMs by casting the inputs to FP32. TensorRT recognizes this pattern and fuses the casts with the GEMM, resulting in a single kernel with FP16 inputs and FP32 accumulation.



Similarly, accumulation promotion to FP32 can be done for `IAttention` on SMs default to FP16 accumulation with FP16 IOs, by inserting cast-to-FP32 operators for Q/K/V inputs and cast-to-FP16 operators for `IAttention` outputs accordingly.

7.4. Data Formats and Tensors

This page covers I/O data formats at network boundaries, structured sparsity support, empty tensor handling, and input buffer reuse. For a reference of all supported data types and layout formats, refer to *Data Format Descriptions*.

7.4.1. I/O Formats

TensorRT optimizes a network using many different data formats. To allow efficient data passing between TensorRT and a client application, these underlying data formats are exposed at network I/O boundaries, 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 can insert reformats to improve performance.

Note

The DLA-specific tensor formats (`kDLA_LINEAR`, `kDLA_HWC4`) and the rows marked "Only for DLA" in the table below correspond to TensorRT API constants exposed in libnvinfer. DLA execution is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5; these entries are retained as a complete API reference and have no effect on DriveOS targets.

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 as 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.

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

The following table shows the supported formats.

Table 7.1: Supported I/O Formats

Format	kINT32	kFLOAT	kHALF	kINT8	kBOO	kUIN	kINT	BF16	FP8	FP4/INT4
kLINEAR	Only for GPU	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
kCHW2	No	No	Only for GPU	No	No	No	No	Yes	No	No
kCHW4	No	No	No	Yes	No	No	No	No	No	No
kHWC8	No	No	Only for GPU	No	No	No	No	Only for GPU	No	No
kCHW16	No	No	Only for DLA	No	No	No	No	No	No	No
kCHW32	No	Only for GPU	Only for GPU	Yes	No	No	No	No	No	No
kDHWC8	No	No	Only for GPU	No	No	No	No	Only for GPU	No	No
kCDHW32	No	No	Only for GPU	Only for GPU	No	No	No	No	No	No
kHWC	No	Only for GPU	No	No	No	Yes	No	No	No	No
kDLA_LI	No	No	Only for DLA	Only for DLA	No	No	No	No	No	No
kDLA_HW	No	No	Only for DLA	Only for DLA	No	No	No	No	No	No
kHWC16	No	No	Only for NVIDIA Ampere GPUs and later	Only for GPU	No	No	No	No	Only for GPU	No
kDHWC	No	Only for GPU	No	No	No	No	No	No	No	No

Important

For 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 the *Data Format Descriptions* section for how the data are laid out in memory for these formats.

7.4.2. Sparsity

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

- For Convolution, for each output channel and 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 Constant produces an input, every four elements of the reduction axis (K) must have at least two zeros.

Polygraphy (`polygraphy inspect sparsity`) can 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 the Convolution/MatrixMultiply runs in FP16 or INT8 precisions. For example:

C++

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

Python

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

At the end of the TensorRT logs, when the TensorRT engine is built, TensorRT reports the layers that contain weights meeting the structured sparsity requirement and the layers where TensorRT selects tactics that use the structured sparsity. Sometimes, tactics with structured sparsity can be slower than normal, and TensorRT will choose normal tactics. The following output shows an example of TensorRT logs showing information about sparsity:

Note

You need to enable the VERBOSE level log.

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

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

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

7.4.3. Empty Tensors

TensorRT supports empty tensors. A tensor is an empty tensor if it has no elements, that is, has one or more dimensions with a length of 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.

A 0-dimension tensor is a scalar. Since it has one element, it is never empty. Scalars cannot be broadcasted.

7.4.4. Reusing Input Buffers

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

C++

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

Python

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

7.5. 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 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 how values are ordered in memory. 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 always precede these, `C`, `H`, and `W` are typically sorted as `CHW` or `HWC`.

Note

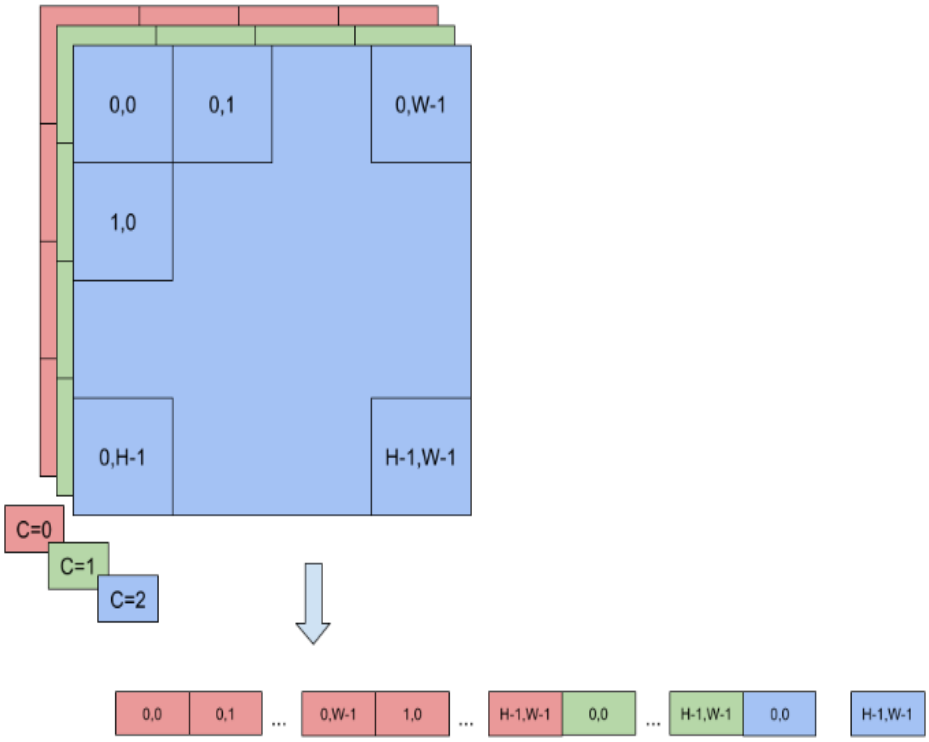
The DLA-specific entries (`kDLA_LINEAR`, `kDLA_HWC4`) and the "DLA FP16" annotation on `kCHW16` correspond to TensorRT API constants exposed in libnvinfer. DLA execution is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5; these entries are retained as a complete API reference and have no effect on DriveOS targets.

Table 7.2: TensorFormat Enum Quick Reference

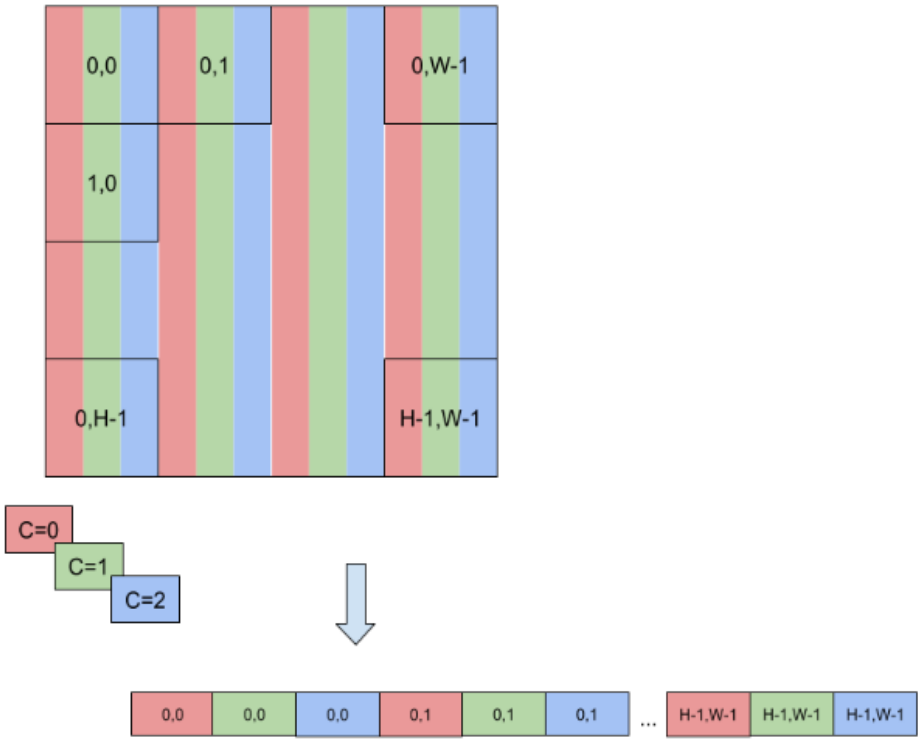
Format Name	TensorFormat Enum	Description
Linear (row-major)	kLINEAR	Default CHW ordering with no vectorization.
NC/2HW2	kCHW2	Channel pairs packed per HxW element (FP16, BF16).
NC/4HW4	kCHW4	4-channel vectors per HxW element (INT8).
NHWC8	kHWC8	8-channel vectors in HWC order (FP16, BF16).
NC/16HW16	kCHW16	16-channel vectors (DLA FP16).
NC/32HW32	kCHW32	32-channel vectors (FP32, FP16, INT8).
NDHWC8	kDHWC8	3D variant of kHWC8 (FP16, BF16).
NC/32DHW32	kCDHW32	3D variant of kCHW32 (FP16, INT8).
NHWC	kHWC	Channel-last without vectorization (FP32, UINT8).
DLA Linear	kDLA_LINEAR	DLA-specific row-major (FP16, INT8).
DLA HWC4	kDLA_HWC4	DLA-specific 4-channel vectors (FP16, INT8).
NHWC16	kHWC16	16-channel vectors in HWC order (FP16, INT8, FP8).
NDHWC	kDHWC	3D channel-last without vectorization (FP32).

For supported data type combinations per format, refer to the *I/O Formats* table.

The default CHW (kLINEAR) layout stores one HxW matrix per channel: all values of channel 0 are stored first, then all of channel 1, and so on.



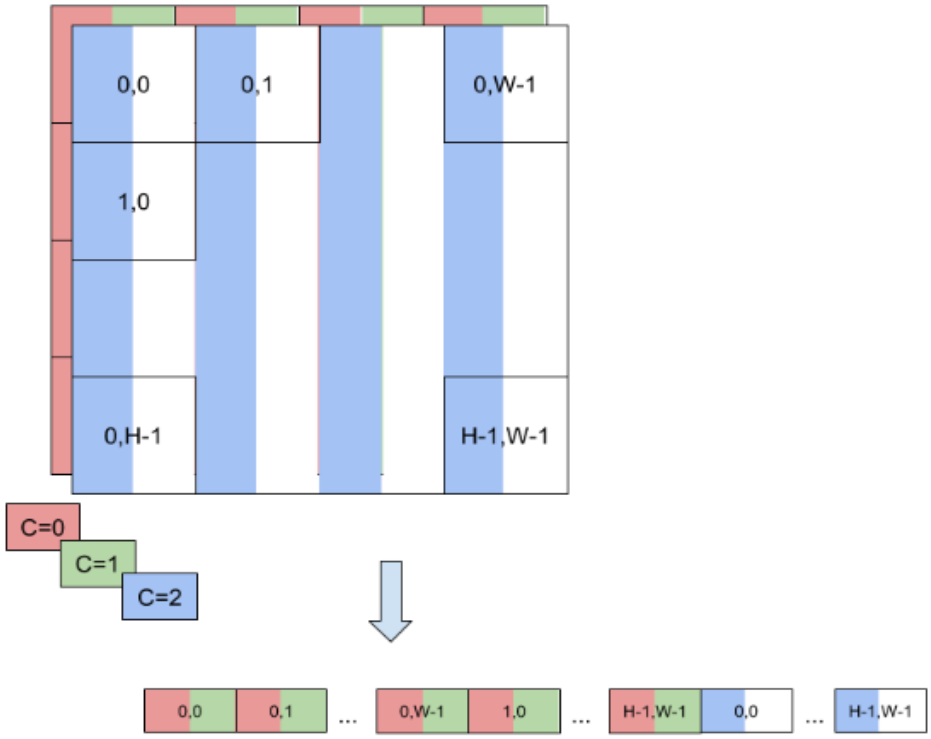
In HWC (kHWC), a single HxW matrix holds the data, and each entry is a C-tuple containing all channel values for that pixel. All channel values for one pixel are stored consecutively before moving to the next pixel.



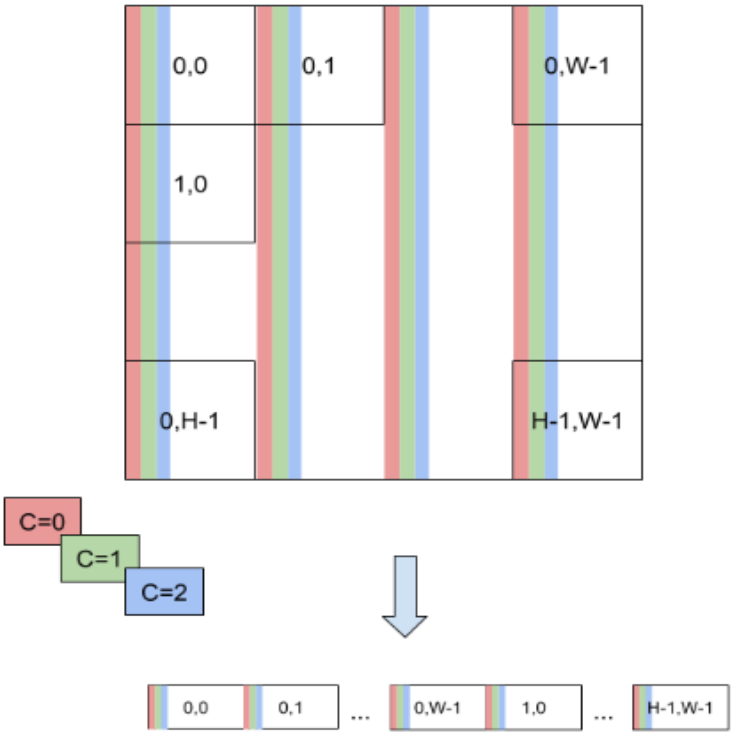
TensorRT also defines formats that pack channel values into fixed-width groups per pixel, which lets

kernels load multiple channels with one vector instruction. NC/2HW2 (kCHW2) and NHWC8 (kHWC8) below are two examples; the table above lists the full set.

In NC/2HW2 (kCHW2), channel values are packed into pairs per pixel; the tensor is stored as `ceil(C/2)` HxW matrices of two-channel pairs. If C is odd, the last pair contains one padded slot. Within a pair, consecutive channels have stride 1; between pairs, the stride is 2*HxW. For example, a 3-channel input is stored as two pair-planes: `[C=0, C=1]` and `[C=2, pad]`.



In NHWC8 (kHWC8), each pixel of the HxW matrix stores its channel values packed into 8-tuples, and C is padded up to a multiple of 8. For example, a 3-channel input has one 8-tuple per pixel containing 3 real values plus 5 padded slots.



Other `TensorFormat` values follow similar packing rules to `kCHW2` and `kHWC8`.

See also

Understanding Formats Printed in Logs
How to interpret tensor format and stride information in TensorRT log output.

Glossary
Definitions of TensorRT terms including tensor formats and precision types.

7.6. Engine Tools and Debugging

This page covers tools for examining and debugging TensorRT engines, including the Engine Inspector API, optimizer callbacks, preview features, and debug tensors.

7.6.1. 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());

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()
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

```
"__mye15868_conv_act_pool_myl0_2"
```

kDETAILED

```
{
    "Name": "__mye15868_conv_act_pool_myl0_2",
    "LayerType": "conv_act_pool",
    "Inputs": [
    {
        "Name": "__mye18111__mye18121_2_myl0",
        "Dimensions": [1,3,224,224],
        "Strides": [150528,50176,224,1],
        "StrideOrder": [3,2,1,0],
        "Datatype": "Int8",
        "Format": "(Linear) Row major linear format"
    }],
    "Constants": [
    {
        "Name": "__mye15805_dconst_myl0",
        "Dimensions": [64,3,7,7],
        "Strides": [147,49,7,1],
        "StrideOrder": [3,2,1,0],
        "Datatype": "Int8",
```

(continues on next page)

(continued from previous page)


```
        "Format": "(Linear) Row major linear format"
    },
    {
        "Name": "__mye15854_dconst_myl0",
        "Dimensions": [1,64,1,1],
        "Strides": [64,1,1,1],
        "StrideOrder": [3,2,1,0],
        "Datatype": "Float",
        "Format": "(Linear) Row major linear format"
    },
    {
        "Name": "__mye15861_dconst_myl0",
        "Dimensions": [1,64,1,1],
        "Strides": [64,1,1,1],
        "StrideOrder": [3,2,1,0],
        "Datatype": "Float",
        "Format": "(Linear) Row major linear format"
    }],
    "Outputs": [
    {
        "Name": "__mye18111_myl0",
        "Dimensions": [1,64,56,56],
        "Strides": [200704,3136,56,1],
        "StrideOrder": [3,2,1,0],
        "Datatype": "Int8",
        "Format": "(NC/32HW32) Thirty-two wide channel vectorized format"
    }],
    "TacticName": "sm80_trt_conv_act_pool_v3_tile_rows_8_tile_cols_120",
    "StreamId": 0,
    "Metadata": "[ONNX Layer: Conv_12]\u001f[ONNX Layer: DequantizeLinear_5]\
↪u001f[ONNX Layer: DequantizeLinear_11]\u001f[ONNX Layer: QuantizeLinear_62]\
↪u001f[ONNX Layer: Relu_14]\u001f[ONNX Layer: QuantizeLinear_8]\u001f[ONNX
↪Layer: MaxPool_15]"
}
```

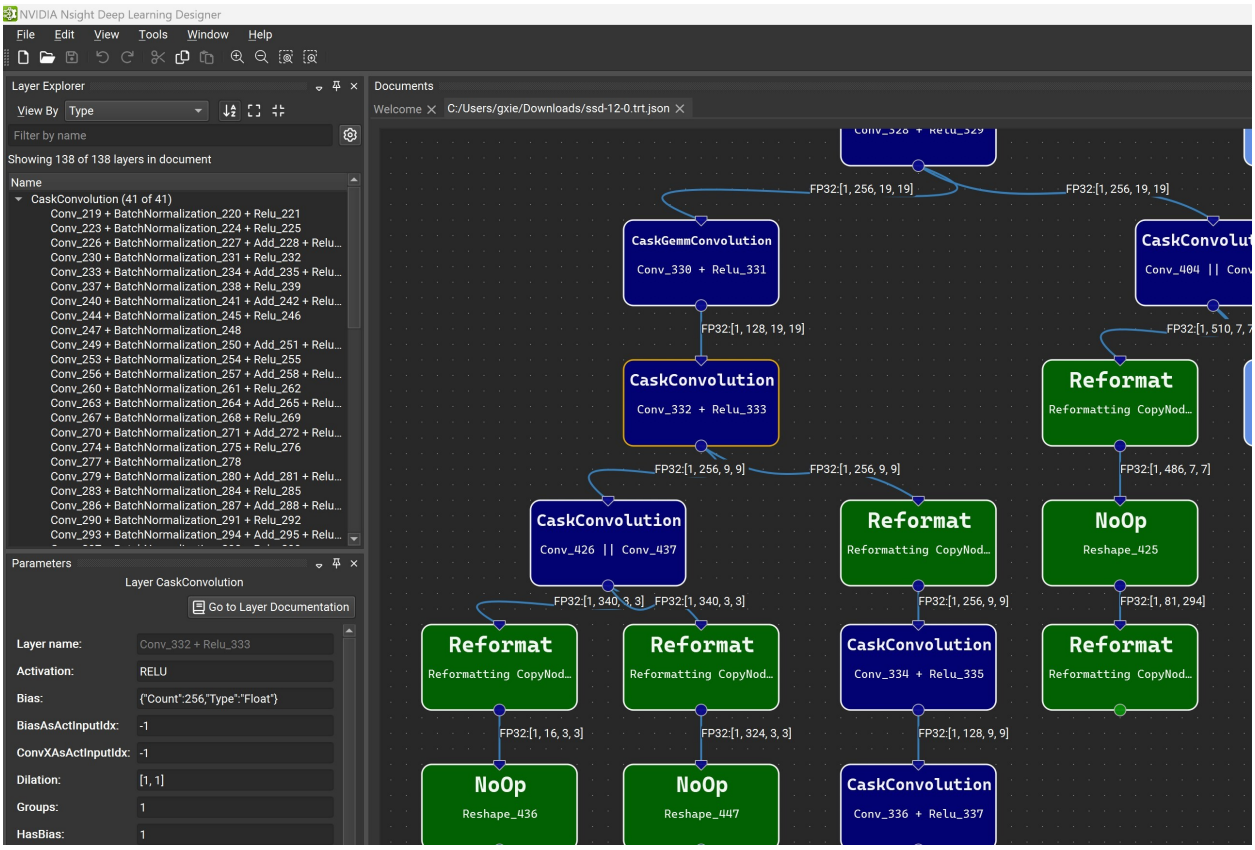

In addition, when the engine is built with dynamic shapes, the dynamic dimensions in the engine information will be shown as -1, and the tensor format information will not be shown because these fields depend on the actual shape at the inference phase.

The `trtexec` tool provides the `--profilingVerbosity`, `--dumpLayerInfo`, and `--exportLayerInfo` flags for getting engine information for a given engine. Refer to the *trtexec* section for more details.

Currently, I/O tensor 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. In future TensorRT versions, more information can be added to the engine inspector output as new keys in the output JSON object. More specifications about the keys and the fields in the inspector output will also be provided.

Engine graph visualization with Nsight Deep Learning Designer

When detailed TensorRT engine layer information is exported to a JSON file with the `--exportLayerInfo` option, the engine's computation graph can be visualized with Nsight Deep Learning Designer. Open the application, and from the **File** menu, select **Open File**, then choose the **.trt.json** file containing the exported metadata.



The Layer Explorer window allows you to search for a particular layer or explore the layers in the network. The Parameter Editor window lets you view the selected layer's metadata.

7.6.2. Optimizer Callbacks

The optimizer callback API feature allows you to monitor the progress of the TensorRT build process, such as 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 several 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.

7.6.3. Preview Features

The preview feature API is an extension of `IBuilderConfig` that allows the gradual introduction of new features to TensorRT. Selected new features are exposed under this API, allowing you to opt in or 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. The feature name and the TensorRT version concatenate feature identifiers.

```
<FEATURE_NAME>_XXYY
```

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

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

Deprecated preview features are marked per 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 `set_preview_feature` and `get_preview_feature` functions.

7.6.4. 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 the 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, at build time, we need to mark tensors as debug tensors, and at runtime, register a DebugListener to process the tensor data and enable debugging.

1. Mark debug tensors prior to network compilation during the build time. There are two methods to mark debug tensors:
   - The first method is marking all unfused tensors as debug tensors.

     C++

     ```
     networkDefinition->markUnfusedTensorsAsDebugTensors();
     ```

Python

```
network.mark_unfused_tensors_as_debug_tensors();
```

This method does not prevent the optimizer from performing fusion and preserves performance. It can remain enabled to help debugging, unless there are concerns about potential data leakage in production.

> **Note**
>
> Some unfused tensors cannot be marked for now.

- The second method is marking the specific tensors as debug tensors.

C++

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

Python

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

This method can disable some fusion optimizations that would otherwise remove the debug tensors, thereby preserving them for issue diagnosis at the expense of reduced performance.

2. At runtime, 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 through the parameters:

```
location: TensorLocation of the tensor. For unfused tensors marked, this
↪will always be kHOST.
addr: pointer to buffer.
type: data type of the tensor.
shape: shape of the tensor
```

(continues on next page)

(continued from previous page)

```
name: name of the tensor. For unfused tensor dump, this name is internal
 ↪and inconsistent with the Engine Inspector. For tensors marked by
 ↪markDebug(), this is the name of tensors in INetwork.
stream: CUDA stream object
```

The data is stored in linear format. 4-bit data, such as kINT4, is stored by packing two elements into one byte.

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

3. After defining the listener, attach it to `IExecutionContext`.

C++

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

Python

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

4. Set the debug state for the tensors of interest before the engine's execution to enable listening.

C++

```
executionContext->setTensorDebugState(tensorName, flag);
executionContext->setUnfusedTensorsDebugState(flag);
executionContext->setAllTensorsDebugState(flag);
```

Python

```
execution_context.set_debug_state(tensorName, flag)
execution_context.unfused_tensors_debug_state = flag
execution_context.set_all_tensors_debug_state(flag)
```

The `trtexec` tool provides several command-line flags that allow users to mark debug tensors and dump their data in various formats such as NumPy, string, or raw data. For more information, refer to the *trtexec* section.

7.7. 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 a 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 host's peak memory usage. You can use `IStreamReaderV2` to deserialize directly from the engine file, avoiding needing a temporary buffer, which helps reduce peak memory usage. `IStreamReaderV2` replaces the existing `IStreamReader` deserialization method.

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.

You can also 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 can be suboptimal and 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.

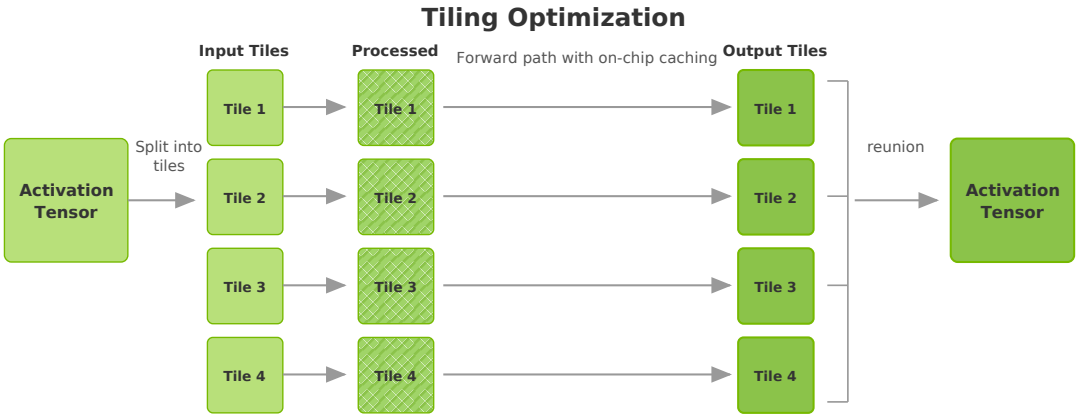
You can query the budget set by `getWeightStreamingBudgetV2()`.

The budget can be adjusted by setting it again when there is no active context for the engine.

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

7.8. Tiling Optimization

Tiling optimization enables cross-kernel tiled inference. This technique leverages on-chip caching for continuous kernels in addition to kernel-level tiling. It can significantly enhance performance on platforms constrained by memory bandwidth.



To activate tiling optimization, perform the following steps:

1. Set the tiling optimization level. Use the following API to specify the duration TensorRT should dedicate to searching for a more effective tiling solution that could improve performance:

   ```
   builderConfig->setTilingOptimizationLevel(level)
   ```

   The optimization level is set to `0` by default, which means TensorRT will not perform any tiling optimization.

   Increasing the level enables TensorRT to explore various strategies and larger search spaces for enhanced performance. However, note that this can significantly increase the engine build time.

2. Configure the L2 cache limit for tiling. Use the following API to provide TensorRT with an estimate of the L2 cache resources that can be allocated for the current engine during runtime:

   ```
   builderConfig->setL2LimitForTiling()
   ```

   This API is a hint to tell TensorRT how much L2 cache resources can be considered dedicated to the current TensorRT engine in the runtime. This will help TensorRT apply a better tiling solution for multiple tasks concurrently running on one GPU. Note that the usage of the L2 cache depends on the workload and heuristic; TensorRT cannot apply this limit for all layers.

   TensorRT manages the default value.

7.9. Multi-Device Inference

The multi-device inference scales TensorRT inference across multiple GPUs. Use it when models exceed single-GPU memory or when parallel execution reduces latency for memory-bound workloads.

To support these multi-device workflows, TensorRT relies on NVIDIA NCCL. For improved environment compatibility and deployment flexibility, TensorRT features automated NCCL library discovery, checking for `libnccl.so.2` before seamlessly falling back to `libnccl.so` using `LD_LIBRARY_PATH`.

Multi-device provides two capabilities:

- **DistCollective** – Distributed collective operations (`AllReduce`, `AllGather`, `Broadcast`, `Reduce`, `ReduceScatter`, `AllToAll`, `Gather`, `Scatter`) using NVIDIA NCCL. Requires Ampere (SM 80) or later.
- **Multi-device attention** – Attention layers with context parallelism that split the key-value sequence across GPUs. BF16 and FP16 only. Requires Blackwell (SM 100) or later.

For more information, refer to the sampleDistCollective and attention_mdtrt samples.

7.9.1. Setup

1. Configure layers for multi-device after adding them to the network.
   - For `DistCollective` layers, set the number of ranks:

     C++

     ```
     collectiveLayer->setNbRanks(numGpus);
     ```

     Python

     ```
     collective_layer.num_ranks = num_gpus
     ```

   - For multi-device attention, set the number of ranks:

     C++

     ```
     attention->setNbRanks(numGpus);
     ```

     Python

     ```
     attention.num_ranks = num_gpus
     ```

2. Initialize a NCCL communicator with `ncclCommInitRank` or `ncclCommInitAll` and set it on the execution context before inference.

   C++

   ```
   context->setCommunicator(ncclComm);
   ```

Python

```
context.set_communicator(nccl_comm)
```

3. Execute inference on all ranks with synchronized enqueueV3 (C++) or `execute_async_v3` (Python) calls.

> **Warning**
>
> All participating ranks must call the execution method concurrently because NCCL collective operations block until every rank has participated. If any rank fails to issue its execution call, the other ranks will hang indefinitely.

C++

```
// Each rank calls enqueueV3 on its own CUDA stream
context->enqueueV3(stream);
```

Python

```
# Each rank calls execute_async_v3 on its own CUDA stream
context.execute_async_v3(stream_ptr)
```

Each rank must load the same engine, allocate its own input/output buffers, and use its own `IExecutionContext` and CUDA stream. Use standard CUDA synchronization (`cudaStreamSynchronize` or CUDA events) to wait for completion on each rank.

7.9.2. Platform Support

Architecture	OS	CUDA
x86_64	▶ Ubuntu 24.04 ▶ Rocky 8	12.9, 13.2
AArch64	Ubuntu 22.04	13.2

- Multi-device inference is not supported in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5. It is also unavailable in other special-purpose builds (RTX, Coverity).
- GPU requirements:
  - `DistCollective`: Ampere (SM 80) and later
  - Multi-device attention: Blackwell (SM 100) and later

7.9.3. Feature Compatibility

Feature	Supported	Notes
DLA	No	
Ragged Tensor	No	
Weight stripped engine	No	
Refittable weights	No	
Weight streaming	Yes	Rank-local: each rank streams its own sharded weights independently.
Safety build	No	
Strongly typed	Yes	
Precisions	Partial	▶ Multi-device attention: BF16 and FP16 only. ▶ `DistCollective`: FP32, FP16, BF16, FP8, INT64, INT32, INT8, UINT8, and BOOL.
Timing cache	Yes	
CUDA graphs	Yes	
Quantization	Yes	

7.10. Set Internal Library Path API

TensorRT internally loads builder resource libraries (`libnvinfer_builder_resource_*.so`) from the system library path (for example, `LD_LIBRARY_PATH`). If you load TensorRT from a custom location using `dlopen`, these resource libraries might not be found automatically.

Use `nvinfer1::setInternalLibraryPath` to specify the directory where TensorRT should look for its internal resource libraries at runtime.

```
// Load TensorRT from a custom location
dlopen("/path/to/custom/libnvinfer.so");
// Tell TensorRT where to find its internal resource libraries
nvinfer1::setInternalLibraryPath("/path/to/custom");
// Create builder and build engine
```

Chapter 8. Working with Quantized Types

TensorRT enables high-performance inference by supporting quantization, a technique that reduces model size and accelerates computation by representing floating-point values with lower-precision data types.

Key Benefits: - Reduces memory footprint - Improves energy efficiency - Enables deployment on resource-constrained edge devices - Achieves greater cost-efficiency in large-scale data center deployments

Quantization Approach: TensorRT uses a symmetric quantization scheme, where both activations and weights are mapped to quantized values centered around zero. This approach simplifies the transformation between quantized and floating-point representations, typically involving only a scaling factor.

Supported Data Types:

- INT8 (signed 8-bit integer)
- INT4 (signed 4-bit integer, weight-only quantization)
- FP8E4M3 (FP8, 8-bit floating point with 4 exponent and 3 mantissa bits)
- FP4E2M1 (FP4, 4-bit floating point with 2 exponent and 1 mantissa bit)

These low-precision formats allow TensorRT to deliver efficient inference while maintaining accuracy, making it suitable for deployment in resource-constrained environments and high-throughput applications.

8.1. Quantization Workflows

TensorRT supports both post-training quantization (PTQ) and quantization-aware training (QAT) workflows. These workflows enable you to optimize models for low-precision data types.

The quantization process uses per-tensor, per-channel, or block-wise scaling. The scaling method depends on the layer and data type, which helps preserve model accuracy during conversion.

Post-Training Quantization (PTQ): - Quantizes a pre-trained model without retraining - Requires representative “calibration data” set to compute quantization parameters - Practical when retraining is infeasible due to resource limitations or data privacy concerns - Can lead to accuracy degradation for complex models or sensitive layers

Quantization-Aware Training (QAT): - Simulates quantization during training by quantizing weights and activation layers - Training actively compensates for quantization and dequantization effects -

Generally achieves superior accuracy recovery compared to PTQ - More time-consuming and requires access to the entire labeled training dataset

The TensorRT Model Optimizer is a Python toolkit designed to help you create QAT models. These models are fully compatible with TensorRT's optimization and deployment workflows.

The toolkit also provides a PTQ recipe that lets you perform PTQ on models developed in both PyTorch and ONNX formats, streamlining the quantization process across different frameworks.

8.1.1. Explicit Quantization Overview

A TensorRT network is explicitly quantized when it contains Quantize and Dequantize layers (Q/DQ for short), which mark exactly where conversions to and from a quantized type occur. The optimizer performs only the conversions dictated by the semantics of the model.

In explicitly quantized networks, the quantization and dequantization operations are represented by `IQuantizeLayer` and `IDequantizeLayer` nodes in the graph - these will henceforth be referred to as Q/DQ nodes. `IDynamicQuantizeLayer` extends this API for Dynamic Quantization workflows where scales are computed at inference time.

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 a Quantize node followed by a Dequantize node. Since TensorRT preserves the semantics of these layers, users can expect accuracy that is very close to that seen in the deep learning framework. While optimizations preserve the arithmetic semantics of quantization and dequantization operators, they can change the order of floating-point operations in the model, so results will not be bitwise identical.

Key Capabilities:

- **Broad Data Type Support**: Supports INT8, FP8, INT4, and FP4 quantized data types.
- **Precise Placement Control**: Q/DQ nodes specify exactly where conversions to and from quantized types occur, so the optimizer applies only the conversions dictated by the model's semantics.
- **Compatibility with ONNX Export**: Performing QAT or PTQ in a deep learning framework and exporting to ONNX naturally produces an explicitly quantized model, because ONNX represents fake-quantization operations as Q/DQ pairs.

Calibration is performed before ONNX export (for example, during PTQ or QAT with Model Optimizer), and the resulting scales are embedded directly into the Q/DQ nodes.

8.1.2. Quantization Granularities

Quantization granularity refers to how quantization scale factors are applied across a model's tensors. Selecting the appropriate granularity is a direct lever for balancing the benefits of quantization (such as memory reduction) against its potential drawbacks (such as accuracy loss). A more granular approach increases potential accuracy but also increases the computational and memory overhead of managing multiple scaling factors. TensorRT supports three quantization scale granularities:

1. *Per-tensor quantization*: a single scale value (scalar) is used to scale the entire tensor.
2. *Per-channel quantization*: a scale tensor is broadcast along the given axis - for convolutional neural networks, this is typically the channel axis.
3. *Block quantization*: the tensor is divided into fixed-size blocks, and a scale factor is defined for each block. `IQuantizeLayer` and `IDequantizeLayer` support 1D block shapes (along a single dimension). For 2D block shapes (along the last two dimensions), use `IDynamicQuantizeLayer`.

When using per-channel quantization with Convolutions, the quantization axis 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 blocks, with all elements in a block sharing a common scale factor. Block quantization is supported for inputs of up to 3 dimensions.

INT4 block quantization supports weight-only quantization (WoQ).

FP4 block quantization supports both weights and activations. To minimize quantization error, use Dynamic Quantization for activations.

When using block quantization, the scale tensor dimensions equal the data tensor dimensions except for one blocking axis (the dimension that blocking is performed over). 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[ceil(r//B), s]))
```

The scale is a 2D array of coefficients with dimensions `(ceil(R//B), S)`.

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

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

8.1.3. Quantized Types Rounding Modes

TensorRT primarily uses the `round-to-nearest-even` method (also known as “banker’s rounding”), which rounds ties to the nearest even value. For example, 2.5 rounds to 2 and 3.5 rounds to 4. This method helps reduce systematic bias in the quantization process, preventing the consistent upward or downward drift that can occur with other rounding strategies.

For more information about rounding modes, refer to Rounding.

8.1.4. Dynamic Quantization

Dynamic Quantization is a form of quantization where the scales are computed during inference according to the input data. It produces two outputs: quantized data and scales. TensorRT supports Dynamic Quantization only with block quantization granularity.

Dynamic Quantization has two main benefits:

1. **Accuracy**: With Dynamic Quantization, TensorRT selects a scale that maps only the dynamic range of a single block to the quantized type. Because the dynamic range of a single block is often much smaller than the dynamic range of the entire tensor, the quantization error is reduced. This is most significant for sub-byte quantized types because of the small range of values representable in these data types.
2. **Reduced PTQ overhead**: Because TensorRT computes the scales automatically during inference, you do not need to calibrate them using sample data.

For each block, the scale is computed by:

$$scale = max_{i \in \{0 \ldots blockSize-1\}} \left(\frac{abs(x_i)}{qTypeMax} \right)$$

Where:

- $qTypeMax$ is the maximum value in the quantized type (such as 6 for FP4E2M1).

8.1.4.1 MX-Compliant Dynamic Quantization

Dynamic Quantization according to the OCP Microscaling Formats (MX) Specification v1.0. The MX-compliant recipe performs block quantization, quantizing across 32 high-precision elements to produce 32 quantized output values and one E8M0 scaling factor.

TensorRT currently supports MX-compliant Dynamic Quantization only for the FP8E4M3 vector format, referred to as MXFP8.

The scale computation for a single block is defined as:

$$scale_{E8M0} = round\_up\_to\_e8m0 \left(max_{i \in \{0 \ldots blockSize-1\}} \left(\frac{abs(x_i)}{qTypeMax} \right) \right)$$

Where:

- $E8M0$ is an 8-bit exponent-only floating point type, as described in *Supported Types*.
- $round\_up\_to\_e8m0$ is the computed scale rounded up to the smallest power of two that is greater than or equal to it.
- $qTypeMax$ is the maximum value representable in the quantized type used for data.

The scale computation is repeated for each block, computing a total of $\frac{inputVolume}{blockSize}$ block scales.

8.1.4.2 Dynamic Double Quantization

A variant of Dynamic Quantization, in which the computed scales are also quantized. Putting together the scale computation and scale quantization for a single block:

$$scale_{quantized} = quantize \left(max_{i \in \{0 \ldots blockSize-1\}} \left(\frac{abs(x_i)}{qTypeMax} \right), scale = globalSf \right)$$

Where:

- $globalSf$ is an offline-calibrated per-tensor quantization scale (scalar).

TensorRT currently supports Dynamic Double Quantization only for the NVFP4 vector format (FP4E2M1 data, FP8E4M3 scales, block size of 16).

Using $qTypeMax = 6$ and the FP8 range of `[-448, 448]`, the quantized scale can be written as:

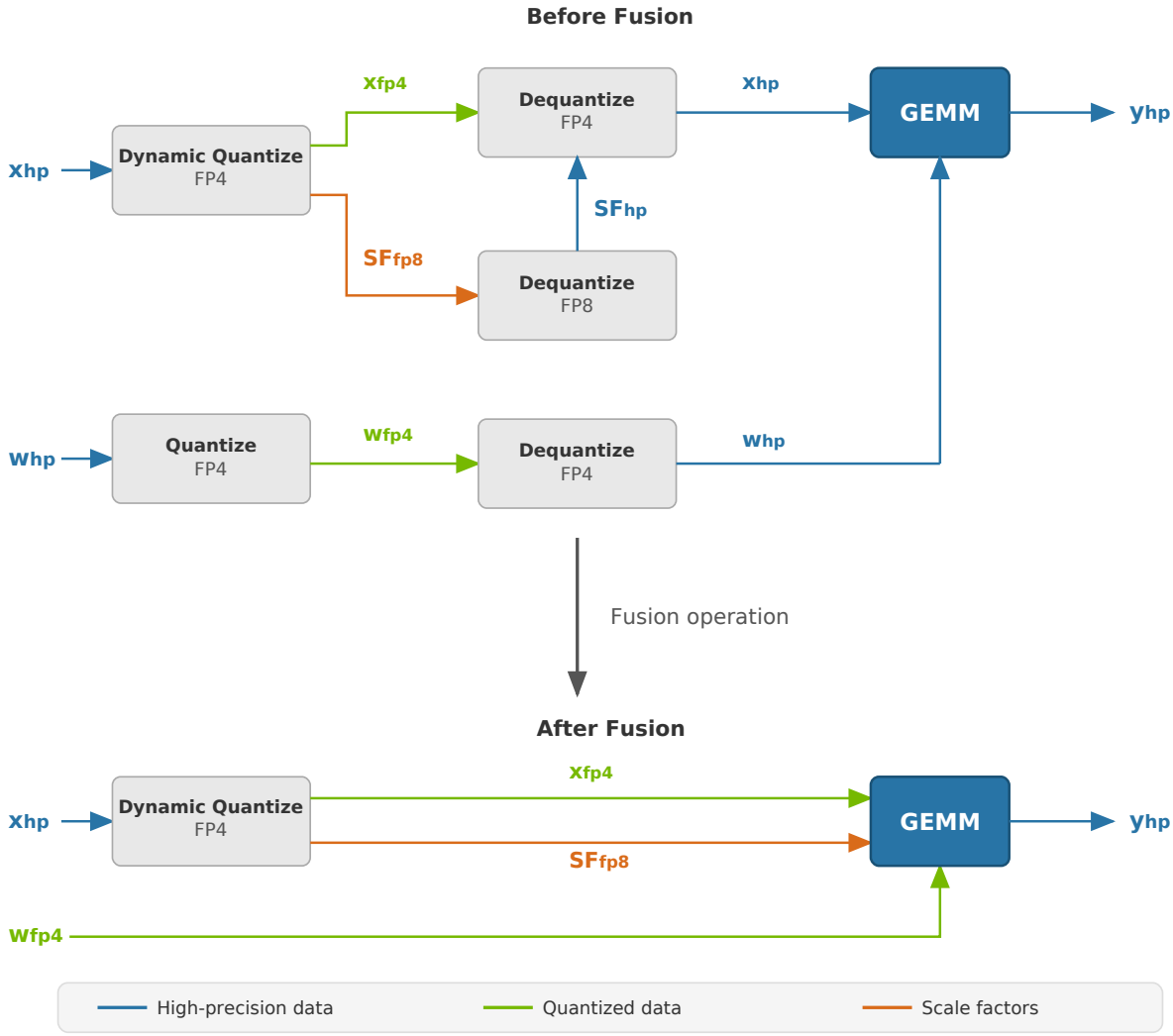
$$scale_{fp8} = castToFp8 \left(\frac{max_{i \in \{0 \ldots blockSize-1\}} \left(abs(x_i) \right)}{6 * globalSf} \right)$$

The quantized data is computed using block quantization and the computed scales.

To dequantize data that was quantized using Dynamic Double Quantization, two consecutive Dequantize operations are required (which is why it is called *double* quantization): the first dequantizes the scales using per-tensor quantization, and the second dequantizes the data.

$data_{DQ} = dequantize\left(data_Q, dequantize\left(scale_Q, scale = globalSf\right)\right)$

Dynamic Double Quantization Fusion with GEMM



8.2. Quantization Schemes

INT8

INT8 quantization and dequantization operations are defined as follows:

$x_q = quantize\left(x, s\right) = roundWithTiesToEven(clip(\frac{x}{s}, -128, 127))$

$x = dequantize\left(x_q, s\right) = x_q * s$

Where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized INT8 value in range `[-128, 127]`.
- s is the quantization scale expressed using a 16-bit or 32-bit floating point scalar.
- $roundWithTiesToEven$. Refer to Rounding.

FP8

FP8 quantization and dequantization operations are defined as follows:

$x_q = quantize\,(x, s) = castToFp8(clip(\frac{x}{s}, -448, 448))$

$x = dequantize\,(x_q, s) = x_q * s$

Where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized FP8E4M3 value in the range `[-448, 448]`.
- s is the quantization scale expressed using a 16-bit or 32-bit floating point scalar.
- $castToFp8$ rounds to the nearest value representable in FP8E4M3, ties are rounded to an even number. Refer to Rounding.

MXFP8

MXFP8 is a dynamic per-block quantization scheme. The output type is FP8E4M3, the scale type is E8M0, and the block size is 32. The quantization and dequantization formulas are identical to the FP8 quantization scheme.

When quantizing activations, *Dynamic Quantization* is required.

INT4

INT4 quantization and dequantization operations are defined as follows:

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

$x = dequantize\,(x_q, s) = x_q * s$

Where:

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

INT4 quantization requires per-block scales. The supported block sizes are `{64, 128}`. The block dimension should be one of the last two dimensions.

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

NVFP4

NVFP4 quantization requires per-block scales. The only supported block size is 16. The block dimension should be one of the last two dimensions.

$x_q = quantize\,(x, s) = castToFp4(clip(\frac{x}{s}, -6, 6))$

$x = dequantize\,(x_q, s) = x_q * s$

Where:

- x is a high-precision floating point value to be quantized.
- x_q is a quantized FP4 value in the range `[-6, 6]`.
- s is the block's quantization scale expressed using a 16-bit or 32-bit floating point.
- $castToFp4$ rounds to the nearest value representable in FP4E2M1, ties are rounded to an even number. Refer to Rounding.

When quantizing activations, *Dynamic Quantization* is required.

Table 8.1: Quantization Schemes and Precision

Quantization Schemes	INT8	FP8	MXFP8	INT4	NVFP4
Representation	8-bit signed 2's complement	S1E4M3 floating point	▶ S1E4M3 floating point	4-bit signed 2's complement	S1E2M1 floating point
Weight quantization	Per-tensor/per-axis	Per-tensor/per-axis	Per-block (block size = 32)	Per-block (block sizes = {64, 128})	Per-block
Activation quantization	Per-tensor	Per-tensor	Dynamic, per-block (block size = 32)	No	Dynamic, per-block
Explicit quantization	Yes	Yes	Yes	Yes	Yes
Scale data type	FP32, FP16, BF16	FP32, FP16, BF16	E8M0	FP32, FP16, BF16	FP32, FP16, BF16

8.3. Explicit Quantization

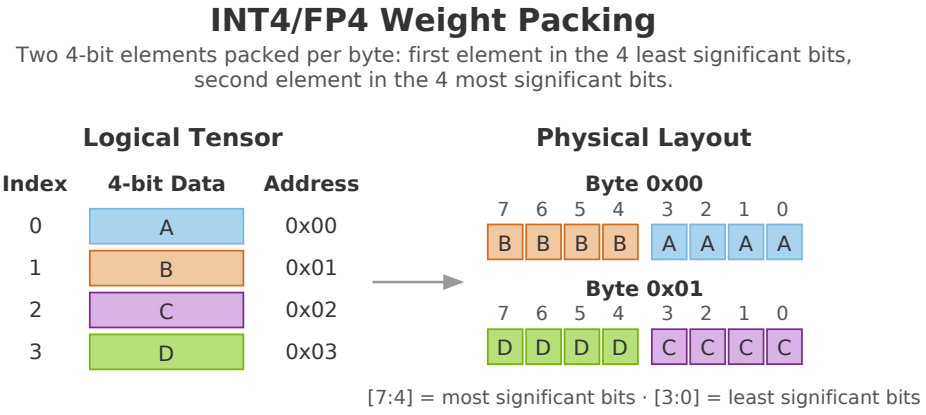
When TensorRT detects Q/DQ layers in a network, it builds an engine using explicit quantization processing logic. The rest of this section describes how explicit quantization operates in more detail.

Use explicit quantization with Strong Typing. Precision-control build flags are not required and should not be specified.

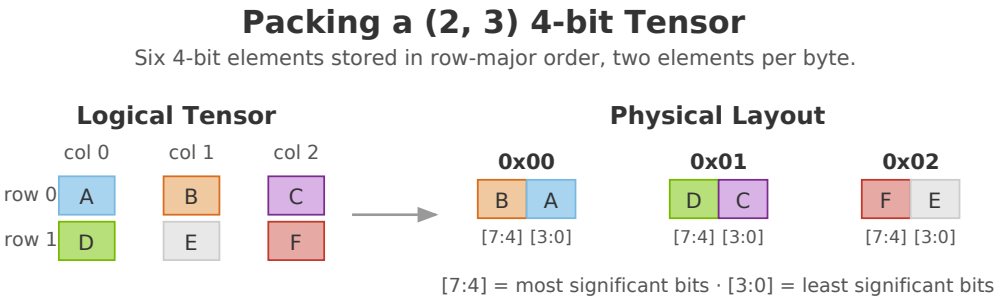
8.3.1. Quantized Weights

You can specify weights of Q/DQ models using a high-precision data type (FP32, FP16, or BF16) or a low-precision quantized type (INT8, FP8, INT4, or FP4). When TensorRT builds an engine, it quantizes high-precision weights using the `IQuantizeLayer` scale, which operates on the weights. The quantized (low-precision) weights are stored in the engine plan file. When using pre-quantized weights, an `IDequantizeLayer` is required between the weights and the linear operator that uses them.

INT4 and FP4 quantized weights are stored by packing two elements per byte. The first element is stored in the 4 least significant bits, and the second is stored in the 4 most significant bits.



The following example illustrates this packing for a (2, 3) tensor.



8.3.2. 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 an ONNX model, 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` and `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). Many PyTorch releases do not export native FP8 Q/DQ operators to ONNX opset 19. 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 can parse both the custom operators and standard opset 19 Q/DQ operators. Note that TensorRT does not fully support opset 19, and other tools such as ONNX Runtime cannot parse the custom operators. ONNX opset 21 added support for the INT4 data type and block quantization, and ONNX opset 23 added support for the FP4E2M1 type.

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 the `transB` GEMM attribute enabled, which transposes the weights before performing the GEMM operation. TensorFlow, on the other hand, pre-transposes the weights `(C, K)` before ONNX export:

- PyTorch: $y = xW^T$
- TensorFlow: $y = xW$

TensorRT, therefore, transposes PyTorch weights. TensorRT quantizes the weights 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`).

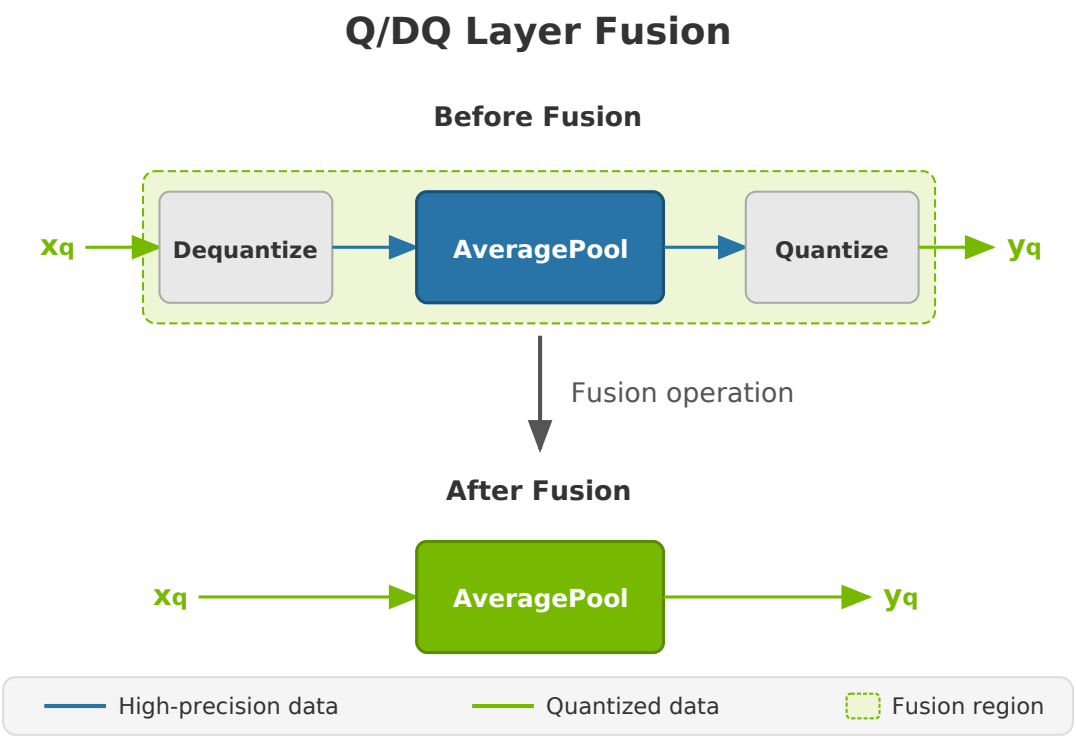
TensorRT does not support pre-quantized ONNX models that use INT8 or FP8 quantized operators. Specifically, the following ONNX quantized operators are *not* supported and generate an import error when TensorRT encounters them while importing the ONNX model:

- QLinearConv and QLinearMatmul
- ConvInteger and MatmulInteger

8.3.3. 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 because 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 the same 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, and an `IDequantizeLayer` instance converts a quantized tensor back to a high-precision floating-point tensor. TensorRT expects a Q/DQ layer pair on each input 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 operate on quantized data using compute operations suitable for quantized types.

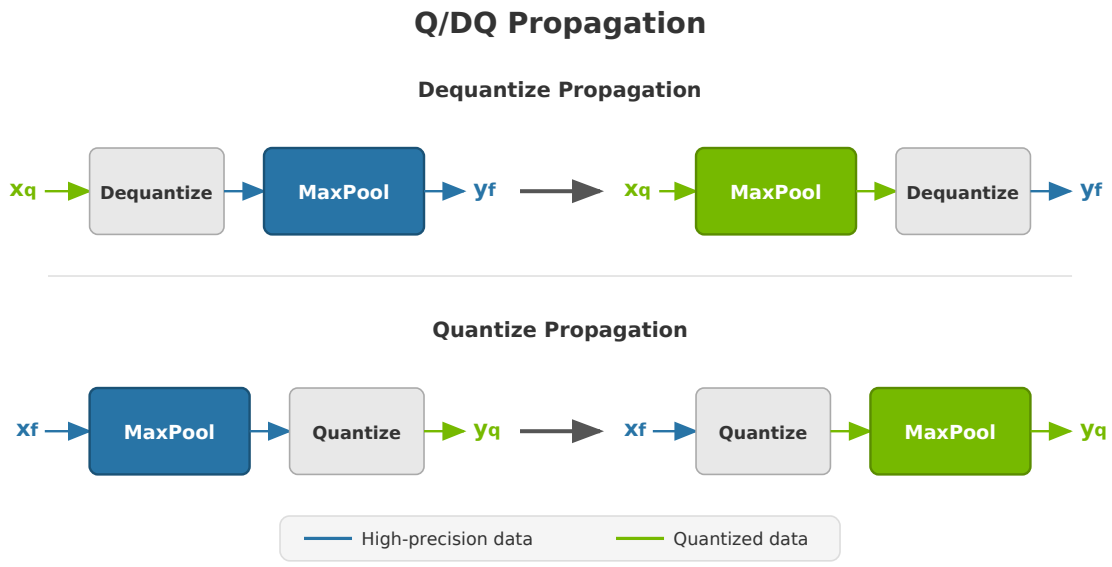


During network optimization, TensorRT moves Q/DQ layers in a process called Q/DQ propagation. The goal of propagation is to maximize the proportion of the graph that can be processed at low precision. To achieve this, TensorRT propagates Quantize nodes backward (so quantization happens as early as possible)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

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

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

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



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 $\{x_0..x_m\}$:

$output_i := max\left(\{x_0, x_1, ...x_m\}\right) = max\left(\{max\left(\{max\left(\{x_0, x_1\}\right), x_2\}\right), ...x_3\}\right)$

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

$x_j = max\left(\{x_j, x_k\}\right) for\ x_j \geq x_k$

For the quantization function $Q\left(a, scale, x_{max}, x_{min}\right) := truncate\left(round\left(\frac{a}{scale}\right), x_{max}, x_{min}\right) scale > 0$, note that (without providing proof and using simplified notation): $Q\left(x_j, scale\right) \geq Q\left(x_k, scale\right) for x_j \geq x_k$

Therefore: $max\left(\{Q\left(x_j, scale\right), Q\left(x_k, scale\right)\}\right) = Q\left(x_j, scale\right) for x_j \geq x_k$

However, by definition: $Q\left(max\left(\{x_j, x_k\}\right), scale\right) = Q\left(x_j, scale\right) for x_j \geq x_k$

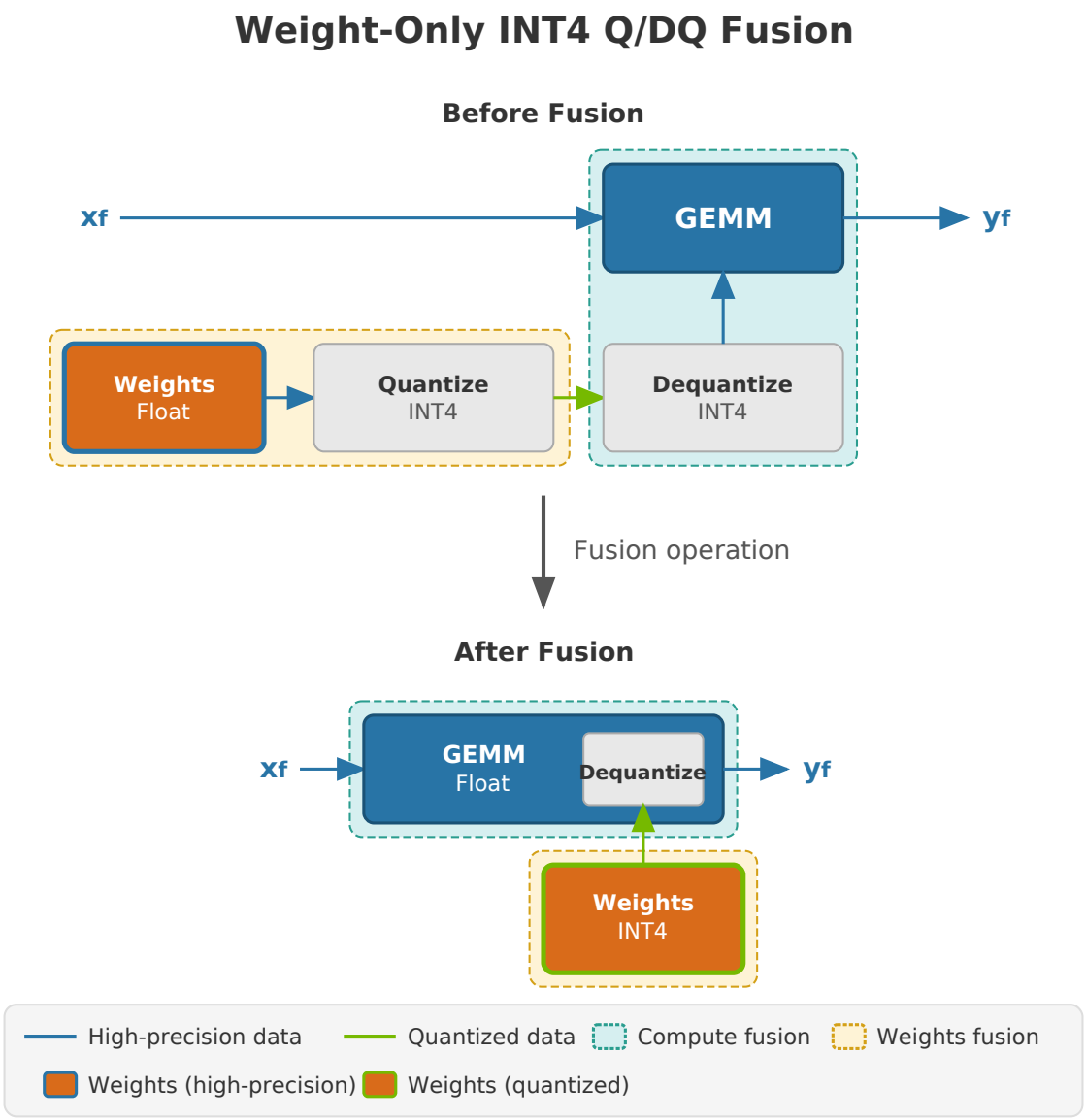
Function max commutes with quantization, and so does Max Pooling.

Similarly, for dequantization, function $DQ\left(a, scale\right) := a * scale$ with $scale > 0$ it can be shown that: $max\left(\{DQ\left(x_j, scale\right), DQ\left(x_k, scale\right)\}\right) = DQ\left(x_j, scale\right) = DQ\left(max\left(\{x_j, x_k\}\right), scale\right) for x_j \geq x_k$

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

8.3.4. Weight-Only Quantization

Weight-only quantization (WoQ) is an optimization 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 before performing the dot product in high precision.



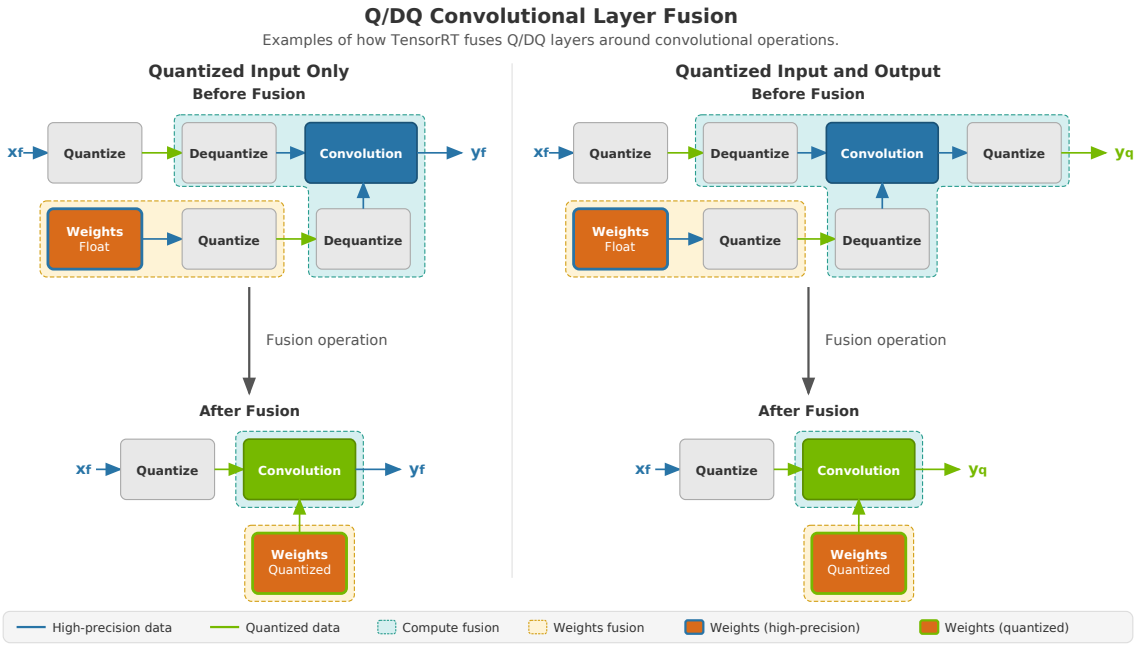
WoQ is available only for INT4 block quantization with GEMM layers. The GEMM data input is specified in high precision (FP32, FP16, or BF16), and the weights are quantized using Quantize and Dequantize layers as usual. TensorRT creates an engine with INT4 weights and a high-precision GEMM operation.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

8.3.5. Q/DQ Layer-Placement Recommendations

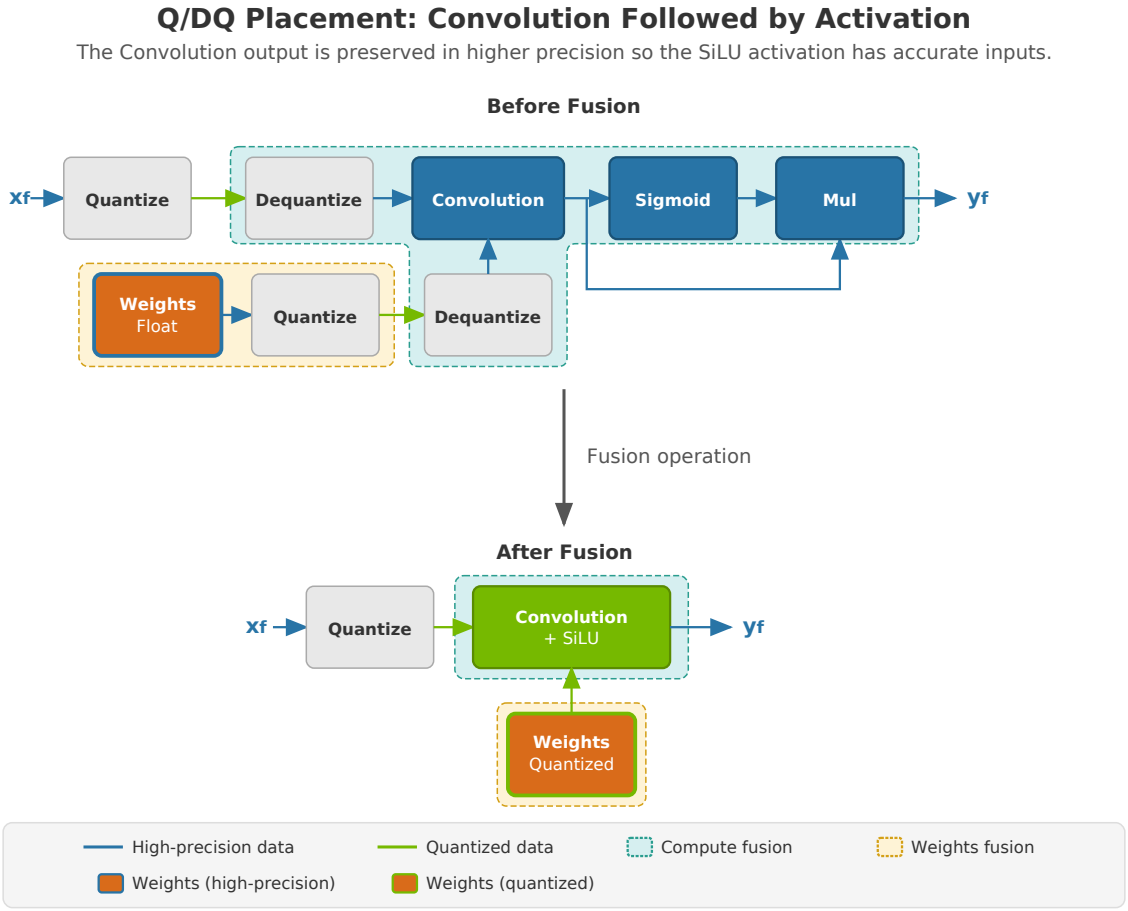
The placement of Q/DQ layers in a network affects performance and accuracy. Aggressive quantization can degrade model accuracy because quantization introduces error, but quantization also reduces latency. The following recommendations help you place Q/DQ layers effectively in your network.

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

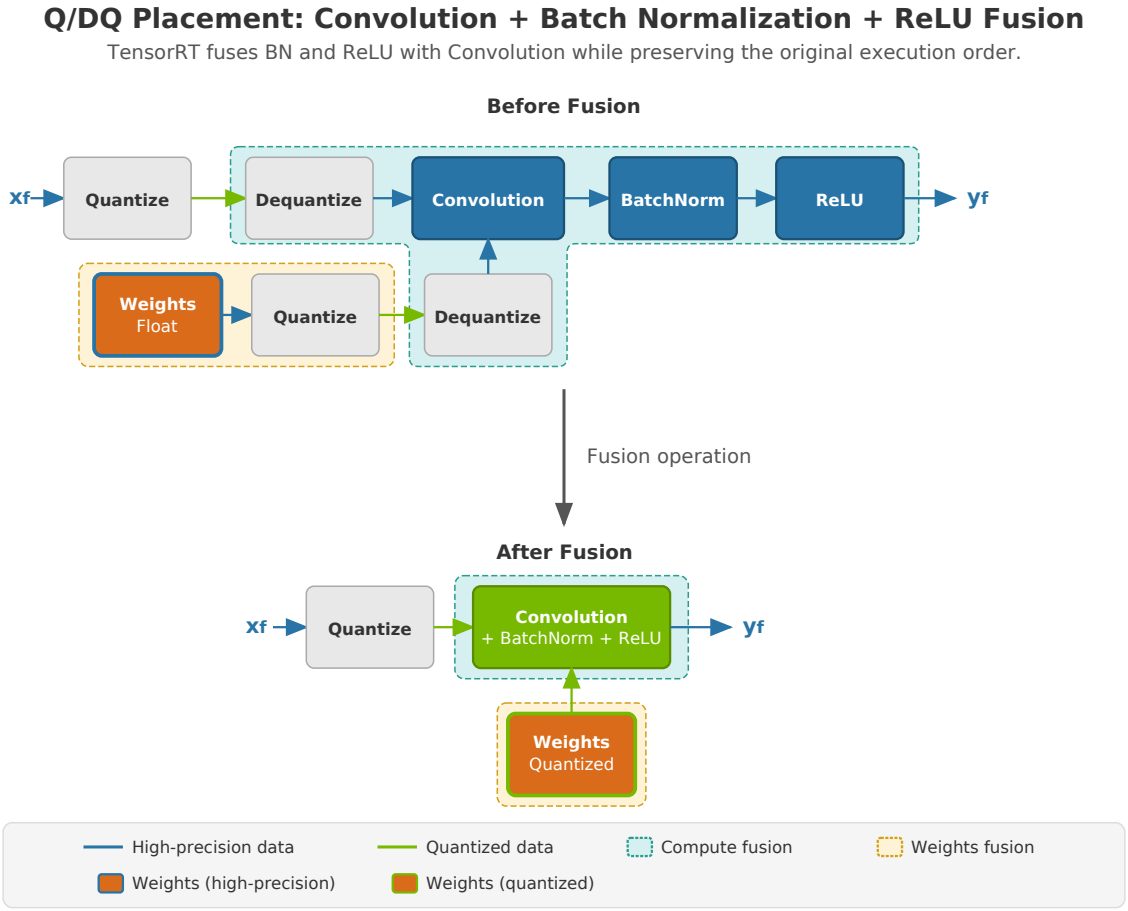
Quantize all inputs of weighted operations (Convolution, Transposed Convolution, and GEMM). Quantizing the weights and activations reduces bandwidth requirements and enables INT8 computation to accelerate bandwidth-limited and compute-limited layers.



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), the activation requires higher-precision input to produce acceptable accuracy.

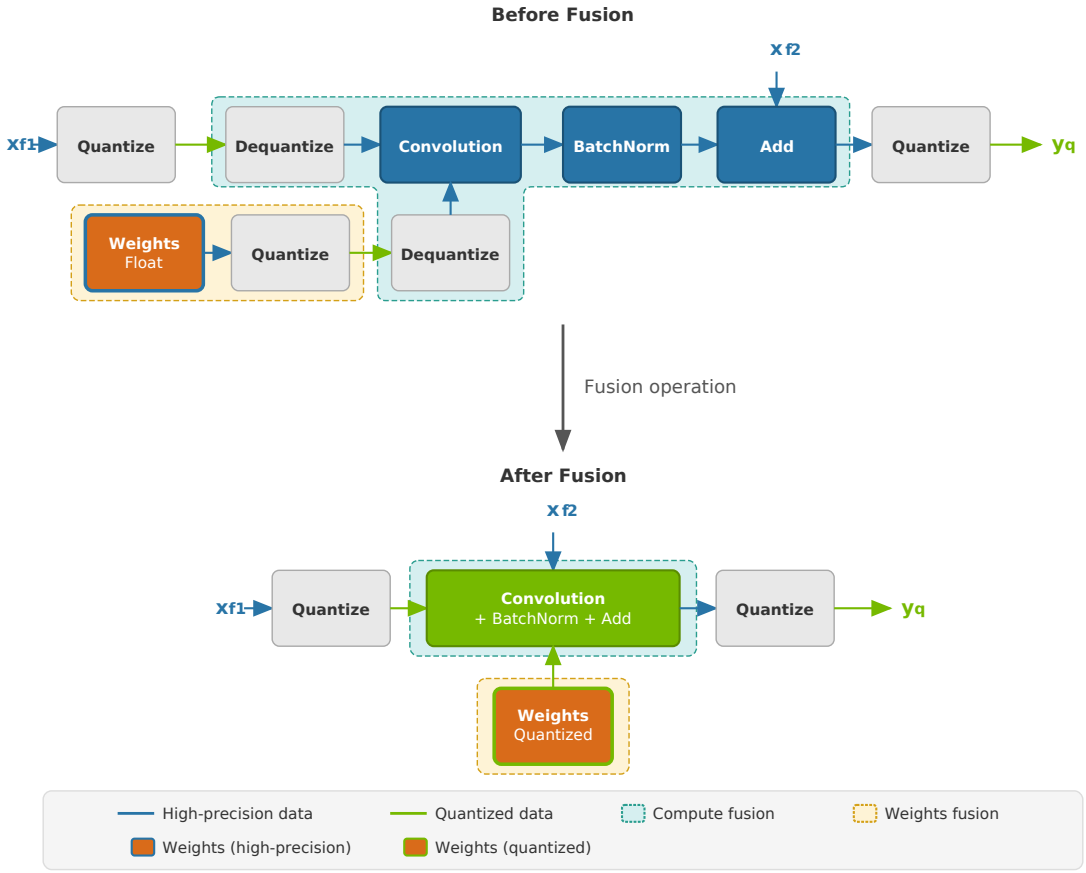


Do not simulate batch normalization and ReLU fusions in the training framework because TensorRT optimizations guarantee the preservation of these operations' arithmetic semantics.

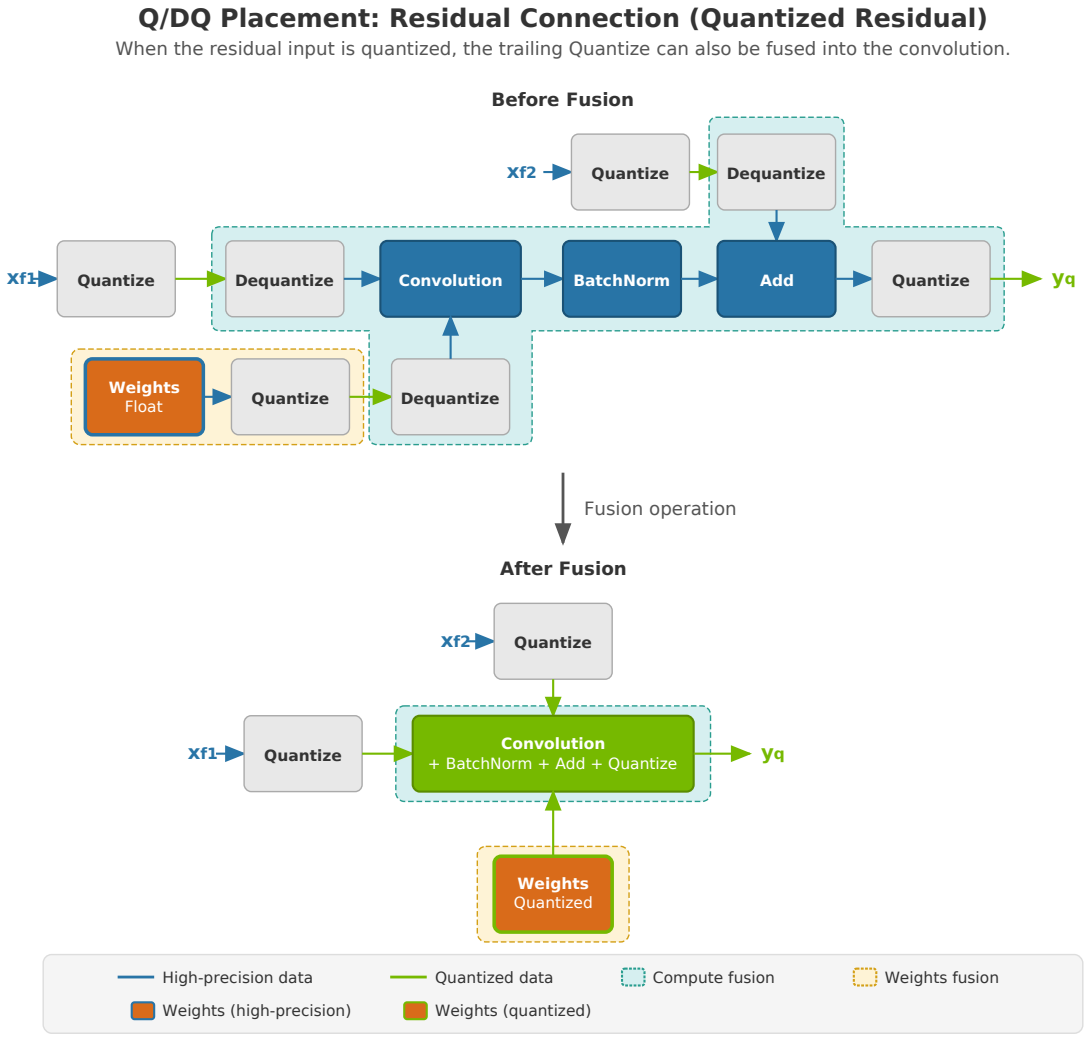


Quantize the residual input in skip connections. TensorRT can fuse element-wise addition following weighted layers, which is useful for models with skip connections like ResNet and EfficientNet. The precision of the first input to the element-wise addition layer determines the fusion output's precision.

For example, in the following diagram, the precision of x_f^2 is high precision, so the output of the fused convolution is limited to high precision, and the trailing Q-layer cannot be fused with the convolution.



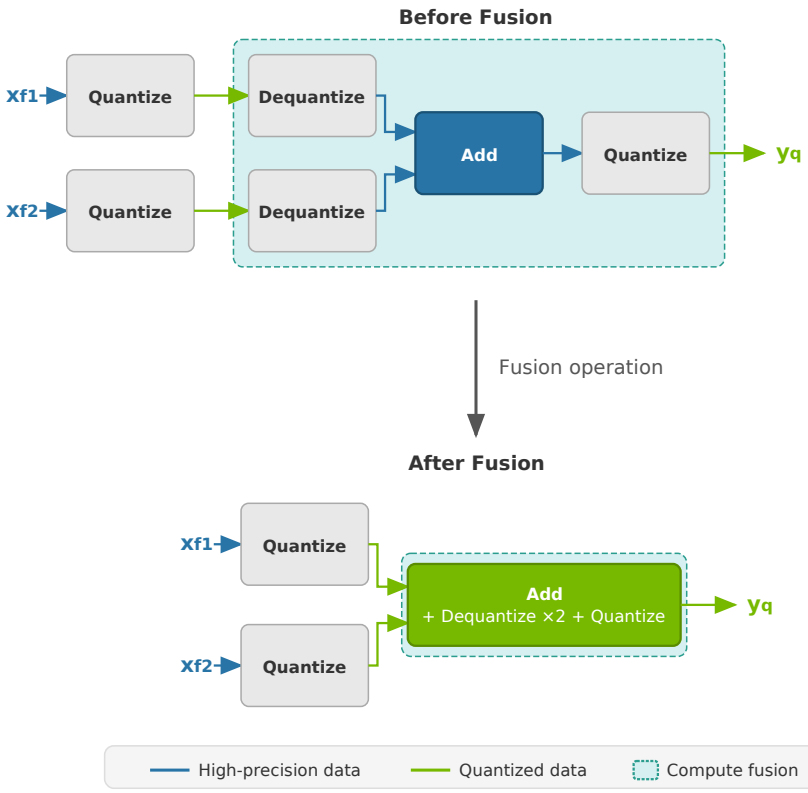
In contrast, when x_f^2 is quantized to INT8, as depicted in the following diagram, the output of the fused convolution is also INT8, and the trailing Q-layer is fused with the convolution.



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

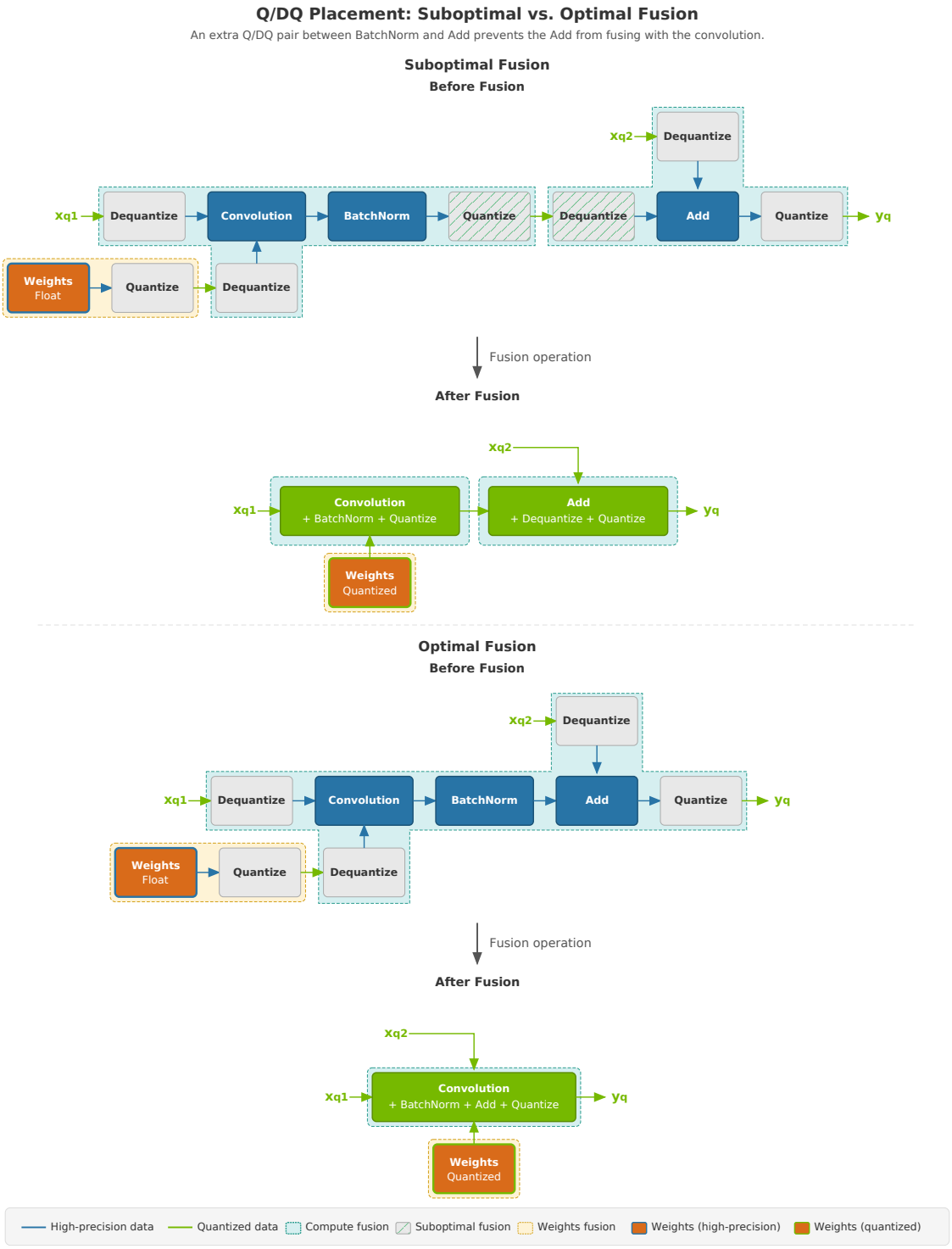
Q/DQ Placement: Quantizing a Non-Weighted Operation

An Add with two quantized inputs is fused with the input Dequantize layers and the output Quantize layer.



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 both accuracy and TensorRT performance in mind**.

The following figure contrasts a suboptimal Q/DQ placement against an optimal one for a convolution followed by an 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).

8.3.6. Q/DQ Limitations

Some of the Q/DQ graph-rewrite optimizations that TensorRT performs compare the quantization scales of two or more Q/DQ layers and only apply the rewrite when those scales are equal. When you refit a refittable TensorRT engine, the scales of Q/DQ nodes can be assigned new values. During refitting, TensorRT checks whether any Q/DQ layers that participated in scale-dependent optimizations have new values that break those rewrites. If they do, TensorRT throws an exception.

8.3.7. Q/DQ Interaction with Plugins

Plugins extend TensorRT's capabilities 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 applies to a TensorRT network with Q/DQ layers. When a plugin consumes quantized inputs (INT8/FP8) and generates quantized outputs, the input Dequantize and output Quantize nodes must be included in 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):

$$Input > Q \rightarrow DQ > Conv > Q \rightarrow DQ\_i > MyInt8Plugin > Q\_o \rightarrow DQ > Conv > Output$$

The $>$ arrows indicate activation tensors with FP32 precision, and the $\rightarrow$ arrows indicate INT8 precision.

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

$$Input > Q \rightarrow [DQ \rightarrow Conv \rightarrow Q] \rightarrow DQ\_i > MyInt8Plugin > Q\_o \rightarrow [DQ \rightarrow Conv] > Output$$

In the graph above, the plugin consumes and generates FP32 inputs and outputs. Because the plugin `MyInt8Plugin` uses INT8 precision, you must manually integrate DQ_i and Q_o into the plugin and then call `setOutputType(kINT8)` for that plugin layer. TensorRT then interprets the network as follows:

$$Input > Q \rightarrow DQ > Conv > Q \rightarrow MyInt8Plugin \rightarrow DQ > Conv > Output$$

Which it will fuse to:

$$Input > Q \rightarrow [DQ \rightarrow Conv \rightarrow Q] > MyInt8Plugin \rightarrow [DQ \rightarrow Conv] > Output$$

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 applies to using the scale from Q_o to quantize your plugin's output.

8.3.8. QAT Networks Using TensorFlow

You can use the TensorRT Model Optimizer 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.

TensorFlow 1 does not support per-channel quantization (PCQ), which is recommended for weights to preserve the model's accuracy.

8.3.9. QAT Networks Using PyTorch

PyTorch 1.8.0 and later supports exporting QuantizeLinear and DequantizeLinear ONNX operators with per-channel scales.

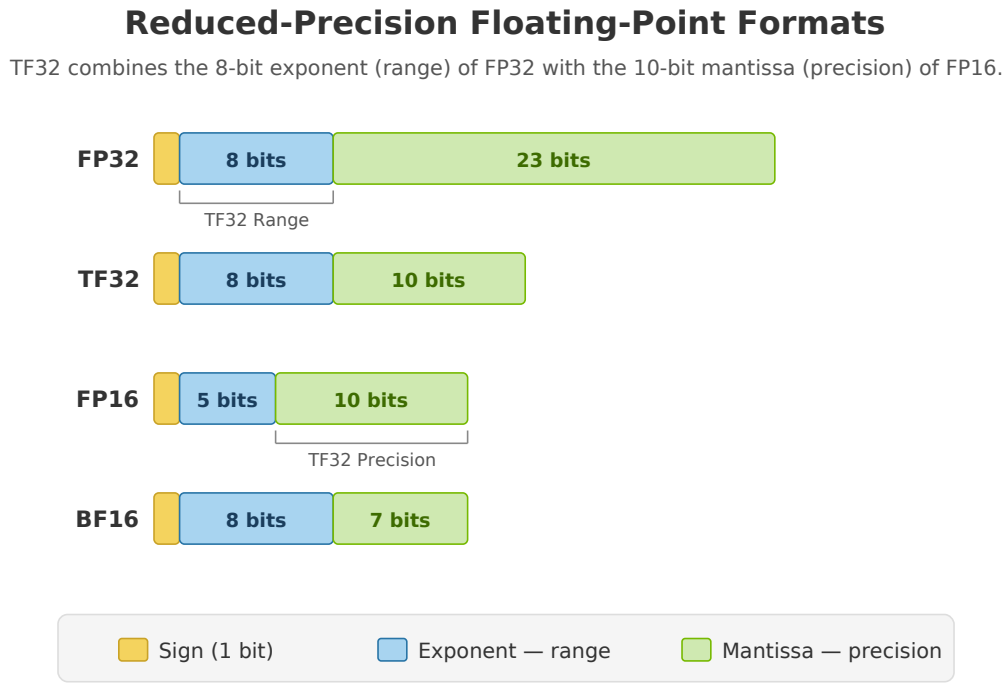
You can use the TensorRT Model Optimizer to calibrate INT8, perform QAT and PTQ for the various precisions that TensorRT supports, and export to ONNX.

Chapter 9. Accuracy Considerations

9.1. Reduced Precision Data Types

The choice of floating-point precision can significantly impact both performance and accuracy. In addition to a standard single-precision floating-point (FP32), TensorRT supports three reduced precision formats: TensorFloat-32 (TF32), half-precision floating-point (FP16), and Brain Floating Point (BF16).

TF32, enabled by default in TensorRT, uses an 8-bit exponent and a 10-bit mantissa, combining the dynamic range of FP32 with the computational efficiency of FP16. FP16, with a 5-bit exponent and a 10-bit mantissa, offers significant speed and memory efficiency benefits but can exhibit reduced precision due to its limited precision and range. BF16, featuring an 8-bit exponent and a 7-bit mantissa, provides a larger dynamic range than FP16 but with lower precision. The larger exponent makes BF16 suitable for scenarios where overflow is a concern, but precision can be reduced. Each format offers unique trade-offs, and the choice depends on the specific task requirements, such as the need for speed, memory efficiency, or numerical accuracy.



9.1.1. Impact of ULP on Large-Magnitude Values

ULP, "Unit in the Last Place," measures precision in floating-point arithmetic. It represents the smallest difference between two distinct floating-point numbers. ULP quantifies the gap between consecutive representable numbers in a given floating-point format. The size of this gap varies depending on the magnitude of the numbers being represented.

For large-magnitude values, the ULP can become quite large. This means that the difference between two consecutive floating-point numbers becomes significant. High ULP can lead to substantial rounding errors when numerical computations involve large-magnitude values. These errors accumulate and can cause numerical instability, ultimately degrading the accuracy of the computations.

In some models, the magnitude of the data increases as it passes through the network layers, especially in the absence of normalization layers. For instance, cascaded convolutional layers without normalization can amplify magnitudes, challenging reduced precision formats to maintain accuracy.

9.1.2. FP16 Overflow

FP16 has a narrower range of representable values than FP32, TF32, and BF16, making it more susceptible to overflow. FP16's 5-bit exponent limits its maximum value to 65,504, whereas FP32, TF32, and BF16 use an 8-bit exponent, offering a broader range. Overflow in FP16 results in Inf (infinity) values, which can propagate errors and lead to NaN (not-a-number) values, severely degrading model accuracy.

For example, the `IReduceLayer` is prone to overflows due to accumulating all values along a certain axis (except for min-reduce and max-reduce).

9.1.3. Sensitive Calculations

Certain model calculations are highly sensitive to precision changes. Using reduced precision formats for these calculations can cause significant accuracy loss due to their reduced precision.

For example, Sigmoid or Softmax amplify small numerical differences due to their exponential component.

9.1.4. Mitigation Strategies

To mitigate accuracy loss when using reduced precision during inference, consider the following strategies:

Mixed Precision Inference
: Combine FP16, BF16, TF32, and FP32 operations. Perform critical, precision-sensitive calculations in FP32 and use reduced precision for less sensitive operations to gain performance benefits. Linear operations are particularly good candidates for reduced precision because, besides the reduced bandwidth, their compute is accelerated by the Tensor Cores. This can be achieved by adding `ICastLayer` in a strongly typed mode or setting layer precision constraints in a weakly typed mode. For more information, refer to the *Strongly Typed Networks* and *Reduced Precision in Weakly-Typed Networks* sections.

Control Computation Precision
: In addition to setting a layer's input/output precisions, it is sometimes possible to control the internal computation precision. For more information, refer to the *Control of Computational Precision* section.

Magnitude Adjustment
Scale the input data to prevent the accuracy loss associated with high-magnitude data.

See also

Reduced Precision in Weakly-Typed Networks
How TensorRT selects precisions automatically and how to constrain per-layer precision.

Control of Computational Precision
Controlling internal accumulation precision for GEMM and normalization layers.

Tradeoffs Between Accuracy and Performance
Step-by-step checklist for diagnosing and improving accuracy in best practices.

How TensorRT Works
Why different builds or precisions can produce non-bitwise-identical results (see Determinism section).

Troubleshooting
Debugging NaN/Inf with FP16/BF16, calibration warnings, and reporting accuracy issues.

9.2. Quantized Data Types

9.2.1. Uniform and Non-Uniform Distributions in Quantized Types

Integer formats, such as INT8 and INT4, use uniform quantization, which means the range of values is divided into equal-sized intervals. This method is simple and efficient but might not optimally capture the distribution of weights and activations.

Floating point formats such as FP8E4M3 and FP4E2M1 have non-uniform distributions with more values concentrated near 0, which better aligns with the typical distribution of neural network weights and activations.

9.2.2. Quantization Errors

Quantization introduces two sources of error that can significantly impact the accuracy of deep learning models: rounding errors and clamping errors.

Rounding Errors occur when continuous values are approximated to the nearest quantized value, causing a loss of information. TensorRT uses the round-to-nearest-even method, which rounds to the nearest even value in case of ties, helping reduce bias in the quantization process.

Clamping Errors occur when values exceed the quantization range and are clipped to the nearest boundary, causing a loss of dynamic range. There is a tradeoff between the clamping error and the rounding error, and this is expressed in the scale selection: Setting a scale that allows fewer values to be clipped means a higher rounding error.

See also

Working with Quantized Types
Comprehensive guide to PTQ, QAT, calibration, Q/DQ placement, and quantization workflows.

Working with Transformers
Precision support for multi-head attention fusion (FP16/BF16/FP8/INT8) and numerical considerations.

Chapter 10. 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. Specify each runtime dimension of an input tensor by using -1 as a placeholder for the dimension.
2. Specify one or more *optimization profiles* at build time that specify the permitted range of dimensions for inputs with runtime dimensions and the dimensions that the auto-tuner will optimize for. For more information, refer to the *Optimization Profiles* section.
3. To use the engine:
    1. Create an execution context from the engine, the same as without dynamic shapes.
    2. Specify one of the optimization profiles from step 2 that covers the input dimensions.
    3. 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.
    4. Enqueue work.

To change the runtime dimensions, repeat steps 3b and 3c, which do not have to be repeated until the input dimensions change.

See also

Architecture Overview
Foundational context for the TensorRT developer experience - complementary GPU/software ecosystem (MIG, Triton, DALI, Model Optimizer, Polygraphy, Nsight tools), ONNX import, API versioning, and the deprecation policy.

10.1. Dynamic Shapes: Core Concepts

10.1.1. 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))
```

After choosing an optimization profile, you must set the input dimensions at run time (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)`.

The return value of `setInputShape` for input only indicates consistency for the optimization profile set for that input. After all input binding dimensions are specified, you can check whether the entire network is consistent with 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, such as it depends on the input of `INonZeroLayer`, `outDims.d[k]` will be `-1`. For more information on such outputs, refer to the *Dynamically Shaped Output* section.

10.1.2. Named Dimensions

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

1. 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))`.
2. Dimensions with the same name are implicitly equal, which can help the optimizer generate a more efficient engine and diagnose mismatched dimensions at runtime. For example, suppose two inputs have dimensions `[n,10,m]` and `[n,13]`. In that case, the optimizer knows the lead dimensions are always equal, and accidentally using the engine with mismatched values for n will be reported as an error.

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

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

C++

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

Python

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

There are corresponding methods to get a dimension name:

C++

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

Python

```
tensor.get_dimension_name(2) # returns the name of the third dimension of the
↪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.

10.1.3. Dimension Constraint using IAssertionLayer

Sometimes, two dynamic dimensions are not known to be equal statically but are guaranteed 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:

1. Give the dimensions the same name as described in the *Named Dimensions* section.
2. 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.
```

(continues on next page)

(continued from previous page)

```
↪ElementWiseOperation.EQUAL).get_output(0)
n.add_assertion(are_equal, "oops")
```

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

10.1.4. Optimization Profiles

An *optimization profile* describes a range of dimensions for each network input and the dimensions the auto-tuner will use for optimization. You must create at least one optimization profile at build time when using runtime dimensions. 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

The memory usage for different profiles can change dramatically based on the dimensions specified by the min, max, and opt parameters. Some operations 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 them 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 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 `enqueueV3()` invocation in this context. This ensures that the context executions happen after the optimization profile setup.

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

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

`setOptimizationProfileAsync()` replaces the now deprecated `setOptimizationProfile()`. Using `setOptimizationProfile()` to switch between optimization profiles can cause GPU memory copy operations in the subsequent `enqueueV3()` operations. To avoid these calls during enqueue, use the `setOptimizationProfileAsync()` API instead.

10.1.5. Dynamically Shaped Output

If the output of a network has a dynamic shape, several strategies are 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 the dimensions of the input tensors and *input shape tensors*. Use the `IExecutionContext::inferShapes()` method to check if you forgot to supply the necessary information.

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

1. Derive your 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 ...*/
```

(continues on next page)

(continued from previous page)

```
        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 -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 `IExecutionContext::setTensorAddress`, or call it with a `nullptr` for the tensor address.

B

Preallocate enough memory based on what `IExecutionContext::getMaxOutputSize` reports as an upper bound. This guarantees that the engine will not fail due to insufficient output memory, but the upper bound can be so high that it is useless.

C

If you have preallocated enough memory based on experience, use `IExecutionContext::setTensorAddress` to tell TensorRT about it. If the tensor does not fit, make `reallocateOutput` return `nullptr`, 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
    {
```

(continues on next page)

(continued from previous page)

```
        cudaFree(outputPtr);
    }
};
```

TensorRT internally allocates memory asynchronously in the device's current memory pool for networks with data-dependent shapes. Suppose the current device memory pool does not have a release threshold set. In that case, performance degradation between runs can occur as the memory is returned to the operating system upon stream synchronization. In these cases, it is recommended that you either provide the TensorRT runtime with a custom `IGpuAllocator` with a custom memory pool or experiment with setting the release threshold. More information about setting the release threshold can be found in Retaining Memory in the Pool and the Code Migration Guide.

10.1.6. Looking up Binding Indices for Multiple Optimization Profiles

If you use `enqueueV3` instead of the deprecated `enqueueV2`, you can skip this section because name-based methods such as `IExecutionContext::setTensorAddress` do not expect a profile suffix.

Each profile has separate binding indices in an engine built from multiple profiles. The names of I/O tensors for the K 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]`.

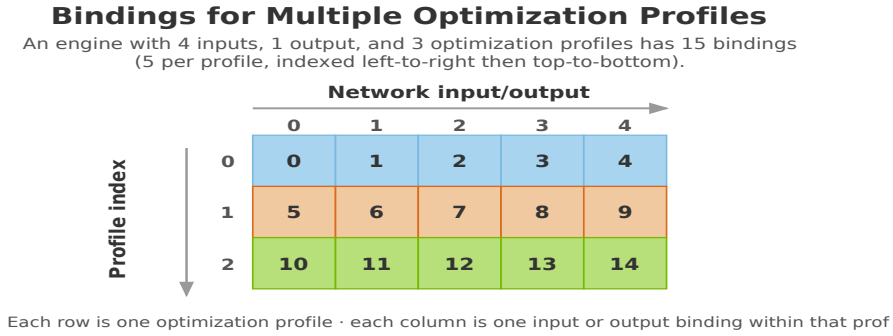
Likewise, if using `ICudaEngine::getBindingIndex(name)` to get the index for a profile K beyond the first profile (`K=0`), append `[profile K]` to the name used in the `INetworkDefinition`. For example, if the tensor was called `foo` in the `INetworkDefinition`, `engine.getBindingIndex("foo [profile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

Always omit the suffix for `K=0`.

10.1.7. 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, and three optimization profiles in the `IBuilderConfig`. The engine has 15 bindings, five for each optimization profile, conceptually organized as a table:



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" in the scenario where the binding belongs to the first profile, but another profile was specified. TensorRT warns about the mistake in this case and then chooses the correct binding index from the same column.

10.2. Dynamic Shapes: Advanced Topics

10.2.1. Layer Extensions For Dynamic Shapes

Some layers have optional inputs that allow specifying dynamic shape information; `IShapeLayer` can access a tensor's shape at runtime. Some layers also allow for 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 reshaped 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 calculate shapes and create new shape tensors.

10.2.2. 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.
- `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.

Not all required build-time constants need to be set manually. TensorRT will infer shapes through the network layers, and only those that cannot be inferred to be build-time constants must be set manually.

10.2.3. Execution Tensors vs Shape Tensors

TensorRT 8.5 largely erased the distinctions between execution tensors and shape tensors. However, when designing a network or analyzing performance, it can 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 you run out of work or reach an unknown shape. 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.

When TensorRT needs a shape tensor, but the tensor has been classified as an execution tensor, the runtime copies the tensor from the GPU to the CPU, which incurs synchronization overhead.

Some layers are polymorphic in terms of the kinds of tensors they handle. For example, `IElementWiseLayer` can sum two INT32 execution tensors or two INT32 shape tensors. The type of the output tensor depends on its ultimate use. If the sum is used to reshape another tensor, it is a shape tensor.

10.2.4. 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 concerning 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 that also serves as a reference for the layers that can be used to manipulate shape tensors:

```
IAssertionLayer: S
IConcatenationLayer: t × t × ...  t
ICumulativeLayer: t × t   t
IIfConditionalInputLayer: t   t
IIfConditionalOutputLayer: t   t
IConstantLayer:   t
IActivationLayer: t   t
IElementWiseLayer: t × t   t
IFillLayer: S   t
IFillLayer: S × t × t   t
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 × ...
```

The inferred types are not exclusive because the output can be the input of more than one subsequent layer. 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, suppose the third parameter specifies the result's size and is not a build-time constant. In that case, 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 an 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 its second optional input to be a shape tensor. If the output of `IElementWiseLayer` is a shape tensor, its inputs are, too.

10.2.5. 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:

1. At build time: the optimization profile *values* of the shape tensor.
2. At run time: the *values* of the shape tensor.

The shape of an input shape tensor is always known at build time. The values 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* or *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, and 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 the inputs of B that are shape tensors and mark the corresponding output tensor in network A.

Shape tensors at network boundaries must have the 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 using `ICudaEngine::isShapeInferenceIO()`.

Chapter 11. Extending TensorRT with Custom Layers

NVIDIA TensorRT supports many layers, and its functionality is continually extended; however, there can be cases where the layers supported do not cater to a model's specific needs. In such cases, TensorRT can be extended by implementing custom layers, often called 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.

You can write and add your own if these plugins do not meet your needs.

See also

Architecture Overview
Foundational context for the TensorRT developer experience - complementary GPU/software ecosystem (MIG, Triton, DALI, Model Optimizer, Polygraphy, Nsight tools), ONNX import, API versioning, and the deprecation policy.

11.1. Adding Custom Layers Using the C++ API

There are four steps to ensure that TensorRT properly recognizes your plugin:

1. Implement a plugin class from one of TensorRT's plugin base classes. Currently, the only recommended one is `IPluginV3`.
2. Implement a plugin creator class tied to your class by deriving from one of TensorRT's plugin creator-based 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 by directly using TensorRT's network APIs or loading an ONNX model using the TensorRT ONNX parser APIs.

The following sections explore each of these steps in detail.

11.1.1. 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 the 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`, `IPluginV3OneBuild`, and `IPluginV3OneRuntime` are the base classes that an `IPluginV3` plugin must implement to display the core, build, and runtime capabilities, respectively. If I/O aliasing is required, `IPluginV3OneBuildV2` can be used as the build capability, which contains a superset of the functionalities in `IPluginV3OneBuild`.

11.1.2. Implementing a Plugin Creator Class

To use a plugin in a network, you must first register it with TensorRT's `PluginRegistry`. Rather than registering the plugin directly, you register an instance of a factory class for the plugin, derived from a child class of `IPluginCreatorInterface`. 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`. `IPluginCreatorV3One::createPlugin()`, which has the signature below.

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()` can 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`.

- The returned `IPluginV3` object must have a valid core capability in both phases.
- 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.

11.1.3. Registering a Plugin Creator with the Plugin Registry

There are two ways that you can register plugin creators with the registry:

1. Statically register by calling `REGISTER_TENSORRT_PLUGIN`. `REGISTER_TENSORRT_PLUGIN` always registers the creator under the default namespace (“”).
2. Dynamically register by creating an 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 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` that was serialized is passed back as the `fc` argument.

11.1.4. 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 field parameters for the plugin layer
PluginFieldCollection *pluginData = parseAndFillFields(pluginFC, layerFields);
// Create the plugin object using the layerName and the plugin metadata 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)
```

The `createPlugin` method described previously creates a new plugin object on the heap and returns a pointer. As shown previously, ensure you delete the `pluginObj` to avoid a memory leak.

When the engine is deleted, the engine destroys any clones of the plugin object created during the build. You are responsible for ensuring the plugin object you created is freed after it is added to the network.

Note

- Do not serialize all plugin parameters, only those required 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`. Refer to the NVIDIA TensorRT Safety Production Guide for DriveOS for any safety-related activities.

11.1.5. Example: Adding a Custom Layer with Dynamic Shapes Using C++

Imagine that a custom layer is needed for a padding-like operation where each image in an input batch must be reshaped to 32 x 32. The input tensor X would be of shape (B, C, H, W), and the output Y would be of shape (B, C, 32, 32). To accomplish this, a TensorRT plugin can be written using the `IPluginV3` interface; let us call it `PadPlugin`.

Since an `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. However, a multiple inheritance approach is easier for most use cases, particularly when coupling build and runtime capabilities in a single class is tolerable.

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
{
    // All plugin interface methods are noexcept and care should be
    // taken not to throw exceptions across the API boundary. It is
    // recommended to catch any exceptions and return a value that
    // appropriately represents the error status.
    try
    {
        if (type == PluginCapabilityType::kBUILD)
        {
            return static_cast<IPluginV3OneBuild*>(this);
        }
        if (type == PluginCapabilityType::kRUNTIME)
        {
            return static_cast<IPluginV3OneRuntime*>(this);
        }
```

(continues on next page)

(continued from previous page)

```
        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`

`INetworkDefinition::addPluginV3` can add the plugin to the network.

```
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, which 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 equal 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 the plugin accepts a given type and format combination for a connection at a given position `pos` and 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) and when the engine is being executed (in the runtime phase).

1. `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` values 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.
2. `IPluginV3OneRuntime::onShapeChange`: Called during both the build-phase and runtime phase before `enqueue()` to communicate the 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
```

(continues on next page)

(continued from previous page)

```
{
    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 each input and output's dimensions, type, and format.

```
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
}
```

11.1.6. Example: Adding a Custom Layer with 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 and 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 depends on the shape of X.
- The fourth dimension of Y is data-dependent.

Similar to `PadPlugin` in the prior example, `BarPlugin` uses multiple inheritance.

To add the plugin to the network, `INetworkDefinition::addPluginV3` can be used similarly. After the device tensor inputs, `addPluginV3` takes two additional arguments to specify the shape tensor inputs.

```
std::vector<ITensor*> inputs{X};
std::vector<ITensor*> shapeInputs{S};
```

(continues on next page)

(continued from previous page)

```
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 and the bounds of each data-dependent dimension of each output tensor. The bounds can be expressed using 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 input 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 output Y is the same type as the device input X and that the data-dependent dimension size fits INT32 (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 that size tensors must be explicitly declared OD.

```
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.
```

(continues on next page)

(continued from previous page)

```
    // 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 device input `X` must have `DataType::kFLOAT` or `DataType::kHALF`.
- The output `Y` must have the same type as `X`.
- The size tensor output has the type `DataType::kINT32`.

Note

Shape inputs passed to the plugin through `addPluginV3` only appear 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;
```

(continues on next page)

(continued from previous page)

```
        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 `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 differs greatly 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 large enough to hold the corresponding output tensor (based on the upper bound specified through `getOutputShapes`).

11.1.7. Example: Adding a Custom Layer with INT8 I/O Support Using C++

`PoolPlugin` is a plugin demonstrating how to `addINT8` I/O for a custom pooling layer using `IPluginV3`. `PoolPlugin` uses multiple inheritance to derive 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 the INT8 I/O combination that is allowed. This interface is similar to *Example: Adding a Custom Layer with Dynamic Shapes 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 ||
```

(continues on next page)

(continued from previous page)

```
            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 tensor scales must be set explicitly.
- Calibration cannot determine the dynamic range of a plugin's internal tensors. Plugins that operate on quantized data must calculate their dynamic range for internal tensors.
- A plugin can be designed to accept 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 on a per-tensor basis have been obtained from `PluginTensorDesc::scale`.

11.2. Adding Custom Layers using the Python API (TensorRT >= 10.6)

For most use cases, defining Python plugins with a decorator-based approach is recommended (available starting in TensorRT 10.6). Note that embedding Python-defined plugins to TensorRT engines such that the engine is independent of Python and the plugin source itself, is only possible with this approach.

11.2.1. Adding Custom Layers using the Python API (Advanced/TensorRT <= 10.5)

Python with a class-based approach (it is also the only supported approach for TensorRT <= 10.5). In contrast to decorator-based Python plugins (described in the preceding section), class-based plugins offer the following:

- Statefulness: class-based plugins have stated (such as configured/non-configured) and more granular querying by TensorRT for different plugin properties and behaviors.
- Shape tensor input support.
- Fine-grained control of the plugin instances TensorRT creates during engine deserialization is only possible with custom *plugin creator* definitions, which are only available with a class-based approach.
- Manual serialization and deserialization of plugin attributes.
- Ability to pre-request a device memory scratch space (workspace in addition to input/output buffers) to avoid execution-time device memory allocations.

These often come at the expense of increased implementation complexity and code bloat, which can lead to more bugs. Therefore, a tradeoff analysis is recommended before considering class-based plugin implementations in Python.

Implementing a class-based plugin in Python is similar to C++ in that implementation of `IPluginV3` and `IPluginCreatorV3One` is necessary. Interface methods in Python have mostly similar APIs to their C++ counterparts, and most differences are minor and self-explanatory.

The following list includes a few selected changes. Subsequent subsections describe the differences involved in more detail.

- The following plugin APIs have been omitted in favor of reading/writing to an appropriately named attribute.

Table 11.1: Adding Custom Layers using the Python API (Advanced/TensorRT <= 10.5)

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	getFormatCombination-Limit()	format_combination_limit [int]
IPluginCre-atorV3One	getPluginNamespace()	plugin_namespace [str]
IPluginCre-atorV3One	getFieldNames()	field_names [PluginFieldCollec-tion]
IPluginCre-atorV3One	getPluginName()	name [str]
IPluginCre-atorV3One	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 an `IPluginV3` C++ destructor within the `IPluginV3.destroy()` method.

  For full examples demonstrating Python plugins, refer to the python_plugin sample.

11.2.1.1 Registration of a Python Plugin

Python plugins must be registered dynamically through the `IPluginRegistry.register_creator()` API. There is no analog to the `REGISTER_TENSORRT_PLUGIN` available for static registration.

11.2.1.2 Building and Running TensorRT Engines Containing Python Plugins

It is possible to build TensorRT engines using Python-based plugins. However, running such engines outside of Python is currently impossible 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.

11.2.1.3 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, a `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 this 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)
```

(continues on next page)

(continued from previous page)

```
...
checkCudaErrors(cuda.cuLaunchKernel(_float_kernel,
                                    num_blocks, 1, 1,
                                    block_size, 1, 1,
                                    0,
                                    stream_ptr,
                                    kernel_args, 0)
```

11.2.1.4 Translating Device Buffers/CUDA Stream Pointers in enqueue to other Frameworks

Constructing CuPy arrays on top of device buffers is possible 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)
```

11.2.1.5 Automatic Downcasting

TensorRT 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` and another type `MyOtherPluginCreator`, both plugin creators will be present as those respective types in the list.

11.2.1.6 Example: Adding a Custom Layer to a TensorRT Network Using Python

Using plugin nodes, custom layers can be added to any TensorRT network in Python. The Python API has a function called `add_plugin_v3` that enables adding a plugin node to a network. The following example illustrates this. It creates a simple TensorRT network and adds a hypothetical 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))
```

11.3. Enabling Timing Caching and Using Custom Tactics

`IPluginV3` provides more control over the profiling of custom layers, which were unavailable 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 (such as 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 could 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++

```cpp
char const* FooPlugin::getTimingCacheID() noexcept override
{
    // return nullptr to disable timing caching (default behavior)
    // return non-null string to enable timing caching
}
```

Python

```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 considered a suffix to a larger cache ID; TensorRT automatically forms a prefix by considering the plugin's input/output shape and format information. Usually, the user-provided timing cache ID could consist of plugin attributes and their values.
- It must reflect the plugin's creation state and not evolve after creation.

For V2 plugins, TensorRT only times the plugin for any (multiple) type/format combinations it claims to support. With `IPluginV3`, plugins can also ensure custom tactics are timed, and TensorRT uses the fastest tactic. For example, the plugin can have one of two kernels to compute the output, and it cannot be possible to predict the one that 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 can choose not to time the plugin if it only supports one type/format combination and either does not use custom tactics or only advertises one.
- 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 (TensorRT reserves `0` as the default tactic).

When the plugin is timed, `configurePlugin()` is guaranteed to be called with the current input/output format combination before `getValidTactics()` is called. Therefore, it is possible to advertise a different set of tactics per input/output format combination. For example, for a plugin that supports FP32 and FP16, tactic 1 can be restricted to only FP16 while supporting tactics 1 and 2 for FP32.

During the engine build, when auto-tuning the plugin, TensorRT will communicate the tactic for the subsequent `enqueue()` by invoking `IPluginV3OneRuntime::setTactic`. When an engine is deserialized, TensorRT will invoke `setTactic` after the plugin has been created to communicate the best tactic chosen for the plugin. Even if custom tactics are not used, `setTactic` will be called with the default tactic value `0`.

11.3.1. Sharing Custom Resources Among Plugins

Starting in TensorRT 10.0, a key-value store is associated with the plugin registry. This store can store user-implemented `IPluginResource` objects against a string key. This functionality can be used to share state or some resources among different plugins. Note that it is not tied to `IPluginV3` (or even to plugin interfaces).

Let us explore an example.

11.3.1.1 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 might prefer that these weights be downloaded when the engine runs. However, due to `W`'s large size, it is also desirable that only one copy is downloaded, which is shared among all plugins needing 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)
```

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 a `resource`. The registered object is obtained by invoking `resource->clone()`. Therefore, it is best practice only to initialize clones – in this case, the weight download can be done in `IPluginResource::clone()`.

3. After each plugin has finished using the weights, it can call `IPluginRegistry::releasePluginResource()` to signal that it no longer wishes to use them.

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
        {
            PLUGIN_CHECK(cudaFree(mWeights));
```

(continues on next page)

(continued from previous page)

```
            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
    {
        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 and runtime phases.

```
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
    }
}
```

All plugins requiring access to the weights can use the same code above. The reference counting mechanism will ensure the weights' availability and proper freeing.

11.3.2. 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 node's attributes to the plugin creator as plugin field parameters 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.

When a TensorRT plugin shares a name with a standard ONNX operator, the ONNX parser now provides better control over operator resolution through two key enhancements:

1. Attribute-Based Prioritization

The ONNX parser determines the implementation to use based on the presence of a `plugin_namespace` attribute on the graph node:

- **If the attribute is present:** The parser prioritizes matching the node to a plugin.
- **If the attribute is absent:** The parser prioritizes matching to a standard ONNX operator or function.

This behavior ensures that you can explicitly specify when a plugin should be used by setting the `plugin_namespace` attribute.

2. Plugin Override Flag

A new parser flag, `kENABLE_PLUGIN_OVERRIDE`, provides direct control over plugin precedence:

- **C++ API:** `OnnxParserFlag::kENABLE_PLUGIN_OVERRIDE`
- **Python API:** `OnnxParserFlag.ENABLE_PLUGIN_OVERRIDE`

By default, this flag is **OFF** to prevent unintended overrides of standard ONNX operators. When enabled, the parser behavior changes:

- The parser directly matches the node type to any loaded plugin name, giving plugins precedence.
- The `plugin_namespace` attribute is no longer required for plugin matching.
- The parser only falls back to a standard ONNX operator or function if no matching plugin is found.

Note

Use the `kENABLE_PLUGIN_OVERRIDE` flag with caution. When enabled, plugins take precedence over standard ONNX operators, which can lead to unexpected behavior if plugins are inadvertently loaded with names that conflict with ONNX operators.

Sometimes, you can 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.

11.4. 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(...)`. Refer to the NVIDIA TensorRT Safety Production Guide for DriveOS for any safety-related activities.

11.4.1. `IPluginV3` API Description

The following section describes the functions of `IPluginV3` and, by extension, `IPluginV3OneCore`, `IPluginV3OneBuild` or `IPluginV3OneBuildV2`, and `IPluginV3OneRuntime`.

Since an `IPluginV3` object consists of different capabilities, `IPluginV3::getCapabilityInterface` can be called 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 can be omitted for objects added for the runtime phase.

There are a few methods used to request identifying information about the plugin. They can 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 plugin methods:

- `IPluginV3OneBuild::getNbOutputs`: Used to specify the number of output tensors.

- `IPluginV3OneBuild::getOutputShapes`: This function specifies the output shapes as a function of the input shapes or constants. The exception is data-dependent shapes with a specified upper bound and optimal tuning value.
- `IPluginV3OneBuild::supportsFormatCombination`: Used to check if a plugin supports a given data type and format combination.
- `IPluginV3OneBuild::getOutputDataType`: This function retrieves the data types of the output tensors. The returned data types must be in a format supported by the plugin.

If the `IPluginV3OneBuildV2` build capability is used, the plugin can also communicate to TensorRT that certain input-output pairs are aliased (share the same data buffer). TensorRT will query `IPluginV3OneBuildV2::getAliasedInput` to determine any such aliasing behavior. To use this feature, `PreviewFeature::kALIASED_PLUGIN_IO_10_03` must be enabled.

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.

`PluginFormat` counts the formats.

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 `IPluginV3OneBuild::getWorkspaceSize` method, which the builder calls to determine and preallocate scratch space.

The layer is configured, executed, and destroyed at build time to discover optimal configurations. After selecting the optimal configuration for a plugin, the chosen tactic and concrete shape/format information (except for data-dependent dimensions) are communicated to the plugin during inference. It is executed as many times as needed for the lifetime of the inference application and finally destroyed when the engine is destroyed.

The builder controls these steps 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*`: This function requests that a plugin clone be attached to an `ExecutionContext`, allowing the plugin to access any context-specific resources.
- `IPluginV3OneBuild::getTimingCacheId`: This function queries for any timing cached ID that TensorRT can use. If provided, it enables timing caching (it is disabled by default).
- `IPluginV3OneBuild::getNbTactics`: Used to query for the number of custom tactics the plugin chooses to use.
- `IPluginV3OneBuild::getValidTactics`: This function queries for any custom tactics the plugin can use. The plugin will be profiled for each tactic up to a maximum indicated by `IPluginV3OneBuild::getFormatCombinationLimit()`.
- `IPluginV3OneBuild::getFormatCombinationLimit`: This function queries the maximum number of format combinations that can be timed for each tactic (`0` if no custom tactics are advertised for the default tactic).
- `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` values of the optimization profile that the plugin is currently profiled for. The `desc.dims` field corresponds to the dimensions of plugin inputs specified at network creation. Wildcard dimensions can exist during this phase in the `desc.dims` field. At this point, the plugin can 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 wildcards will indicate.
- `IPluginV3OneRuntime::enqueue*`: Encapsulates the actual algorithm and kernel calls of the plugin and provides pointers to input, output, and scratch space, as well as 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 function properly during the inference stage after the engine has been deserialized.

11.4.2. `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. The default can be `""`.
- `getFieldNames`: To successfully create a plugin, you must know all the plugin's field parameters. 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 creates a plugin, passing 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()`. In this case, TensorRT takes ownership of plugin objects returned by `createPlugin`.

You can also invoke `createPlugin` to produce plugin objects to add to a TensorRT network. In this case, setting the phase argument to `TensorRTPhase::kBUILD` is recommended. The data passed with the `PluginFieldCollection` should be allocated and 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.

11.4.3. 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 should be refactored.

Key migration points from IPluginV2DynamicExt to IPluginV3

Keep in mind the following key points when migrating an `IPluginV2DynamicExt` plugin to `IPluginV3`:

- The plugin creator associated with the plugin must be migrated to `IPluginCreatorV3One`, the factory class for `IPluginV3` (`IPluginCreator` is the factory class for `IPluginV2` derivatives). This simply consists of migrating `IPluginCreator::deserializePlugin`. For more information, refer to the *Plugin Serialization and Deserialization* section.
- There is no equivalent to `IPluginV2::initialize()`, `IPluginV2::terminate()`, and `IPluginV2::destroy()` in `IPluginV3`. For more information, refer to the *Plugin Initialization and Termination* section.
- There is no equivalent to `IPluginV2Ext::detachFromContext()` in `IPluginV3`. For more information, refer to the *Accessing Context-Specific Resources Provided by TensorRT* section.
- `IPluginV3OneRuntime::attachToContext()` is markedly different from `IPluginV2Ext::attachToContext()` regarding arguments and behavior. For more information, refer to the *Accessing Context-Specific Resources Provided by TensorRT* section.
- 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(...)`. For more information, refer to the *Plugin Serialization and Deserialization* section.
- The `IPluginV3` equivalents of void return methods in `IPluginV2DynamicExt` will expect an integer status code as a return value (such as `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 a return value. It also shifts from per-output index querying to one-shot querying. A similar transition applies from `IPluginV2Ext::getOutputDataType` to `IPluginV3OneBuild::getOutputDataTypes`.

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 termination logic in `IPluginV2::terminate()` or `IPluginV2::destroy()` can be moved to the class destructor. The exception is in the Python API; `IPluginV3.destroy()` is provided as an alternative for a C++-like destructor.

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 an `IPluginResourceContext`, which in turn exposes resources that plugins can request.

In a departure from `IPluginV2Ext::attachToContext()`, cuDNN and cuBLAS handles are no longer provided by `IPluginResourceContext`; any plugins that 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. For more information, refer to the *Sharing Custom Resources Among Plugins* section.

`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 can need to hold a reference to the `IPluginV3` object that it is part of.

Any context-specific resource obtained through `IPluginResourceContext` can be used until the plugin is destroyed. Therefore, any termination logic implemented in `IPluginV2Ext::detachFromContext()` can be moved to the plugin destructor.

Plugin Serialization and Deserialization

For V2 plugins, serialization and deserialization were determined by the implementation of `IPluginV2::serialize`, `IPluginV2::getSerializationSize`, and `IPluginCreator::deserializePlugin`; `IPluginV3OneRuntime::getFieldsToSerialize` and `IPluginCreatorV3One::createPlugin` have replaced these. Note that the workflow has shifted from writing to/reading from a raw buffer to constructing and parsing a `PluginFieldCollection`.

TensorRT handles the serialization of types defined in `PluginFieldType`. 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();
```

(continues on next page)

(continued from previous page)

```
}

nvinfer1::PluginFieldCollection const* DummyPlugin::getFieldsToSerialize()
↪noexcept override
{
    return &mFCToSerialize;
}
```

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.

11.4.4. Side-by-Side V2 → V3 API Mapping

The following tables map `IPluginV2DynamicExt` and `IPluginCreator` methods to their `IPluginV3` / `IPluginCreatorV3One` equivalents, grouped by lifecycle phase. Use this as the at-a-glance reference when porting a plugin; the conceptual sections above describe the *why* for each change.

Core / Identity

IPluginV2* method	IPluginV3* equivalent	Notes
IPlug-inV2::getPluginType()	IPlug-inV3OneCore::getPluginN	Renamed; same semantics.
IPlug-inV2::getPluginVersion(	IPlug-inV3OneCore::getPluginV	Unchanged.
IPlug-inV2::getPluginNamespac	IPlug-inV3OneCore::getPluginN	Unchanged.
N/A	IPlug-inV3::getCapabilityInte	New. Required dispatch entry point for core/build/runtime capabilities.
N/A	IPlug-inV3OneBuild::getMetada	New (optional). Used by engine inspector and logs.

Build phase

IPluginV2DynamicExt method	IPluginV3OneBuild equivalent	Notes
`getNbOutputs()`	`getNbOutputs()`	Unchanged.
`getOutputDimensions(int DimsExprs const*, int, IExprBuilder&)`	`getOutputShapes(DimsExprs const*, int, DimsExprs const*, int, DimsExprs*, int, IExprBuilder&)`	Per-index → one-shot; output via parameter, returns `int32_t` status. Adds shape inputs and supports data-dependent shapes via `IExprBuilder::declareSizeTensor`.
`IPluginV2Ext::getOutput DataType const*, int)`	`getOutputDataTypes(Data int, DataType const*, int)`	Per-index → one-shot; returns `int32_t` status.
`supportsFormatCombinati PluginTensorDesc const*, int, int)`	`supportsFormatCombinati DynamicPluginTensorDesc const*, int, int)`	Receives `DynamicPluginTensorDesc` (includes min/opt/max).
`configurePlugin(DynamicPluginTensor const*, int, DynamicPluginTensorDesc const*, int)`	`configurePlugin(DynamicPluginTensor const*, int, DynamicPluginTensorDesc const*, int)`	Parameter list unchanged; now returns `int32_t` status instead of `void`.
`getWorkspaceSize(PluginTensorDesc const*, int, PluginTensorDesc const*, int) const`	`getWorkspaceSize(DynamicPluginTensc const*, int, DynamicPluginTensorDesc const*, int) const`	Switched to dynamic descriptors.
N/A	`getNbTactics()`, `getValidTactics(int32_t int32_t)`, `getFormatCombinationLimit()`	New (optional). Enable custom tactic profiling.
N/A	`getTimingCacheID(...)`	New (optional). Enables timing-cache reuse for the plugin.
N/A	`IPluginV3OneBuildV2::getAlia`	New (optional). Requires `PreviewFeature::kALIASED_PLUGIN_IO_10_03`.

Runtime phase

IPluginV2DynamicExt method	IPluginV3OneRuntime equivalent	Notes
enqueue(PluginTensorDesc const*, PluginTensorDesc const*, void const* const*, void* const*, void*, cudaStream_t)	enqueue(PluginTensorDesc const*, PluginTensorDesc const*, void const* const*, void* const*, void*, cudaStream_t)	Signature unchanged.
N/A	onShapeChange(PluginTensc const*, int, PluginTensorDesc const*, int)	New. Called when concrete shapes change between enqueue invocations.
N/A	setTactic(int32_t)	New. Communicates the chosen tactic before enqueue.
IPluginV2Ext::attachToC cublasContext*, IGpuAllocator*)	attachToContext(IPluginResourceCor	Now a *clone-and-attach* operation that returns a new IPluginV3*. cuDNN/cuBLAS handles are no longer provided.
IPluginV2Ext::detachFromCont	Removed	Move teardown logic to the destructor.

Serialization and lifetime

IPluginV2* method	IPluginV3* equivalent	Notes
IPluginV2::getSerializationS const + IPluginV2::serialize(void*) const	IPluginV3OneRuntime::getFiel	Raw byte buffer → structured PluginFieldCollection.
IPluginCreator::deserializePlugir const*, void const*, size_t)	IPluginCreatorV3One::createPlugir const*, PluginFieldCollection const*, TensorRTPhase)	Unified create/deserialize; phase == kRUNTIME indicates deserialization.
IPluginV2::clone() const	IPluginV3::clone()	Non-const; returns IPluginV3*.
IPluginV2::initialize()	Removed	Plugin must be constructed in an initialized state. Defer lazy init to configurePlugin or onShapeChange.
IPluginV2::terminate(), IPluginV2::destroy()	Removed	Move teardown logic to the destructor.

Network attachment

10.x	11.x	Notes
`INetworkDefinition::addPluginV2(ITens const*, int, IPluginV2&)`	`INetworkDefinition::addPluginV3(ITens const*, int, ITensor* const*, int, IPluginV3&)`	Adds a separate shape-inputs argument list.
`IPluginV2Layer`	`IPluginV3Layer`	1:1 layer-class replacement.

11.4.5. Known Migration Issues

Use strongly-typed networks with `IPluginV3`

Mixing `IPluginV3` plugins with weakly-typed networks (those still relying on `BuilderFlag::kFP16` / `kBF16` style precision selection in 10.x) can hit fusion paths that were not exercised by `IPluginV2DynamicExt`. The supported and recommended configuration in 11.x is to build with strongly-typed networks. Refer to *Migrating from Weak Typing to Strong Typing* in the *TensorRT Migration Guide* for the conversion steps.

In TensorRT 11.0 all precision-enabling builder flags have been removed, so any plugin migrated as part of a 10.x 11.x upgrade is automatically running in a strongly-typed network. Authors back-porting V3 plugins to a 10.x build for evaluation should explicitly opt into a strongly-typed network with `createNetworkV2(NetworkDefinitionCreationFlag::kSTRONGLY_TYPED)`.

`attachToContext` is now clone-and-attach

`IPluginV2Ext::attachToContext` mutated the existing plugin instance. `IPluginV3OneRuntime::attachToContext` instead clones the *entire* `IPluginV3` object and returns the new instance. Plugins that store a back-pointer from a runtime-capability sub-object to the owning `IPluginV3` must update that back-pointer in the cloned instance, otherwise the cloned runtime capability will dispatch back into the original plugin and produce stale or freed-memory accesses.

`PluginFieldCollection` lifetime during deserialization

In V2, `deserializePlugin` received a raw byte buffer that the plugin was free to consume immediately. In V3, `IPluginCreatorV3One::createPlugin` receives a `PluginFieldCollection` whose underlying buffers are owned by TensorRT and only valid for the duration of the call. Plugins must **copy** any data they need to retain (vectors, structs, weight blobs) into plugin-owned storage before returning from `createPlugin`. Holding pointers into the incoming `PluginField::data` past the call is a use-after-free.

Aliased I/O migration

Plugins that relied on overlapping input/output buffers in V2 should migrate to `IPluginV3OneBuildV2` and implement `getAliasedInput` to declare the aliasing explicitly, then enable `PreviewFeature::kALIASED_PLUGIN_IO_10_03` on the builder config. Refer to the *IPluginV3 API Description* section for details.

11.4.6. Performance: Resolving V2 → V3 Regressions

A plugin that was performance-tuned against `IPluginV2DynamicExt` may regress when first ported to `IPluginV3` because the V3 lifecycle exposes new opportunities (and a few new costs) that the V2 path did not. The following checklist resolves the most common regressions.

1. **Hoist allocations out of ``enqueue``.** `IPluginV2` had explicit `initialize()` / `terminate()` hooks that authors often used as a one-time setup site. `IPluginV3` removes these, so any setup that previously lived in `initialize()` should move to the constructor, `configurePlugin`, or `onShapeChange`, *not* into `enqueue`. Per-call allocations are a frequent source of measured regressions.
2. **Advertise tactics where multiple kernels exist.** Implement `getNbTactics` and `getValidTactics` so the builder can profile each kernel variant your plugin ships and pick the fastest. V2 had no equivalent and forced the plugin to choose at build time.
3. **Enable timing-cache reuse.** Implement `getTimingCacheID` so repeated builds of the same network reuse cached timings for the plugin. Without this, every build re-times every tactic.
4. **Use ``getWorkspaceSize`` instead of internal allocations.** Request scratch space through `getWorkspaceSize` so TensorRT pools the allocation across the network. Internal `cudaMalloc` in `enqueue` defeats this pooling.
5. **Build with strongly-typed networks.** Strong typing avoids autotuner fallback paths and exposes more fusion opportunities to the V3 plugin's neighbors. See *Known Migration Issues* above.
6. **Avoid device allocations in ``clone``.** `clone` is called frequently during the build phase; defer device-side allocations to `configurePlugin` and release them in the destructor. Refer to *Coding Guidelines for Plugins* below.
7. **Profile build vs. runtime separately.** Use `trtexec --verbose --profilingVerbosity=detailed` to confirm whether the regression is in the build phase (extra autotuning) or the inference phase (extra per-call work). They have different remediations.

11.4.7. 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 be available during `enqueue`.

Add Checks to Ensure Proper Configuration and Validate Inputs

A common source for unexpected plugin behavior is improper configuration (such as 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` or `onShapeChange`: Input dimension checks
- `enqueue`: Input value checks

Return Null at Errors for Methods That Create a New Plugin Object

Methods like `createPlugin`, `clone`, and `attachToContext` can be expected to create and return new plugin objects. In these methods, ensure 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 configured incorrectly.

Avoid Device Memory Allocations in clone()

Since the builder calls clone multiple times, device memory allocations could be significantly expensive. One option is to do persistent memory allocations in the constructor, copy to a device when the plugin is ready (such as 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.

11.4.8. 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*). However, you may want TensorRT to manage the 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. To take advantage of this, you can build shared libraries with specific entry points recognized by TensorRT.

11.4.8.1 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* getCreators(int32_t& nbCreators) const;
```

`extern "C"` above is only used to prevent name mangling, and the methods 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 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`.

You can also package plugins in the plan when building version-compatible engines. The packaged plugins will have the same lifetime as the engine and will be automatically registered/deregistered when running the engine.

11.4.8.2 Using Plugin Shared Libraries

After building your shared libraries, you can configure the builder to serialize them with the engine. Next time you load the engine into TensorRT, the serialized plugin libraries will be loaded and registered automatically.

Note

`IPluginRegistry loadLibrary()` functionality now supports plugin-shared libraries containing both V2 and V3 plugin creators through the `getCreators()` entry point. The `getPluginCreators()` entry point is valid, too, but is deprecated. TensorRT first checks if the `getCreators()` symbol is available, and if not, checks for `getPluginCreators()` as a fallback for backward compatibility. You can then query this to enumerate each plugin creator and register it manually using `IPluginRegistry registerCreator()`.

Load the plugins for use with the builder before 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 before deserializing the engine:

C++

```
// In this example, getExternalPluginLibs() is a user-implemented method that
 ↪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 that 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 12. 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 *scan outputs* and *last value* outputs.

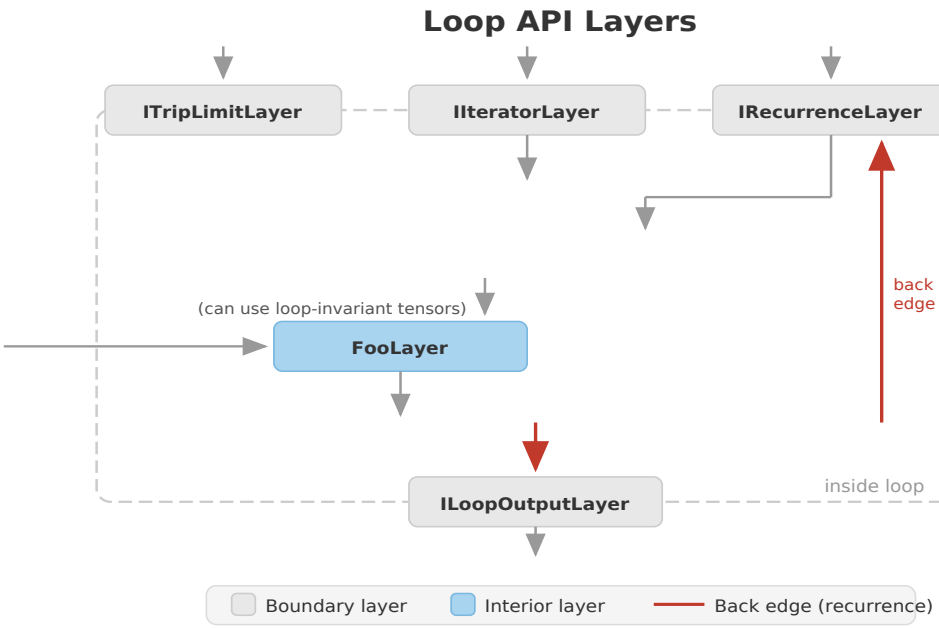
12.1. Defining a Loop

Loop boundary layers define a loop.

- `ITripLimitLayer` specifies how many times 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 boundary layer inherits from the class `ILoopBoundaryLayer`, which has a method `getLoop()` for getting its associated `ILoop`. The `ILoop` object identifies the loop, and all loop boundary layers with the same `ILoop` belong to that loop.

The following figure depicts a loop structure and data flow at the boundary. Loop-invariant tensors can be used directly inside the loop, as shown for `FooLayer`.



As explained later, a loop can have multiple `IIteratorLayer`, `IRecurrenceLayer`, and `ILoopOutputLayer` and, at most, two `ITripLimitLayers`. A loop with no `ILoopOutputLayer` has no output and is optimized by TensorRT.

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 *Nesting and Loops*).

To define a loop, first create an `ILoop` object using the `INetworkDefinition method::addLoop`. Then, add the boundary and interior layers. The rest of this section describes the features of the boundary layers, using a *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}` on 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 matrix column.
- `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 an `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, a 0D INT32 shape tensor specifying the length of the new output dimension. When the length exceeds the number of iterations, the extra elements contain arbitrary values. The second input, such as `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 usually comes from inside. For example, the TensorRT analog of this C++ fragment:

```
for (int32_t i = j; ...; i += k) ...
```

These calls could create this, 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.

12.2. Formal Semantics

TensorRT has applicative semantics, meaning there are no visible side effects besides 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 $\langle X_0, X_1, X_2, ... \rangle$. Elements of the sequence are evaluated lazily, meaning as needed.

The output from `IIteratorLayer(X)` is $\langle X\left[0\right], X\left[1\right], X\left[2\right], ... \rangle$ where `X[i]` denotes subscripting on the axis specified for the `IIteratorLayer`.

The output from `IRecurrenceLayer(X,Y)` is $\langle \mathrm{X}, Y_0, Y_1, Y_2 ... \rangle$.

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 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 subtensors 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` 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 $F\left(X, Y\right) = \langle f\left(X_0, Y_0\right), f\left(X_1, Y_1\right), f\left(X_2, Y_2\right) \rangle$. If a tensor comes from outside the loop, that is, a loop invariant, then the sequence for it is created by replicating the tensor.

12.3. 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.

12.4. 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.

12.5. Replacing `IRNNv2Layer` with Loops

`IRNNv2Layer` was deprecated in TensorRT 7.2.1 and was removed in TensorRT 10.0. Use the loop API to synthesize a recurrent sub-network. For an example, refer to sampleCharRNN, method `SampleCharRNNLoop::addLSTMCell`. Using the loop API, you can express general recurrent networks instead of being limited to the prefabricated cells in `IRNNLayer` and `IRNNv2Layer`.

See also

Working with Transformers
Autoregressive decoding and attention layers that use loop constructs.

Working with Dynamic Shapes
Dynamic input dimensions, which interact with loop trip counts and shape tensors.

Chapter 13. Working with Conditionals

NVIDIA TensorRT supports conditional if-then-else flow control. TensorRT conditionals are used to implement conditional execution of network subgraphs.

13.1. Defining A Conditional

Conditional boundary layers define an if-conditional:

- `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 boundary layer 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 the `condition` evaluates to `true`, and a `falseSubgraph`, which is executed when the `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 in a way similar to the ternary operator in many programming languages.

To define an if-conditional, create an `IIfConditional` instance with `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 the branch 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** the same type and shape. Each branch can independently include zero or more inputs.

  `IIfConditionalInputLayer` is optional and is used to control the layers that 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*).

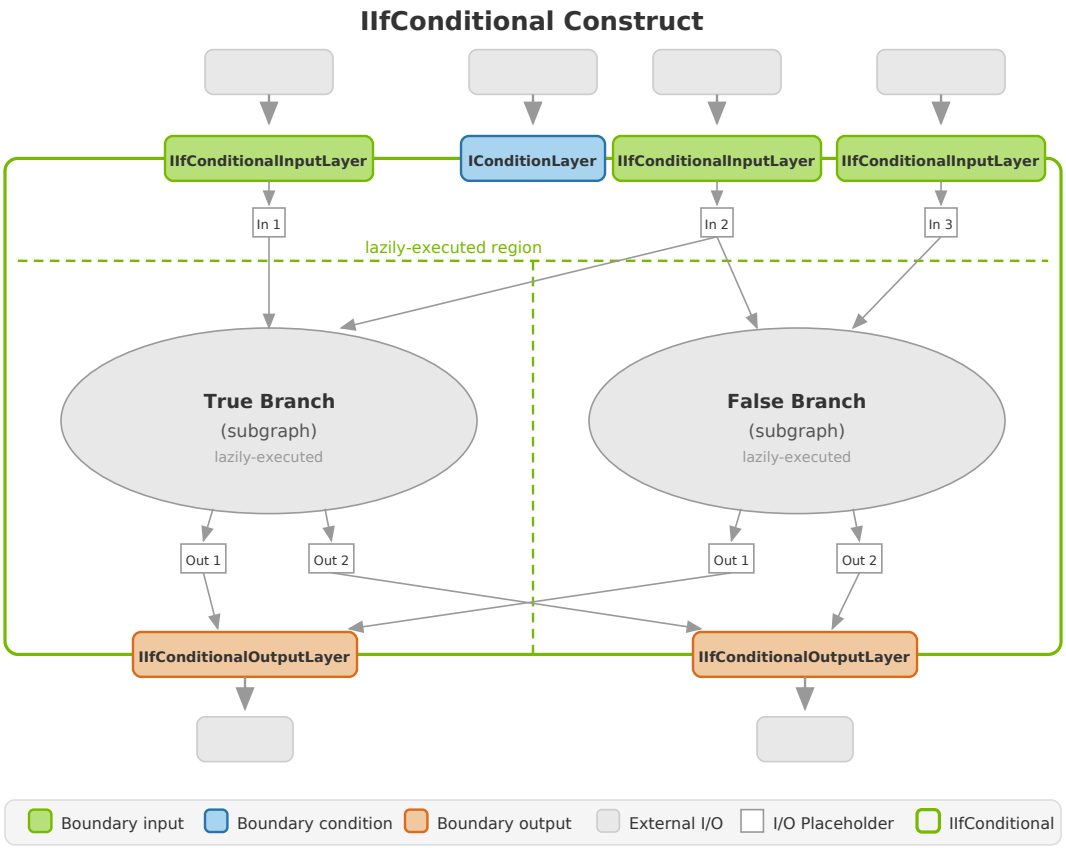
- An `IIfConditionalOutputLayer` abstracts a single output of the if-conditional. It has two inputs: an output from the `trueSubgraph` (input index 0) and an output from the `falseSubgraph` (input index 1). The output of an `IIfConditionalOutputLayer` can be considered a placeholder for the final output that will be determined during runtime.

  `IIfConditionalOutputLayer` serves a role similar to that of a (Phi) function node in traditional SSA control-flow graphs. Its semantics are: choose either the output of the `trueSubgraph` or `falseSubgraph`.

  ```
  // 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 does not affect the rest of the network. Therefore, it is considered ill-formed. Each branch (subgraph) 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.



13.2. Conditional Execution

Conditional execution of network layers is a network evaluation strategy where 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 or false branches are 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).

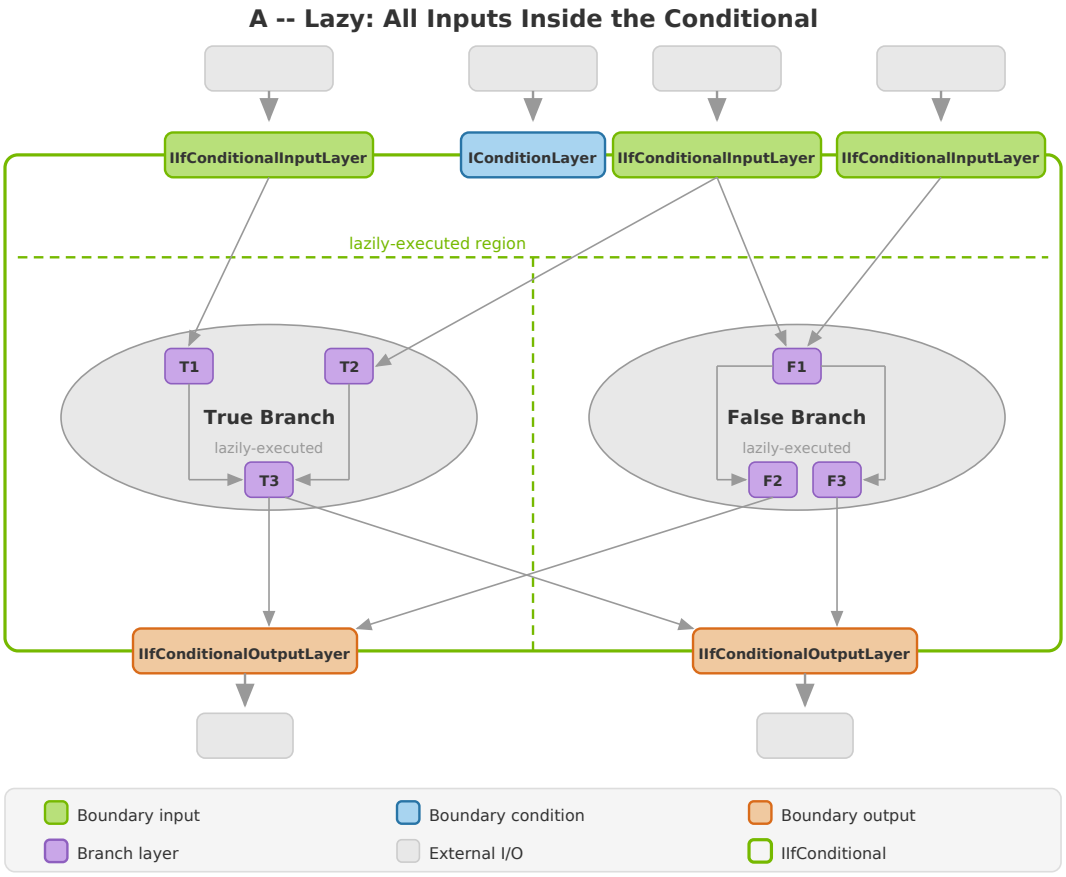
Conditional execution is sometimes called *lazy evaluation*, and predicated execution is sometimes called *eager evaluation*.

Instances of `IIfConditionalInputLayer` can be used to specify the layers that are invoked eagerly and the layers that are invoked lazily. This is done by tracing the network layers backward, starting with each conditional output. 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 layers are executed eagerly, similarly to `ISelectLayer`.

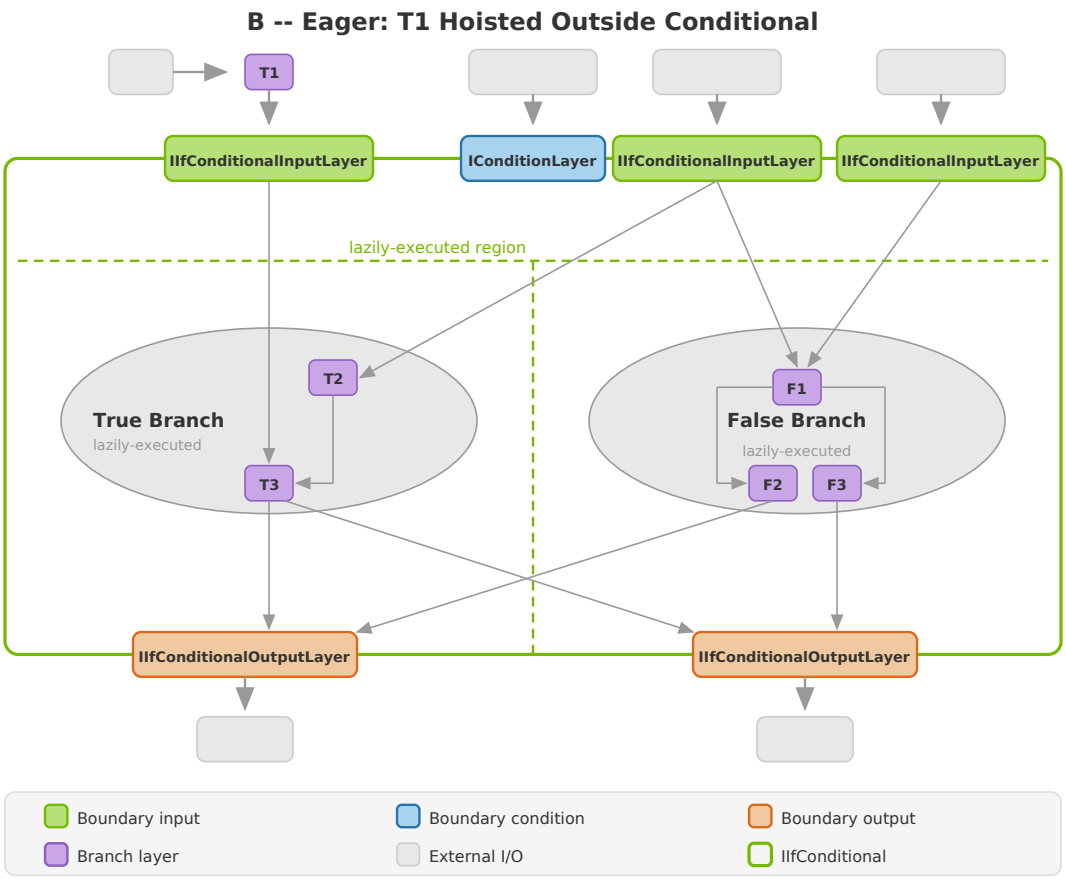
The three diagrams below depict how the choice of `IIfConditionalInputLayer` placement controls execution scheduling.

In diagram A, the true branch comprises three layers (T1, T2, T3). These layers execute lazily when the

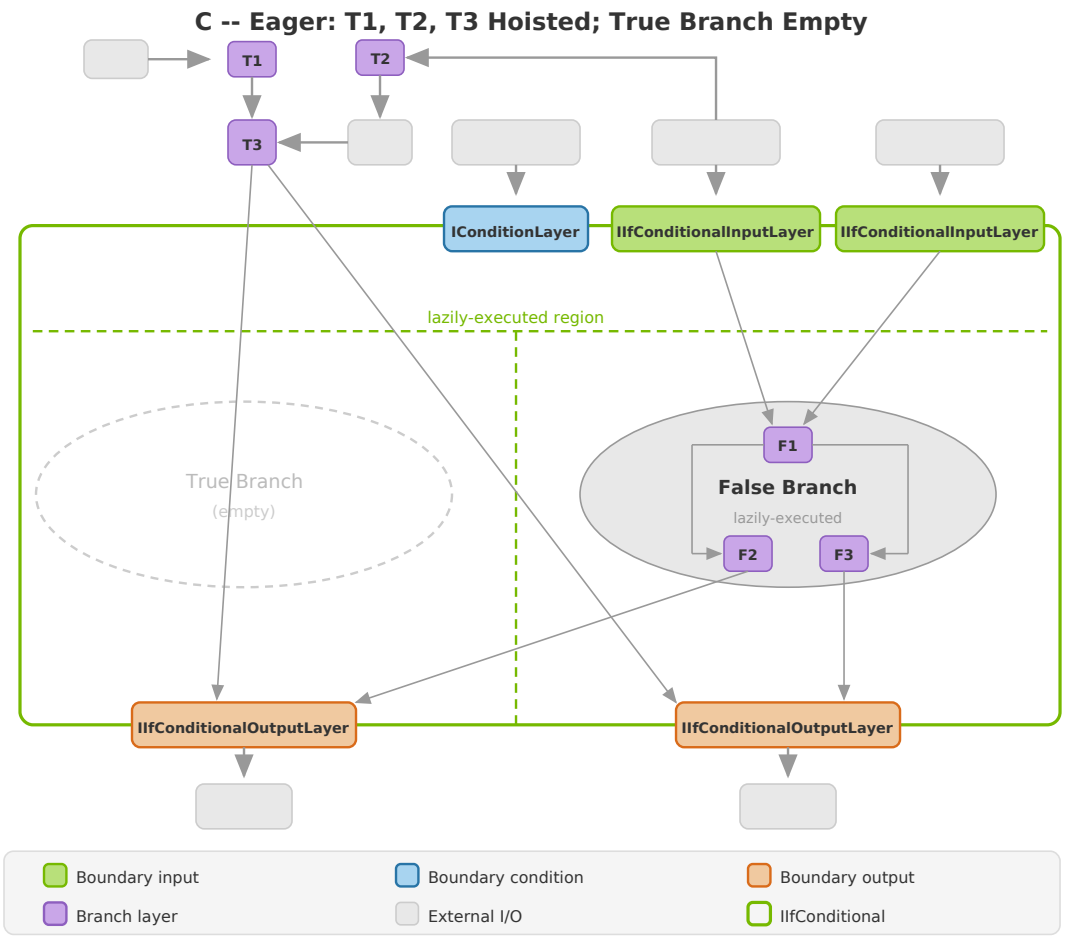
condition evaluates to `true`.



In diagram B, `T1` is placed before the `IIfConditionalLayer`, which moves `T1` out of the true branch. Layer `T1` executes eagerly before evaluating the if-construct.



In diagram C, the left `IIfConditionalLayer` is removed, 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).



13.3. Nesting and Loops

Conditional branches can nest other conditionals and can also nest loops. Loops can 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 the true branch to the false branch layers, and vice versa. In other words, the outputs of one branch cannot depend on layers in the other branch.

Refer to the *Conditional Examples* section for an example of how nesting can be specified.

13.4. Limitations

The number of output tensors in both true/false subgraph branches must be the same. The type and rank (number of dimensions) of each branch output tensor must be the same; as of TensorRT 10.11, the per-dimension shapes themselves may differ between branches.

Note that this is still slightly more constrained than the ONNX specification, which requires only that the true/false subgraphs have the same number of outputs and use the same output types.

13.5. Conditional Examples

13.5.1. 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 does not 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);

```

(continues on next page)

(continued from previous page)

```
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);
```

13.5.2. 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)

@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()
    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)
```

(continues on next page)

(continued from previous page)

```
export_to_onnx()
build_engine("example.onnx")
```

Chapter 14. TensorRT API Capture and Replay

TensorRT API Capture and Replay streamlines the process of reproducing and debugging issues within your applications. It allows you to record the engine-building phase of an application and later replay the engine-building steps, without needing to re-run the original application or access the model's source code.

This process is facilitated by two key components:

Capture Shim (`libtensorrt_shim.so`): This is a library that you can preload or drop into your application. It works by intercepting all TensorRT API calls made during the network-build phase. These intercepted calls, along with any associated constants, are then saved as a pair of files: a JSON file for the API calls and a BIN file for the constants.

Player (`tensorrt_player`): This is a standalone executable that takes the recorded JSON and BIN files generated by the Capture Shim and uses them to rebuild the TensorRT engine. This means you can recreate the engine that was built during the original application run, differing only in details related to timing differences during auto-tuning, aiding significantly in troubleshooting and debugging. Or, use it to recreate an engine-build failure.

14.1. Getting Started

Important

The capture-replay feature is currently restricted to Linux.

There are two ways to run the capture step:

- **LD_PRELOAD approach** - simplest method, works for most applications.
- **Drop-in replacement approach** - required for `trtexec` and any application that dynamically loads TensorRT using `dlopen`/`dlsym`.

LD_PRELOAD (recommended)

Preload the Capture Shim into your application. If your application uses `dlopen` and `dlsym` to load the TensorRT library and map its C-functions (exposed by `extern "C"`) to the process address space, use the drop-in replacement approach instead.

```
export TRT_SHIM_NVINFER_LIB_NAME=<path to libnvinfer.so> # optional
export TRT_SHIM_OUTPUT_JSON_FILE=<output JSON path>
LD_PRELOAD=libtensorrt_shim.so <your-application-command-line>
```

Drop-in replacement

Replace the `libnvinfer.so` library with the Capture Shim and redirect it to the original library. This approach is required for `trtexec` and any application that dynamically loads TensorRT. Capturing the TensorRT Python API and Python ONNX parser does not require this approach.

```
mv <path to the TRT lib that the app loads>/libnvinfer.so.<major version>
 ↪libnvinfer_orig.so.<major version>
cp build/x86_64-gnu/libtensorrt_shim.so <path to the TRT lib that the app
 ↪loads>/libnvinfer.so.<major version>
TRT_SHIM_OUTPUT_JSON_FILE=<JSON file path> TRT_SHIM_NVINFER_LIB_NAME=<path to
 ↪the original libnvinfer.so>/libnvinfer_orig.so.<major version>
```

Player

```
tensorrt_player -j <output JSON file> -o <output engine file>
```

When replaying models that use TensorRT plugins, set `LD_PRELOAD` to the plugin library paths to load them.

14.2. Multi-Network Support

TensorRT API Capture and Replay supports capturing multiple networks within a single process. This capability allows you to record the engine-building phase for applications that create and build multiple TensorRT networks, such as applications with ensemble models or multi-stage inference pipelines.

How It Works

- Each network created using `createNetworkV2` is assigned a unique network ID.
- Objects (tensors, layers, etc.) are tracked per-network to ensure proper isolation.
- Weights are stored with network-scoped identifiers, allowing the same weight pointer to be reused across different networks without conflicts.
- The player automatically manages network context during replay.

Capture Example

The following example demonstrates capturing multiple networks, including interleaved operations:

```
import tensorrt as trt

logger = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)

# Create first network
network1 = builder.create_network()
input1 = network1.add_input("input1", trt.float32, (1, 3, 224, 224))
# ... add layers ...
```

(continues on next page)

(continued from previous page)

```
network1.mark_output(output1)

# Create second network (interleaved operations are supported)
network2 = builder.create_network()
input2 = network2.add_input("input2", trt.float32, (1, 64))
# ... add layers ...
network2.mark_output(output2)

# Build both networks
config1 = builder.create_builder_config()
engine1 = builder.build_serialized_network(network1, config1)

config2 = builder.create_builder_config()
engine2 = builder.build_serialized_network(network2, config2)
```

Replay with Multiple Networks

When replaying a multi-network capture, the player generates one engine file per network. The output files are named with indices appended:

```
tensorrt_player -j capture.json -o output.engine
```

This produces:

- `output.engine0` – Engine for the first network
- `output.engine1` – Engine for the second network, and so on.

Best Practices

- Use descriptive output file names when capturing multiple networks to help identify the capture session that produced the files.
- If you need to replay only specific networks, consider capturing them in separate processes or sessions.
- Verify that all required plugins are available when replaying, as each network may use different custom layers.

14.3. Capture Tool Configuration

Table 14.1: Optional Environment Variables

Environment Variables	Description	Type	Default Value
TRT_SHIM_OUTPUT_JSON_FILE	Path to save the captured `.json` file. A corresponding `.bin` file will be created in the same directory.	stri	capture.json
TRT_SHIM_NVINFER_LIB_NAME	Intercepted TensorRT library name. If unset, `dlopen(<lib_name>)` finds the library using its normal search rules (a full path is also allowed).	stri	libnvinfer.so
TRT_SHIM_DUMP_API	Print `enter` and `exit` messages for every API function call.	bool	false
TRT_SHIM_PRINT_WELCOME	Print `Welcome` to `TensorRT Shim` at the start of the run.	bool	false
TRT_SHIM_FORCE_SINGLE_THRE	Lock every API call to enforce single-threaded execution. Ignored when TRT_SHIM_OUTPUT_JSON_FILE is set.	bool	false
TRT_SHIM_INLINE_WEIGHTS_L(	Inline weights into the `.json` instead of a separate `.bin` when their size (in elements) is this threshold.	int	8
TRT_SHIM_MARK_AS_RANDOM_WE	Skip saving weights with an element count this threshold (they will be marked as random).	int	Max int
TRT_SHIM_FLUSH_AFTER_EVER'	Flush captured calls to the file after every API call instead of aggregating them.	bool	false
TRT_SHIM_FLUSH_ON_BUILD	Flush captured calls to the file after each `buildSerializedNetwork` call.	bool	false
TRT_SHIM_FLUSH_ON_BUILD	Flush captures calls to the file after every `buildSerializedNetwork` call. Useful for debugging crashes during engine building or capturing incremental progress in multi-network sessions.	bool	false
TRT_SHIM_SET_TACTIC_CACHE	Path to a tactic-cache file that will be loaded and applied to TensorRT's `IBuilderConfig` so tactic selection stays consistent across runs.	stri	""

14.4. Known Limitations

Supported

- Linux x86_64 and AArch64 platforms.
- C++ plugins of type `PluginV2` and `PluginV3`, registered statically by calling `REGISTER_TENSORRT_PLUGIN`.

Not supported

- Dynamic plugin registration using `registerCreator()`.
- Python plugins.
- Plugins shipped as part of the engine (that is, `config->setPluginsToSerialize` is not supported); the plugin must be shipped externally.
- The `trtexec --saveEngine` flag.
- The `buildSerializedNetworkToStream` and `buildSerializedNetworkWithKernelText` functions are not captured.

14.5. Flush Behavior

The captured data is flushed (written to disk) under any of the following conditions:

- **Process exit** - flush occurs automatically when the process completes.
- **Every API call** - flush after each captured call when `TRT_SHIM_FLUSH_AFTER_EVERY_CALL` is set.
- **On build** - flush after each `buildSerializedNetwork` call when `TRT_SHIM_FLUSH_ON_BUILD` is set.

On each flush, the entire `.json` file is overwritten, while the `.bin` file is appended.

See also

Engine Inspector and Debug Tensors
Additional tools for inspecting and debugging TensorRT engines.

Troubleshooting
Error resolution and diagnostic guidance.

Chapter 15. Working with Transformers

15.1. Rotary Position Embedding

TensorRT includes built-in support for RoPE (Rotary Position Embedding) for transformers to make it easier to express RoPE and convert ONNX models with the `IRotaryEmbeddingLayer` API to TensorRT.

The `IRotaryEmbeddingLayer` has three inputs, one optional input, and two attributes:

Inputs

- `(index 0) input`: The input activation tensor with shape `[B, N, S, H]`
- `(index 1) cosCache`: The cosine values for calculating the rotary embedding
- `(index 2) sinCache`: The sine values for calculating the rotary embedding

`cosCache` and `sinCache` should have the shape `[B, S, H / 2]`.

Optional input

- `(index 3) positionIds`: Position IDs for indexing into `cosCache` and `sinCache`

`positionIds` should have shape `[B, S]`. When `positionIds` is provided, `cosCache` and `sinCache` should correspondingly have shape `[maxPositionId + 1, H / 2]`.

Attributes

- `interleaved`: A boolean that specifies whether the input tensor is in `interleaved` format, that is, whether the 2d vectors rotated are taken from adjacent 2 elements in the hidden dimension.
- `rotaryEmbeddingDim`: An integer specifying the hidden dimension that participates in RoPE. A special value of 0 means the full hidden dimension participates in RoPE. If it is not `0`, then the last dimension of `cosCache` and `sinCache` should correspondingly be `rotaryEmbeddingDim / 2` instead of `H / 2`.

The `IRotaryEmbeddingLayer` has one output, which is the output activation tensor. It has the same shape and format as `(index 0) input`: the `input` activation tensor with shape.

The `IRotaryEmbeddingLayer` is supported by all SM versions.

15.2. KV Cache

TensorRT supports using KV cache when doing LLM inference with transformers using the `IKVCacheUpdateLayer`.

The `IKVCacheUpdateLayer` has three inputs, one output, and two attributes.

Inputs

- `(index 0) cache`: The K/V cache tensor with shape `[B, N, S_max, H]`. Allocate this tensor and provide it as an input to the TensorRT network.
- `(index 1) update`: The newly calculated K or V tensor with dimension: `[B, N, S_new, H]`.
- `(index 2) writeIndices`: A tensor of size `[B]` represents where to start to write K/V updates for each sequence.

Output

The K/V cache tensor has a shape `[B, N, S_max, H]`.

Attributes

- `cacheMode`: An enum that specifies the K/V cache update strategy. Currently, only `kLINEAR` mode is supported, which performs sequential updates to the cache based on the provided write indices.

Note

`IKVCacheUpdateLayer` does not own or allocate the cache memory. Memory management is handled by higher-level frameworks (such as TensorRT Edge-LLM or your application).

Memory Requirements: The cache input tensor and cache output tensor must share the same device memory to enable in-place updates. Use the `context->setTensorAddress()` API to explicitly set both tensors to the same memory address. If different addresses are assigned to these tensors, TensorRT will report an API error.

TensorRT also includes a public engine API `getAliasedInputTensor(char const* outputTensorName)`.

`IKVCacheUpdateLayer` has the same hardware support matrix as `IAttention`.

15.3. MoE (Mixture of Experts)

TensorRT includes built-in support for MoE (Mixture of Experts) in transformer models using the `IMoELayer` API.

The `IMoELayer` is currently only supported on SM110.

Inputs

- `(index 0) hiddenStates`: Activation tensor with shape `[B, S, H]`
- `(index 1) selectedExpertsForTokens`: Indexes of chosen experts per token, with shape `[B, S, topK]`
- `(index 2) scoresForSelectedExperts`: Scores per chosen expert for the weighted output, with shape `[B, S, topK]`

The following inputs are set using `setGatedWeights`. Each tensor corresponds to a layer in the expert's GLU (gated linear unit), where `I` is the internal hidden size within each expert.

- `fcGateWeights`: GLU gate weights, with shape `[numExperts, H, I]`
- `fcUpWeights`: GLU up-projection weights, with shape `[numExperts, H, I]`
- `fcDownWeights`: GLU down-projection weights, with shape `[numExperts, I, H]`

The following optional inputs are set using `setGatedBiases`:

- `fcGateBias`: GLU gate biases, with shape `[numExperts, I]`
- `fcUpBias`: GLU up-projection biases, with shape `[numExperts, I]`
- `fcDownBias`: GLU down-projection biases, with shape `[numExperts, H]`

The following input is set using `setQuantizationStatic`:

- `fcDownActivationScale`: Quantization scales for the `fcDown` activation input (mul output), with shape `[]` or `[numExperts]`

Attributes

- `activationType`: Activation type in the expert GLU. Supported values: `None` and `SiLU`. Set using `setGatedWeights`.
- `quantizationToType`: Type to quantize the mul output (`fcDown` input) to. Currently only FP8 is supported. Set using `setQuantizationStatic`.

Note

- The following attributes and inputs are not yet supported: `quantizationBlockShape`, `dynQOutputScaleType`, `fcDownActivationDblQScale` (set using `setQuantizationDynamicDblQ`), and the SwiGLU parameters (`swigluParamLimit`, `swigluParamAlpha`, `swigluParamBeta`).
- Performance may be limited when the sequence length (S) exceeds 16.

Output

The `IMoELayer` produces one output with the same shape as `hiddenStates` (`[B, S, H]`), representing the weighted sum of the top-K expert outputs.

Supported Configuration

Parameter	Requirement
Hardware	SM110
Weights	NVFP4 double quantized. Weights pass through a DQ layer (with scales from a second DQ), then optionally a `Transpose`, before the MoE layer. Scales must be FP8.
`hiddenStates` activation input	FP8 static quantization using a DQ layer before the MoE layer.
Internal mul output (`fcDown` input)	FP8 static quantization. Set using `setQuantizationStatic` with `quantizationToType` set to FP8.
SwiGLU parameters	Not supported

15.4. Fused Attention

TensorRT supports two different methods to trigger Attention fusion:

- Adding an attention operator to the network using `IAttention` API.

- Constructing an attention graph using primitive `INetwork` layers like `IMatrixMultiplyLayer` and `ISoftMaxLayer`.

Attention computes `softmax(Q * K  mask) * V`, where:

- `Q` is query embedding
- `K` is key embedding
- `V` is value embeddings
- `mask` can be:
  - A boolean mask indicating that the corresponding position is allowed to attend by Softmax.
  - An add mask offsetting `Q * K`

Note

TensorRT requires `Q` or `K` to be multiplied with $attentionscale$ or both `Q` and `K` to be multiplied with $\sqrt{attentionscale}$ before going into `IAttention` or the first Batched Matrix Multiply (BMM) `IMatrixMultiplyLayer` for numeric stability and performance reasons.

The shape of `Q is [B, N_q, S_q, H]`, and the shapes of `K` and `V` are `[B, N_kv, S_kv, H]`, where:

- `B` is batch size
- `N_q` and `N_kv` are the numbers of attention heads of the query and key/value, respectively
- `S_q` and `S_kv` are the sequence lengths of the query and key/value, respectively.
- `H` is the head/hidden size

Multi-Head Attention (MHA, `N_q == N_kv`), Group-Query Attention (GQA, `N_q % N_kv == 0`) and Multi-Query Attention (MQA, `N_kv == 1`) fusion are all supported.

We highly recommend tailoring your model to the said restrictions in the following tables, so that Attention fusion happens. It is important because Attention fusion optimizes large sequence length cases by significantly reducing memory footprint from `O(S^2)` to `O(S)`, where `S` is the sequence length. On top of that, it shares the common performance benefits of operator fusion, that is, reduced memory traffic, better hardware utilization, less kernel launch, and synchronization overhead.

Table 15.1: Supported Attention Fusions on SM100, SM110

Feature	FP16	BF16	FP8
SM Version (V)	▶ SM100 ▶ SM110	▶ SM100 ▶ SM110	▶ SM100 ▶ SM110
Head Size (H)	▶ `8 ≤ H ≤ 128` ▶ `H % 8 ==0`	▶ `8 ≤ H ≤ 128` ▶ `H % 8 ==0`	▶ `16 ≤ H ≤ 128` ▶ `H % 16 ==0`
Sequence Length (`S_q`, `S_kv`)	No restriction	No restriction	No restriction
Quantization	Not required	Not required	Specify Q/DQ layers in the Attention fusion pattern for FP8.
Accumulation Precision (`BMM1`)	FP32	FP32	FP32
Accumulation Precision (`BMM2`)	FP32	FP32	FP32
Supported Mask Type	1d or 2d vector or scalar	1d or 2d vector or scalar	1d or 2d vector or scalar
Pointwise op	Activation, Constant, Elementwise (including SiLU), Scale, and Unary	Activation, Constant, Elementwise (including SiLU), Scale, and Unary	Activation, Constant, Elementwise (including SiLU), Scale, and Unary

Table 15.2: Supported Attention Fusion for Other SM Versions

Feature	FP16	BF16	INT8	FP8
SM Version (V)	▶ SM75 ≤ V ≤ SM90 ▶ SM120 ▶ SM121	▶ SM80 ≤ V ≤ SM90 ▶ SM120 ▶ SM121	▶ SM75 ≤ V ≤ SM90 ▶ SM120 ▶ SM121	▶ SM89 ▶ SM90 ▶ SM120 ▶ SM121
Head Size (H)	▶ 16 ≤ H ≤ 256 ▶ H % 8 ==0	▶ 16 ≤ H ≤ 256 ▶ H % 8 ==0	▶ 16 ▶ 32 ▶ 64	▶ 32 ≤ H ≤ 256 ▶ H % 16 == 0
Sequence Length (S_q, S_kv)	No restriction	No restriction	S_{q,kv} ≤ 512	No restriction
Quantization	Not required	Not required	Specify Q/DQ layers in the Attention fusion pattern for FP8 and INT8.	Specify Q/DQ layers in the Attention fusion pattern for FP8 and INT8.
Accumulation Precision (BMM1)	▶ FP16 ▶ FP32	FP32	INT32	FP32
Accumulation Precision (BMM2)	▶ FP16 ▶ FP32 (if ≥ SM90 and S_q = S_kv)	FP32	INT32	FP32
Supported Mask Type	Any masking (such as the Select operator in TensorRT)	Any masking (such as the Select operator in TensorRT)	Any masking (such as the Select operator in TensorRT)	Any masking (such as the Select operator in TensorRT)
Pointwise op	Activation, Constant, Elementwise (including SiLU), Pointwise (single input), Scale, and Unary	Activation, Constant, Elementwise (including SiLU), Pointwise (single input), Scale, and Unary	Activation, Constant, Elementwise (including SiLU), Pointwise (single input), Scale, and Unary	Activation, Constant, Elementwise (including SiLU), Pointwise (single input), Scale, and Unary

Note

FP8 fused Attention is expected to outperform FP16 fused Attention when the head size (H) ≥ 128 or the sequence length S_{q,kv} ≥ 128.

Note

Numerical issues in multi-head attention implementations

Attention implementations that use online softmax (for example, Flash Attention) can produce NaN outputs with certain attention masks.

- The attention kernel guards against NaN values at critical computation points.
- Multiply+Add to fused-multiply-add optimization is disabled where unsafe, consistent with other frameworks.

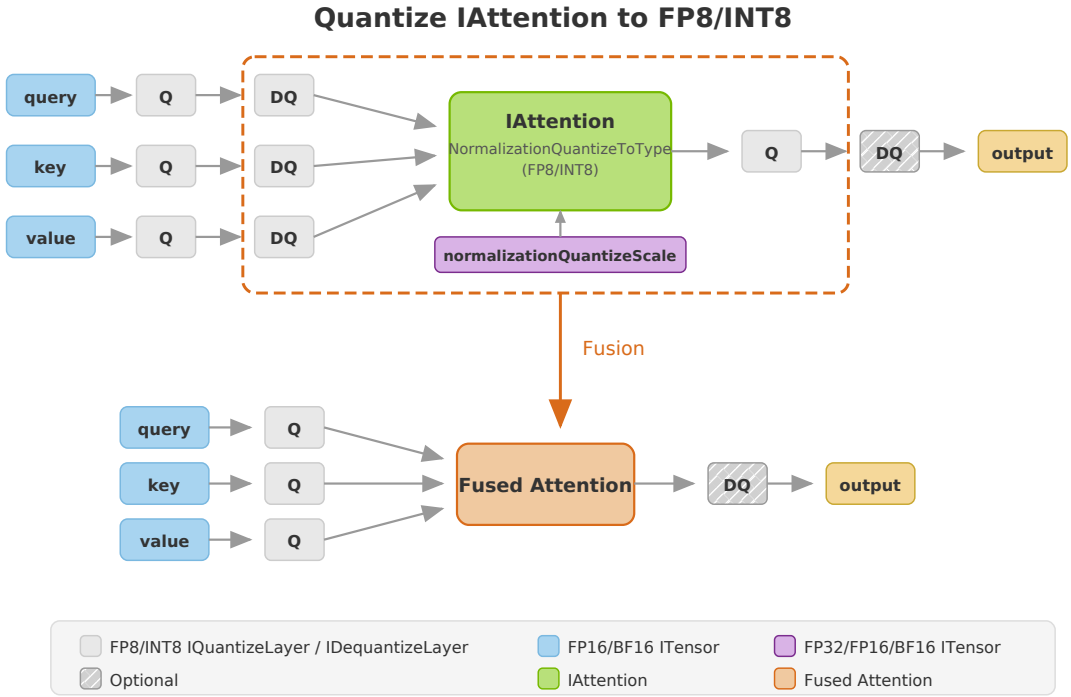
These mitigations may reduce performance by 2–3%, or up to 5–10% in worst cases.

Method 1 (Recommended): Use IAttention API

`IAttention` offers a direct API to incorporate an attention to the network and trigger the Attention fusion in TensorRT.

`IAttention` by default assumes the attention should always use a fused kernel. If the added attention does not comply with the restrictions listed in the tables above, and thus a fused kernel cannot be utilized, the engine build will raise an error. If users want to allow `IAttention` to fallback to use non-fused kernels, users can set the `IAttention` to be decomposable by calling the method `IAttention::setDecomposable`.

To quantize an `IAttention`, Q/DQ pairs following *Explicit Quantization* rules must be used for all query, key and value inputs with another normalization quantize scale and a normalization quantize-to data type supplied to `IAttention` using `IAttention::setNormalizationQuantizeScale` and `IAttention::setNormalizationQuantizeToType` respectively. The normalization quantize scale and the normalization quantize-to data type have to match the data types of the Q/DQ pairs. The output of the `IAttention` can be quantized with an optional quantizer.



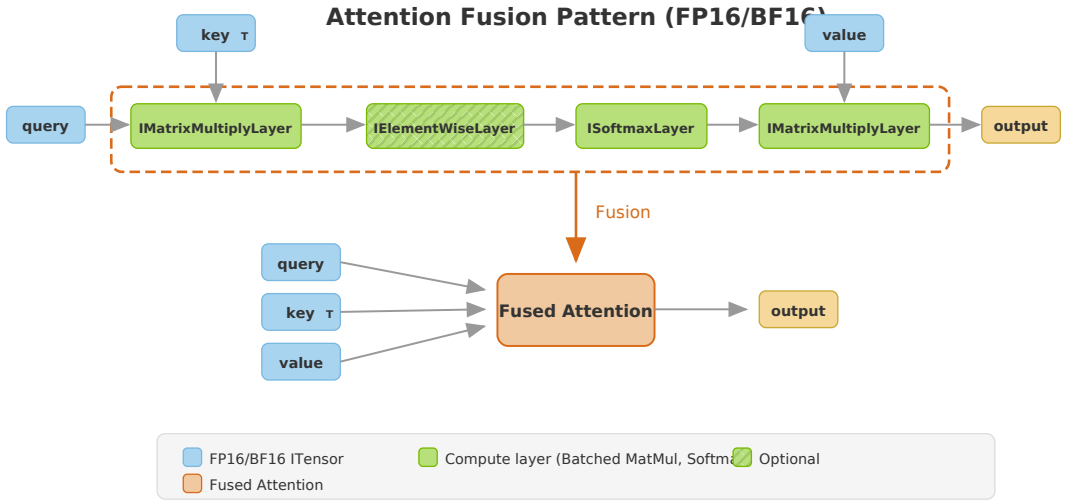
To apply masks with `IAttention`, use `IAttention::setCausalKind`

with `CausalMaskKind::kNONE`, `CausalMaskKind::kUPPER_LEFT`, or `CausalMaskKind::kLOWER_RIGHT` for implicit causal masking, or `IAttention::setMask` for arbitrary masks (boolean or float), the legacy `IAttention::setCausal(bool)` helper is deprecated and now internally maps true to `kUPPER_LEFT` and `false` to `kNONE`.

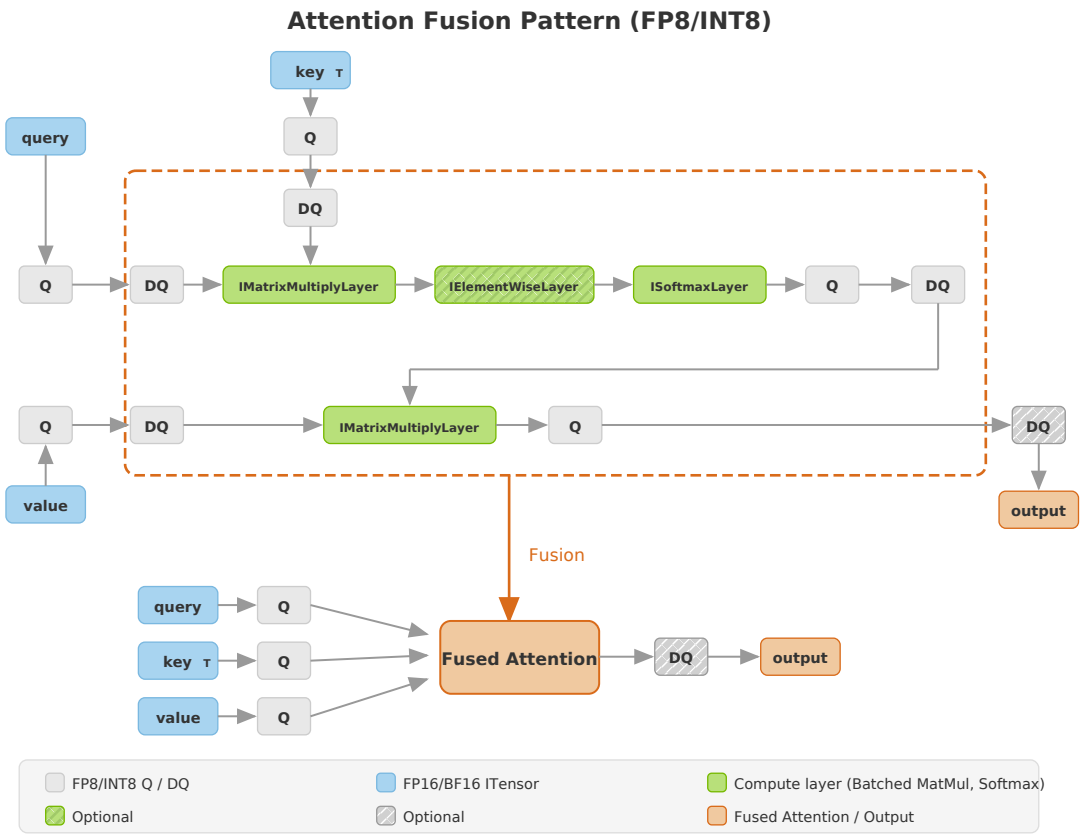
To add a Sage Attention and trigger the fusion, follow the paradigm in the above figure and use `IDynamicQuantizeLayer` in place of `IQuantizeLayer` after query, key, value.

Method 2: Construct an Attention Graph with Primitive ``INetwork`` Layers

The following figures demonstrate how FP16, BF16, FP8, and INT8 attentions can be constructed to trigger Attention fusion. Batched MatMul uses `IMatrixMultiplyLayer`, Softmax uses `ISoftMaxLayer` and Q/DQ uses `IQuantizeLayer` and `IDequantizeLayer`.



TensorRT chooses the accumulation precision by default based on the input types and performance considerations. However, you can also control accumulation precision (refer to *Control of Computational Precision*).



The Attention fusion captures common pointwise operators in series in Attention as mentioned in the *pointwise operation list*. It also covers Q/DQ fusion following Attention for certain quantization and architecture (such as FP16/BF16 to FP8/INT8 on NVIDIA Ampere GPU architecture).

For method 2, TensorRT can decide not to fuse an Attention graph into a single kernel based on performance evaluations or other constraints.

15.4.1. Example Workflow: FP8 MHA Fusion

Assume you have an ONNX model, vit_base_patch8_224_Opset17.onnx, and calibration data, calib.npy, on your local machine.

1. Install the TensorRT model optimizer.

```
pip3 install --no-cache-dir --extra-index-url https://pypi.nvidia.com
↪nvidia-modelop
```

2. Quantize a model with TensorRT model optimizer. For more information, refer to these detailed instructions.

```
python3 -m modelopt.onnx.quantization \
 --onnx_path=vit_base_patch8_224_Opset17.onnx \
 --quantize_mode=<fp8|int8> \
 --calibration_data=calib.npy \
 --calibration_method=<max|entropy> \
 --output_path=vit_base_patch8_224_Opset17.quant.onnx
```

For example, to quantize the model using FP8 and entropy and store as vit_base_patch8_224_Opset17.quant.onnx, use:

```
python3 -m modelopt.onnx.quantization \
 --onnx_path=vit_base_patch8_224_Opset17.onnx \
 --quantize_mode=fp8 \
 --calibration_data=calib.npy \
 --calibration_method=entropy \
 --output_path=vit_base_patch8_224_Opset17.quant.onnx
```

3. Compile the quantized model with TensorRT.

```
trtexec --onnx=vit_base_patch8_224_Opset17.quant.onnx \
--saveEngine=vit_base_patch8_224_Opset17.engine \
--stronglyTyped --skipInference --profilingVerbosity=detailed
```

4. Run the quantized model with TensorRT.

```
trtexec --loadEngine=vit_base_patch8_224_Opset17.engine \
--useCudaGraph --noDataTransfers --useSpinWait
```

Add the following options if you want to check if Attention is fused. Attention should be fused if you find the `mha` op in the `output.log` file.

```
trtexec --loadEngine=vit_base_patch8_224_Opset17.engine \
--profilingVerbosity=detailed --dumpLayerInfo --skipInference &> output.
↪log
```

Tip

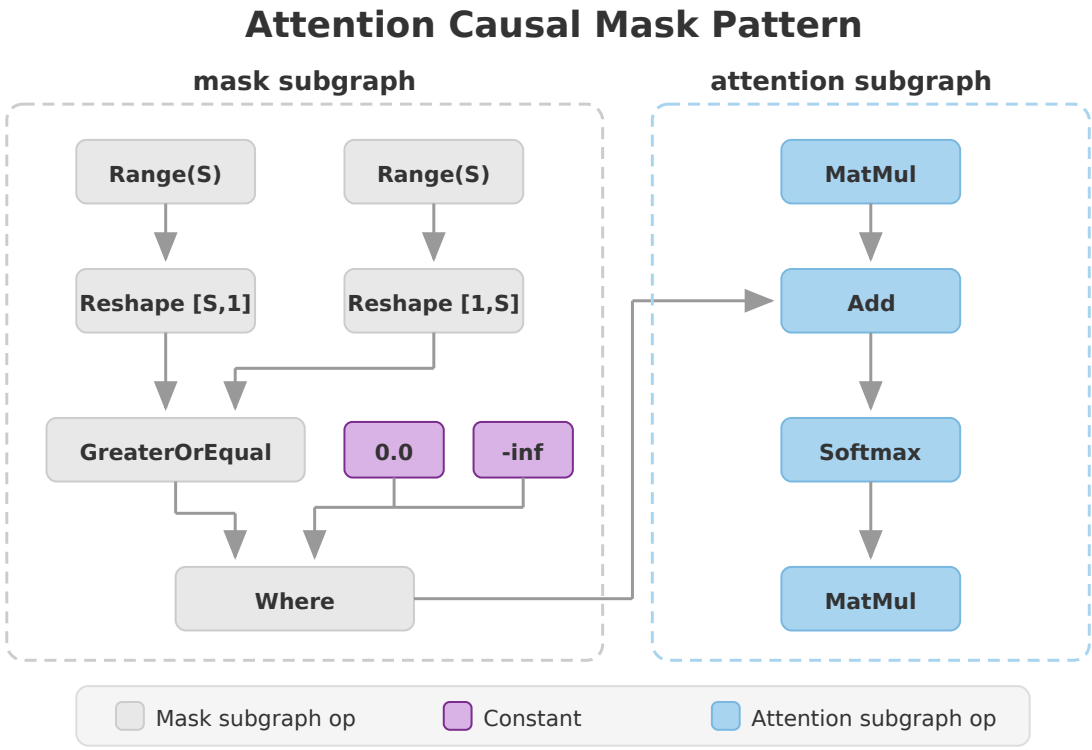
There are two ways to set the accumulation data type to FP32:

1. Manually set computational precision. For more information, refer to these *detailed instructions*.
2. Convert your ONNX model using TensorRT Model Optimizer, which adds the Cast ops automatically.
   - If the Attention has a head size (H) that is not a multiple of 16, for better performance do not add Q/DQ ops in the Attention to fall back to the FP16 or BF16 Attention.
   - Compare Attention fusion performance of INT8 with FP8.

15.4.2. Example: Exploiting Sparsity in Attention Masks

In limited cases, TensorRT can exploit block sparsity in attention masks to reduce the runtime of the attention computation. For example, an attention with a causal mask requires half the computation of a regular attention (since it only considers query tokens with index position in the sequence at or after a corresponding key token's index, for example, `q >= kv`), therefore, TensorRT can achieve up to 2x speedup on the attention computation.

In ONNX, such as here is how you would express a causal mask for primitive op constructed attention.



While TensorRT's overall approach to recognizing sparsity in masks is flexible, there are a few characteristics of this expression that must be present for TensorRT to consider such an optimization.

- The mask is expressed using indices along the sequence length (that is, through the `Range` op). A causal mask is written as `q >= kv`, therefore, we get the index, the sequence length, the query, key dimensions through the `Range` ops, and reshape them so they can be compared.
- The mask subgraph (that is, the subgraph of operations feeding into the `Add`, starting with the `Where`) must be expressed solely using pointwise operations, position expressions (such as `Trilu`, `Range`), or `Expand` operations (or equivalent `Reshape`, `Tile` operations).
- Along the chain of operations from the `MatMul` to the `Softmax`, there must be exactly one operation doing masking, in this case the `Add`. That operation with input dependent on the MatMul `vIn` and output `vOut` must obey the property that at a point `(q, kv)` in the attention computation:
  - If the mask is valid at that point, `vOut = vIn`.
  - If the mask is invalid at that point, `vOut = -inf`.
- The masking operation must be a `Where`, `Add`, or `Subtract` operation.

15.5. Example of Using Transformer-Oriented APIs

Examples of how `IKVCacheUpdateLayer` can be used together with `IAttention` and `IRotaryEmbeddingLayer`:

C++

```
ITensor* qInput = network->addInput("q", DataType::kHALF,
                          Dims4{-1, numQHeads, -1, headSize});
ITensor* kInput = network->addInput("k", DataType::kHALF,
                                    Dims4{-1, numKvHeads, -1, headSize});
ITensor* vInput = network->addInput("v", DataType::kHALF,
                                    Dims4{-1, numKvHeads, -1, headSize});

ITensor* kCacheInput = network->addInput("kCache", DataType::kHALF,
                                         Dims4{-1, numKvHeads, kvCacheCapacity,
↪headSize});
ITensor* vCacheInput = network->addInput("vCache", DataType::kHALF,
                                         Dims4{-1, numKvHeads, kvCacheCapacity,
↪headSize});
ITensor* ropeCosCache = network->addInput("ropeCosCache", DataType::kHALF,
                                         Dims2{maxPositionEmbeddings, headSize
↪/ 2});
ITensor* ropeSinCache = network->addInput("ropeSinCache", DataType::kHALF,
                                         Dims2{maxPositionEmbeddings, headSize
↪/ 2});
ITensor* positionIds = network->addInput("positionIds", DataType::kINT32,
                                         Dims2{-1, -1});
ITensor* kCacheIndices = network->addInput("kCacheIndices", DataType::kINT32,
                                           Dims{1, {-1}});
// Used to slice the present KV from the cache
ITensor* presentLengthInput = network->addInput("presentLength",
↪DataType::kINT32,
                                                Dims{1, {-1}});

// Apply RoPE to Q and K
IRotaryEmbeddingLayer* ropeQ = network->addRotaryEmbedding(*qInput,
↪*ropeCosCache, *ropeSinCache,
                                                           false, 0);  //
↪interleaved=false, rotaryEmbeddingDim=0 (use all)
ropeQ->setInput(3, *positionIds);
IRotaryEmbeddingLayer* ropeK = network->addRotaryEmbedding(*kInput,
↪*ropeCosCache, *ropeSinCache,
                                                           false, 0);
ropeK->setInput(3, *positionIds);

ITensor* ropeQTensor = ropeQ->getOutput(0);
ITensor* ropeKTensor = ropeK->getOutput(0);

// Update KV cache
KVCacheMode kvCacheMode = KVCacheMode::kLINEAR;
IKVCacheUpdateLayer* kCacheLayer = network->addKVCacheUpdate(*kCacheInput,
↪*ropeKTensor,
                                                             *kCacheIndices,
↪kvCacheMode);
IKVCacheUpdateLayer* vCacheLayer = network->addKVCacheUpdate(*vCacheInput,
↪*vInput,
                                                             *kCacheIndices,
```

(continues on next page)

(continued from previous page)

```
↪kvCacheMode);

ITensor* kCacheOutput = kCacheLayer->getOutput(0);
ITensor* vCacheOutput = vCacheLayer->getOutput(0);

// Mark cache outputs
kCacheOutput->setName("kCacheOutput");
network->markOutput(*kCacheOutput);
vCacheOutput->setName("vCacheOutput");
network->markOutput(*vCacheOutput);

// Create slice layers to extract present KV from cache
ITensor* kPresent = /* ... */;
ITensor* vPresent = /* ... */;

// Apply Q/K scale using elementwise multiplication
// Scale Q by sqrt(head size) to get the same effect as scaling QK^T by scale
float qkScale = 1.0f / std::sqrt(static_cast<float>(headSize));
half_float::half scaleData{qkScale};
IConstantLayer* scaleConst = network->addConstant(Dims4{1, 1, 1, 1}, Weights
↪{DataType::kHALF, &scaleData, 1});

IElementWiseLayer* scaledQ = network->addElementWise(*ropeQTensor,
↪*scaleConst->getOutput(0),

↪ElementWiseOperation::kPROD);
ITensor* scaledQTensor = scaledQ->getOutput(0);

// Add attention layer
IAttention* attention = network->addAttentionV2(*scaledQTensor, *kPresent,
↪*vPresent, AttentionNormalizationOp::kSOFTMAX, CausalMaskKind::kNONE);

attention->setDecomposable(true);

ITensor* attentionOutput = attention->getOutput(0);
attentionOutput->setName("attentionOutput");
network->markOutput(*attentionOutput);
```

Python

```
q_input = network.add_input("q", trt.float16,
                (-1, num_q_heads, -1, head_size))
k_input = network.add_input("k", trt.float16,
                        (-1, num_kv_heads, -1, head_size))
v_input = network.add_input("v", trt.float16,
                        (-1, num_kv_heads, -1, head_size))

k_cache_input = network.add_input("k_cache", trt.float16,
                                  (-1, num_kv_heads, kv_cache_capacity, head_
↪size))
v_cache_input = network.add_input("v_cache", trt.float16,
```

(continues on next page)

(continued from previous page)

```
                                    (-1, num_kv_heads, kv_cache_capacity, head_
↪size))

rope_cos_cache = network.add_input("rope_cos_cache", trt.float16,
                                    (max_position_embeddings, head_size // 2))
rope_sin_cache = network.add_input("rope_sin_cache", trt.float16,
                                    (max_position_embeddings, head_size // 2))

position_ids = network.add_input("position_ids", trt.int32, (-1, -1))
k_cache_indices = network.add_input("k_cache_indices", trt.int32,(-1, ))

# Used to slice the present KV from the cache
present_length_input = network.add_input("present_length", trt.int32, (-1, ))

# Apply RoPE to Q and K
rope_q = network.add_rotary_embedding(
    q_input, rope_cos_cache, rope_sin_cache, False,
    0)  # interleaved=False, rotary_ndims=0 (use all)
rope_q.set_input(3, position_ids)

rope_k = network.add_rotary_embedding(
    k_input, rope_cos_cache, rope_sin_cache, False, 0)
rope_k.set_input(3, position_ids)

rope_q_tensor = rope_q.get_output(0)
rope_k_tensor = rope_k.get_output(0)

# Update KV cache
kv_cache_mode = trt.KVCacheMode.LINEAR
k_cache_layer = network.add_kv_cache_update(k_cache_input,
                                            rope_k_tensor,
                                            k_cache_indices,
                                            kv_cache_mode)
v_cache_layer = network.add_kv_cache_update(v_cache_input,
                                            v_input,
                                            k_cache_indices,
                                            kv_cache_mode)

k_cache_output = k_cache_layer.get_output(0)
v_cache_output = v_cache_layer.get_output(0)

# Mark cache outputs
k_cache_output.name = "k_cache_output"
network.mark_output(k_cache_output)
v_cache_output.name = "v_cache_output"
network.mark_output(v_cache_output)

# Create slice layers to extract present KV from cache
k_present = ...
v_present = ...

# Apply Q/K scale using elementwise multiplication
```

(continues on next page)

(continued from previous page)

```
# Scale Q by sqrt(head size) to get the same effect as scaling QK^T by scale
qk_scale = 1.0 / (head_size**0.5)
scale_const = network.add_constant((1, 1, 1, 1),
                                   np.array([qk_scale],
                                   dtype=np.float16))
scaled_q = network.add_elementwise(rope_q_tensor,
                                   scale_const.get_output(0),
                                   trt.ElementWiseOperation.PROD)
scaled_q_tensor = scaled_q.get_output(0)

# Add attention layer
attention = network.add_attention_v2(scaled_q_tensor, k_present, v_present,
 ↪trt.AttentionNormalizationOp.SOFTMAX, trt.CausalMaskKind.NONE)
attention.decomposable = True
attention_output = attention.get_output(0)
attention_output.name = "attention_output"
network.mark_output(attention_output)
```

15.6. Multi-Device Attention

TensorRT supports multi-device attention through context parallelism, splitting the key-value sequence across multiple GPUs. Each GPU processes a portion of the sequence, and NCCL handles inter-device communication. Refer to the attention-mdtrt sample as an example.

- **Supported data types**: BF16 and FP16
- **Platform support**: Blackwell (SM 100) and later

For more information, refer to the GPU Architecture and Precision Support section of the TensorRT Support Matrix.

Note

There is no ONNX compliant operator for multi-device attention. However, standard ONNX files generated by a sharding tool are supported when the `Attention` op includes an `nbRanks` field. You can also use the C++ or Python API directly.

The `IAttention` layer runs in multi-device mode when `nbRanks > 1`. Use `IAttention::setNbRanks` (C++) or `IAttention.num_ranks` (Python) to set the number of devices. Once set to a value greater than 1, this value cannot be changed.

Build-time configuration

1. Add the attention layer with the `IAttention` API.
2. Call `IAttention::setNbRanks(nbRanks)` (C++) or set `attention.num_ranks = nbRanks` (Python).

Runtime requirements

1. Initialize an NCCL communicator with `ncclCommInitRank` or `ncclCommInitAll`.
2. Call `IExecutionContext::setCommunicator(ncclComm_t)` before inference.

3. All participating ranks must execute the same network with synchronized execution calls.

C++

```cpp
// Add attention layer and set number of ranks
auto attention = network->addAttentionV2(query, key, value,
 ↪AttentionNormalizationOp::kSOFTMAX, CausalMaskKind::kNONE);

// At runtime, on each rank, set communicator and run inference
context->setCommunicator(ncclComm);
context->enqueueV3(stream);
```

Python

```python
# Add attention layer and set number of ranks
attention = network.add_attention_v2(query, key, value, trt.
 ↪AttentionNormalizationOp.SOFTMAX, trt.CausalMaskKind.NONE)
attention.num_ranks = num_gpus

# At runtime, on each rank, set communicator and run inference
context.set_communicator(nccl_comm)
context.execute_async_v3(stream)
```

See also

Working with Loops
Loop constructs for autoregressive decoding patterns.

Working with Conditionals
Conditional execution for dynamic decoder logic.

Chapter 16. Best Practices

This guide presents the two pillars of TensorRT performance work: **benchmarking** (measuring what your model actually does) and **optimization** (changing what your model does so it runs faster). Treat them as a feedback loop: measure first, optimize, then measure again to confirm the change had the impact you expected.

Use the table below to decide the chapter to open first.

Table 16.1: Benchmarking vs. Optimization at a glance

	Benchmarking	**Optimization**
Question it answers	"What is my model actually doing - where is time and memory going?"	"How do I make it do less work, or do the same work in parallel?"
Use when...	You need a trusted baseline, or you just changed something and need to confirm the impact.	You have a trusted baseline and a target - lower latency, higher throughput, or smaller memory footprint.
Primary tools	`trtexec`, CUDA events, `IProfiler`, Nsight Deep Learning Designer, NVIDIA Nsight Systems	Builder flags, batching, CUDA graphs, multi-streaming, layer/pointwise/Q-DQ fusion, Tensor Core alignment, precision tuning
Output you get	Latency, throughput, per-layer cost, memory footprint, run-to-run stability	A faster engine, a smaller engine, or a faster engine *build*
Start here	*Performance Benchmarking*	*Optimizing TensorRT Performance*

16.1. Benchmarking

Benchmarking is how you turn a TensorRT model into trustworthy numbers (latency, throughput, per-layer cost) that you can compare across builds, hardware, and configurations. The *Performance Benchmarking* chapter walks through:

- Running `trtexec` against ONNX models, quantized ONNX models, and serialized engines
- Measuring latency and throughput with wall-clock timers and CUDA events, and tracking memory usage
- Profiling per-layer performance with TensorRT's built-in profiler, Nsight Deep Learning Designer, and NVIDIA Nsight Systems

- Controlling the hardware/software environment (GPU clocks, power and thermal throttling, PCIe transfers, driver mode, sync mode) so your numbers are stable and reproducible

Without a stable measurement baseline, every optimization you try is a guess. Start here.

16.2. Optimization

Once you trust your numbers, the *Optimizing TensorRT Performance* chapter covers the techniques you can apply to push them further:

- Increasing parallelism with **batching**, **CUDA graphs**, and **within-/cross-inference multi-streaming**
- Letting the builder do more work for you using **layer fusion**, **pointwise fusion**, and **Q/DQ fusion**
- Tuning specific layer types and aligning tensors for **Tensor Core** acceleration
- Advanced topics: deterministic tactic selection, Python performance, and the **accuracy performance tradeoff**
- Cutting **engine build time** with timing caches and builder optimization levels

Each section is independent, so you can jump straight to the techniques most relevant to the bottlenecks your benchmarks surfaced.

See also

How TensorRT Works
Architecture concepts (builder, runtime, I/O pipeline) that underpin the techniques in both chapters.

Architecture Overview
Foundational context for the TensorRT developer experience - complementary GPU/software ecosystem (MIG, Triton, DALI, Model Optimizer, Polygraphy, Nsight tools), ONNX import, API versioning, and the deprecation policy.

16.2.1. Performance Benchmarking

This guide consolidates everything you need to **measure** TensorRT inference performance, including command-line benchmarking with `trtexec`, advanced timing techniques, profiling tools, and the hardware/software factors that influence the numbers you collect.

What You Will Learn

- How to benchmark an ONNX model or serialized engine with `trtexec`
- How to measure latency and throughput with wall-clock timers and CUDA events
- How to use TensorRT's built-in profiler, Nsight Deep Learning Designer, and Nsight Systems
- Which hardware and software environment factors influence performance measurements

See also

Optimizing TensorRT Performance
After you have established a measurement baseline, apply optimization techniques to improve it.

16.2.1.1 The `trtexec` Command-Line Tool

`trtexec` is the command-line wrapper TensorRT ships around the builder and runtime, letting you exercise both without writing your own application. It calls the same APIs you would from C++ or Python, so the latency, throughput, and per-layer timings it reports translate directly to what your own application will see. Reach for it whenever you need an answer to "how fast does this model run on this GPU?" - or when you need to produce a serialized engine for another application to consume. The source lives in the samples/trtexec directory of the GitHub repository.

What You Will Learn

- How to *benchmark a network* using random or user-provided input data.
- How to *generate a serialized engine* from an ONNX model.
- How to *generate a serialized timing cache* to speed up subsequent builds.
- The most commonly used build- and inference-phase command-line flags.

A typical end-to-end invocation looks like:

```
trtexec --onnx=model.onnx --saveEngine=model.engine
```

Note

`trtexec` is installed at `/opt/tensorrt/bin/trtexec` in the TensorRT NGC container, and is included with every manual TensorRT install. You can also build it from source from the TensorRT OSS repository.

The remaining subsections expand on each purpose and list the flags you are most likely to reach for.

16.2.1.1.1 How `trtexec` Schedules Inferences and Reports Metrics

To maximize GPU utilization, `trtexec` enqueues 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-Streaming* (`--infStreams=N` flag) is used, `trtexec` follows this pattern on each stream separately.

The tool prints the following performance metrics that you can also cross-reference against an Nsight Systems profile of the same run.

- **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 can be underutilized because of host-side overheads or data transfers. The output log indicates the 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 end-to-end latency for 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 the GPU Compute Time, the GPU can be underutilized, and the throughput can be dominated by host-side overhead. Using CUDA graphs (with `--useCudaGraph`) can reduce Enqueue Time.
- **H2D Latency**: The latency for host-to-device data transfers for input tensors of a single inference. Disabled by default. Add `--includeDataTransfers` to include H2D/D2H data transfer measurements.
- **D2H Latency**: The latency for device-to-host data transfers for output tensors of a single inference. Disabled by default. Add `--includeDataTransfers` to include H2D/D2H data transfer measurements.
- **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 the Total Host Walltime, the GPU can be underutilized because of host-side overheads or data transfers.

Note

In the latest Nsight Systems, the GPU rows appear above the CPU rows rather than beneath them.

Add the `--dumpProfile` flag to `trtexec` to show per-layer performance profiles so users can understand the layers in the network that take the most time in GPU execution. The per-layer performance profiling also works with launching inference as a CUDA graph. 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 the operation that each layer in the engine corresponds to and their parameters.

16.2.1.1.2 Benchmarking a Network

If you have a model saved as an ONNX file or a TensorRT plan file, `trtexec` can measure its performance directly. The subsections below cover the three input formats you will typically benchmark, along with how to inspect per-layer timings and how to control the duration of a benchmarking run:

- *Performance Benchmarking with an ONNX File* - the most common starting point, where `trtexec` builds the engine on the fly and runs it.
- *Performance Benchmarking with ONNX+Quantization* - the same flow with FP8 / INT8 quantization applied using ModelOptimizer.
- *Performance Benchmarking with TensorRT Plan File* - benchmarking a serialized engine you (or your build pipeline) produced earlier.

16.2.1.1.2.1 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 use `trtexec` to measure its performance.

For example, the `trtexec` command to measure the performance of ResNet-50 with batch size 4 is:

Listing 16.1: trtexec benchmarking command

```
trtexec --onnx=resnet50-v1-12.onnx --shapes=data:4x3x224x224 --fp16
```

Where:

- The `--onnx` flag specifies the path to the ONNX file.
- The `--shapes` flag specifies the input tensor shapes.

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 this information by visualizing the ONNX model using tools such as Netron or by running a Polygraphy model inspection.

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:

Listing 16.2: Sample FP16 performance summary

```
[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 many performance metrics, but the most important are **Throughput** and median **Latency**. In this case, the ResNet-50 model with batch size 4 can run with a throughput of **507 inferences per second** (2028 images per second since the batch size is 4) and a median latency of **1.969 ms**.

Refer to the *Advanced Performance Measurement Techniques* section for explanations about what **Throughput** and **Latency** mean to your deep learning inference applications, and to the *Commonly Used Command-Line Flags* section for the rest of the flags `trtexec` accepts.

16.2.1.1.2.2 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
```

We use the `--stronglyTyped` flag instead of the `--fp16` flag to require TensorRT to strictly follow the data types in the quantized ONNX model, including all the INT8 Quantize/Dequantize ops.

Here is an example output after running this `trtexec` command with the quantized ONNX model:

Listing 16.3: Sample INT8 quantized performance summary

```
[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,
```

(continues on next page)

(continued from previous page)

```
↪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.
```

The Throughput is **811 inferences per second**, and the median Latency is **1.23 ms**. The Throughput has improved by 60% compared to the FP16 performance results in the previous section.

16.2.1.1.2.3 Per-Layer Runtime and Layer Information

In previous sections, we described using `trtexec` to measure the end-to-end latency. This section will show an example of per-layer runtime and per-layer information using `trtexec`. This will help you determine how much latency each layer contributes to the end-to-end latency and in what 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:

Listing 16.4: trtexec per-layer profiling command

```
trtexec --onnx=resnet50-v1-12-quantized.onnx \
    --shapes=data:4x3x224x224 \
    --stronglyTyped \
    --profilingVerbosity=detailed \
    --dumpLayerInfo \
    --dumpProfile
```

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_tilesize96x64x64_stage3_
```

(continues on next page)

(continued from previous page)

```
↪warpsize2x2x1_g1_tensor16x8x32_simple_t1r1s1, 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 names, and metadata. The `Metadata` field shows the ONNX ops that this layer corresponds to. Since TensorRT has graph fusion optimizations, one engine layer can correspond to multiple ONNX ops in the original model.

The following code is an example log of the per-layer runtime latencies for the last few layers in the quantized ResNet-50 model:

Listing 16.5: Sample per-layer runtime output

```
[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 0.0154 ms and contributes to 1.1% of the end-to-end latency. With this log, you can identify what layers take the largest portion of the end-to-end latency and are the performance bottleneck.

Caution

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 is 1.23 ms.

16.2.1.1.2.4 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 plan file's performance.

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
```

The performance summary output is similar to those in the previous sections.

16.2.1.1.2.5 Duration and Number of Iterations

By default, `trtexec` warms up for at least 200 ms and 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 the *Commonly Used Command-Line Flags* section or run `trtexec --help` for the rest of the flags `trtexec` accepts.

16.2.1.1.3 Generating a Serialized Engine

If you generate a saved serialized engine file, you can pull it into another inference application. 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.

Build an engine from an ONNX model and save it to disk:

```
trtexec --onnx=model.onnx --saveEngine=model.engine
```

16.2.1.1.4 Generating a Serialized Timing Cache

If you provide a timing cache file to the `--timingCacheFile` option, the builder loads existing profiling data from it and appends new profiling entries during layer profiling. The timing cache can be reused across builder instances to shorten build time. Only reuse the cache on matching hardware and software configurations (CUDA, TensorRT, GPU model, and clock frequency); otherwise, you may see functional or performance regressions.

Build an engine while populating (and reusing) a timing cache:

```
trtexec --onnx=model.onnx --timingCacheFile=model.timing.cache --
↪saveEngine=model.engine
```

16.2.1.1.5 Commonly Used Command-Line Flags

Each invocation of `trtexec` runs through up to two phases: a **build phase** that compiles the input model into a TensorRT engine (skip with `--loadEngine` if you already have one), and an **inference phase** that runs the engine and reports performance metrics (skip with `--skipInference` if you only care about producing the engine). Flags fall naturally into one phase or the other and are listed in the corresponding table below. A handful (`--dumpLayerInfo`, `--dynamicPlugins`, `--profilingVerbosity`, and `--verbose`) appear in both tables because they behave differently in each phase.

The lists are curated; for the exhaustive set of flags and detailed descriptions, run `trtexec --help`.

16.2.1.1.5.1 Build Phase Flags

Flag	Description
`--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`.
`--builderOptimizationLevel=N`	Set the builder optimization level to use when building the engine. A higher level allows TensorRT to spend more building time on more optimization options.
`--dumpLayerInfo`, `--exportLayerInfo=<file`	Print and save the engine layer information.
`--dynamicPlugins=<file>`	Load the plugin library dynamically and serialize it with the engine when included in `--setPluginsToSerialize` (can be specified multiple times).
`--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 when loading the engine. Only supported with explicit batch and weights within the engine.
`--layerDeviceTypes=spec`	Explicitly set the per-layer device type. The specs are read left to right and later override earlier ones.
`--markDebug`	Specify a list of tensor names to be marked as debug tensors. Separate names with a comma.
`--markUnfusedTensorsAsDebugTensors`	Mark unfused tensors as debug tensors.
`--maxAuxStreams=N`	Set the maximum number of auxiliary streams per inference stream that TensorRT can 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.
`--memPoolSize=<pool_spec>`	Specify the maximum size of the workspace tactics allowed to be used. Supported pool types include `workspace` and `tacticSharedMem`.
`--minShapes=<shapes>`, `--optShapes=<shapes>`, and `--maxShapes=<shapes>`	Specify the range of input shapes used to build the engine. This is only required if the input model is in ONNX format.
`--noCompilationCache`	Disable the compilation cache in the builder, which is part of the timing cache (the default is to enable the compilation cache).
`--noTF32`	Disable TF32 tactics.
`--onnx=<model>`	Specify the input ONNX model. If the input model is in ONNX format, use the `--minShapes`, `--optShapes`, and `--maxShapes` flags to con-

16.2.1.1.5.2 Inference Phase Flags

Flag	Description
`--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 allocate only what is required for the profile to use. The third option enables `trtexec` to allocate only what is required for the actual input shapes. When setting the allocation strategy to `runtime`, also pass `--preview=+runtimeActivationResize` to enable TensorRT to better estimate the upper bound of the memory size. Note that this will change the allocation algorithm entirely, which is why it is currently a preview feature flag.
`--asyncFileReader=<file>`	Load a serialized engine using an async stream reader. This method uses the `IStreamReaderV2` interface.
`--dumpLayerInfo`, `--exportLayerInfo=<file>`	Print layer information of the engine.
`--dumpProfile`, `--exportProfile=<file>`	Print and save the per-layer performance profile.
`--dynamicPlugins=<file>`	Load the plugin library dynamically when not included in the engine plan file (it can be specified multiple times).
`--getPlanVersionOnly`	Print the TensorRT version when the loaded plan is created. This works without deserializing the plan. Use it together with `--loadEngine`. It is supported only for engines created with 8.6 and later.
`--infStreams=<N>`	Run inference with multiple cross-inference streams in parallel. Refer to the *Cross-Inference Multi-Streaming* section for more information.
`--leanDLLPath=<file>`	External lean runtime DLL is to be used in version-compatible mode. Requires `--useRuntime=[lean|dispatch]`.
`--loadEngine=<file>`	Load the engine from a serialized plan file instead of building it from the input ONNX model.
`--loadInputs=<specs>`	Load input values from files. The default is to generate random inputs.
`--includeDataTransfers`	Turn on host-to-device and device-to-host data transfers.
`--profilingVerbosity=[layer_names_only`	Specify the profiling verbosity to run the inference.
`--saveDebugTensors`	Specify a list of tensor names to turn on the debug state and a filename to save raw outputs. These tensors must be specified as debug tensors during build time.
`--saveAllDebugTensors`	Save all debug tensors to files. Including de-

See also

- Run `trtexec --help` for the full flag list and detailed descriptions.
- The samples/trtexec/README.md file for build instructions and worked examples.

16.2.1.2 Advanced Performance Measurement Techniques

`trtexec` reports throughput, latency, and a per-layer breakdown out of the box, which is enough for most early sizing work. This section covers the techniques you reach for when those numbers are not enough, such as when you are benchmarking inside your own application, when your pipeline includes pre- or post-processing you need to time around, or when you need finer-grained timing than the wrapper can give you.

Before any of those techniques is useful, you need to be precise about *what* you are measuring. The two foundational metrics are:

Latency
: How much time elapses from an input being presented to the network until an output is available, for a single inference. Lower is better. Latency is a hard requirement in safety-critical applications and a quality-of-service signal everywhere users see results in real time. In bulk processing, latency may not matter directly.

Throughput
: How many inferences complete per unit time. Higher is better. Throughput reflects how efficiently the GPU's fixed compute resources are being used and, for bulk workloads, determines the total time to process a batch of work.

A useful third framing is to fix a maximum acceptable latency and measure throughput under that ceiling, which captures the user-experience-versus-efficiency tradeoff most production systems care about.

You also need to choose the exact start and stop points of the measurement. The right choice depends on the network and the application. Most pipelines have pre- and post-processing whose cost belongs in the system-level number but not the inference-only number. This section measures *network inference only*, and leaves wrapper costs to whoever owns the application.

The subsections below cover the three techniques most commonly used to do that:

- *Wall-Clock Timing* - host-side timing using `std::chrono`, useful for end-to-end and pipeline-level numbers.
- *CUDA Events* - device-side timing that avoids host/device synchronization overhead, useful for isolating GPU-only work and for engines with overlapping streams.
- *Tracking Memory* - measuring device memory consumption with a custom `IGpuAllocator`, alongside time-domain numbers.

16.2.1.2.1 Wall-Clock Timing

The wall clock time (the elapsed time between the start of a computation and its end) can be useful for measuring the application's overall throughput and latency and placing inference times in context within a larger system. To measure wall clock time, we can use `std::chrono::steady_clock` provided by the C++11 `<chrono>` standard library.

The following example code snippet shows measuring network inference host time:

C++

```
#include <chrono>

auto startTime = std::chrono::steady_clock::now();
context->enqueueV3(stream);
cudaStreamSynchronize(stream);
auto endTime = std::chrono::steady_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.

16.2.1.2.2 CUDA Events

One problem with timing on the host exclusively is that it requires host/device synchronization. Optimized applications can have many inferences running parallel on the device with overlapping data movement. In addition, the synchronization adds some 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 the GPU will time-stamp 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->enqueueV3(stream);
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)
```

16.2.1.2.3 Tracking Memory

Tracking memory usage can be as important as execution performance. Usually, the device's memory is more constrained than the host's. To keep track of device memory, the recommended mechanism is to create a simple custom GPU allocator that internally keeps some statistics and 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 `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 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
```

16.2.1.3 Performance Profiling Tools

End-to-end latency and throughput numbers tell you *how fast* your engine runs, but not *where the time goes*. Targeted optimizations such as fusing layers, switching precisions, eliminating reformats, or rewriting a hot kernel all require a per-layer or per-kernel breakdown of the inference. The following tools each surface that breakdown at a different level of abstraction:

Table 16.2: Choosing a profiling tool

Tool	Granularity	Use it when...
Built-In TensorRT Profiling	Per-engine-layer timings (after fusion)	You want a quick, in-process breakdown from your own application or from `trtexec`, with no extra tooling installed.
Nsight Deep Learning Designer	Per-engine-layer timings correlated back to the originating ONNX nodes	You are iterating on an ONNX model and want a GUI that maps TensorRT layers, precisions, and timings back onto the source graph.
NVIDIA Nsight Systems	Per-CUDA-kernel timings, GPU/CPU activity timelines, NVTX layer ranges	You need to see kernel launches, stream concurrency, H2D/D2H transfers, or anything the in-process profiler hides; required for networks containing loops or next-gen-optimizer subgraphs that the built-in `IProfiler` cannot decompose.

A typical workflow is to start with the built-in profiler (or `trtexec --dumpProfile`) to find the most expensive engine layers, then drop into Nsight Systems to understand *why* those layers are expensive at the kernel and stream level. Pair every profiling run with the practices in *Hardware/Software Environment for Performance Measurements* so the numbers you collect are stable enough to act on.

16.2.1.3.1 Built-In TensorRT Profiling

Digging deeper into inference performance requires more fine-grained timing measurements within the optimized network.

TensorRT has a `Profiler` interface, which you can implement to have TensorRT pass profiling information to your application. When called, the network will run in a profiling mode. After finishing the 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()` for more information.

Layers inside a loop are compiled into a single monolithic layer; therefore, separate timings for those layers are unavailable. Also, some subgraphs (especially with Transformer-like networks) are handled by a next-generation graph optimizer that has not yet been integrated with the `Profiler` APIs. For those networks, use the *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`).

Given an input network or plan file, you can use `trtexec` to profile a network with TensorRT. For more information, refer to the *trtexec* section.

16.2.1.3.2 ONNX Profiling Tools

Nsight Deep Learning Designer is an integrated design environment for ONNX models, built on top of TensorRT. Its built-in profiler runs an ONNX model through TensorRT and collects per-TensorRT-layer timing correlated back to the originating ONNX operators, so you can see the ONNX nodes that are responsible for the layers consuming inference time.

To profile a model from the GUI:

1. Open Nsight Deep Learning Designer and click **Start Activity** on the Welcome screen.
2. Select the target platform (or define a remote connection to profile on a Linux or L4T target from a remote machine) and choose **Profile TensorRT Model** as the activity.
3. Configure the four activity tabs. The most frequently used settings have analogs in `trtexec`; refer to the Nsight Deep Learning Designer documentation for the full set.
   - **Common** - the ONNX model to profile, its corresponding TensorRT engine if one has already been built, and the location to save the profiler report.
   - **Tactics** - typing mode (default typing, type constraints (*Layer-Level Control of Precision*), or strong typing (*Strongly Typed Networks*)) and on/off toggles for FP16, BF16, TF32, INT8, and FP8 precisions in weakly typed networks (*Network-Level Control of Precision*).
   - **Optimizer** - refittable weights (*Refitting an Engine*) and the INT8 quantization cache path (*Post-Training Quantization (PTQ)*).
   - **Profiler** - locking GPU clocks to base values (*GPU Clock Locking and Floating Clock*) and the GPU counter sampling rate.
4. For networks using dynamic shapes (*Working with Dynamic Shapes*), specify an optimization profile before profiling. You can do this by editing the ONNX network within Nsight Deep Learning Designer, profiling from the command line, or, for compatible networks, setting the **Inferred Batch** option in the Optimizer tab. When a batch size is provided, input shapes with a single leading wildcard are automatically populated with that batch size; this works with input shapes of arbitrary rank.
5. Click **Launch**. The tool deploys TensorRT and CUDA runtime libraries to the target as needed and produces a profiling report.

The report exposes three views useful when correlating TensorRT layers back to ONNX operators.

- **Timeline View** - each network inference stream is shown as a row alongside collected GPU metrics such as SM activity and PCIe throughput. Layers appear as ranges on their stream's row, and overhead such as tensor memory copies or reformats is highlighted in blue.
- **Network Metrics table** - one row per executed TensorRT layer, with type, dimensions, precision, and inference time (both raw and as a percentage of the inference pass). The table is filterable by name, and hyperlinks in the **Name** column open the originating ONNX nodes in their model context. Selecting a range of rows aggregates their statistics into a higher-level summary, with min/max/mean/total times reported in both absolute units and as a percentage of the inference pass.
- **Network Graphs** - aggregate average inference latency and tensor precision distributions by layer type. Use these to spot non-critical computations consuming significant time, and to find quantization opportunities or visualize the effect of TensorRT's tactic precision flags.

Nsight Deep Learning Designer also includes a command-line profiler; refer to the tool documentation for usage instructions.

16.2.1.3.3 CUDA Profiling Tools

The recommended CUDA profiler is NVIDIA Nsight Systems. Some CUDA developers can be more familiar with `nvprof` and `nvvp`. However, these are being deprecated. 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 to report timing information for only a portion of the program's execution or to report traditional CPU sampling profile information and GPU information.

The basic usage of Nsight Systems is first to 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 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, run inference in a second phase, and profile this second phase only.

Suppose the application cannot serialize the engines or must run through the two phases consecutively. In that case, you can also add `cudaProfilerStart()` and `cudaProfilerStop()` CUDA APIs around the second phase and add the `-c cudaProfilerApi` flag to the 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 in the rows under **CUDA HW**, and the CPU activities are shown in the rows under **Threads**. The **CUDA HW** rows are collapsed by default; click to expand them so the GPU work lines up against the CPU activity in the same time window.

In a typical inference workflow, the application calls the `context->enqueueV3()` or `context->executeV3()` APIs to enqueue the jobs and then synchronizes on the stream to wait until the GPU completes the jobs. If you only look at the CPU activities, the system can appear idle for an extended period in the `cudaStreamSynchronize()` call when the GPU is actually busy executing the enqueued work. Always read the **CUDA HW** rows alongside the **Threads** rows to see the full picture.

The `trtexec` tool uses a slightly more complicated approach to enqueue the jobs: it enqueues the next query while the GPU is still executing the jobs from the previous query. For more information, refer to the *trtexec* section.

Use the NVTX Tracing in Nsight Systems

Tracing enables the NVIDIA Tools Extension SDK (NVTX), a C-based API for marking events and ranges in your applications. It allows Nsight Compute and Nsight Systems to collect data generated by TensorRT 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, allowing 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 can 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 builder's choices, multiple additional operations that reorder data can be interspersed with layer computations; these reformat operations can be implemented as device-to-device memory copies or custom kernels.

On the Nsight Systems timeline, this appears as NVTX layer ranges on the CPU thread row aligned with the corresponding kernel ranges on the **CUDA HW** row underneath. Reformat copies show up as

additional kernel ranges between layers, which makes them straightforward to spot when looking for unexpected overhead.

Control the Level of Details in NVTX Tracing

By default, TensorRT only shows layer names in the NVTX markers. At the same time, 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++

Listing 16.6: Disable NVTX tracing

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

Python

Listing 16.7: Disable NVTX tracing

```
builder_config.profiling_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.profiling_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 the *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`, and `--iterations=50` flags allow

you to control how many inference iterations to run. By default, `trtexec` runs inference for three seconds, which can result in a large output of the `nsys-rep` file.

If the CUDA graph not disabled, add `--cuda-graph-trace=node` flag to the `nsys` command to view 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
```

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, and Tensor Core utilization. If the flag is added, these GPU metrics appear in the Nsight Systems web interface.

16.2.1.4 Hardware/Software Environment for Performance Measurements

A performance number is only as trustworthy as the environment it was collected in. Two runs of the same engine on the same GPU can disagree by tens of percent if the clock floats during one run and locks during the other, or if thermal throttling kicks in halfway through. This section catalogs the parts of the hardware and software environment that bias TensorRT measurements, what to watch for, and how to control or work around each one.

The subsections fall into three groups.

Stability of the GPU itself

- *GPU Information Query and GPU Monitoring* - capture the baseline (`nvidia-smi -q`) and a live trace (`nvidia-smi dmon`) of clocks, power, and temperature alongside every benchmark.
- *GPU Clock Locking and Floating Clock* - lock the clock for deterministic comparisons, let it float for peak performance.
- *GPU Power Consumption and Power Throttling* - why power-capped GPUs (T4, A2) skew short-window throughput numbers.
- *GPU Temperature and Thermal Throttling* - what happens at ~85 °C, and how passive cooling and ambient airflow factor in.

Cost of moving data and launching work

- *H2D/D2H Data Transfers and PCIe Bandwidth* - PCIe gen/lanes, pinned memory, NUMA placement, and the `--includeDataTransfers` flag in `trtexec`.
- *Enqueue-Bound Workloads and CUDA Graphs* - when CPU launch overhead, not GPU compute, is the bottleneck, and how *CUDA Graphs* help.

Driver and synchronization configuration

- *GPU Driver Mode* - on Windows, prefer TCC mode for dedicated inference GPUs.
- *BlockingSync and SpinWait Synchronization Modes* - choose between more stable measurements (spin-wait) and lower CPU and power use (blocking-sync).

Important

The items involving `nvidia-smi` in this section are only supported on dGPU systems, not mobile platforms (Jetson, iGPU).

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 encounter unexpected performance measurement results.

Before the inference starts, call the `nvidia-smi -q` command to get detailed information on the GPU, including the product name, power cap, clock settings. 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.

GPU Clock Locking and Floating Clock

By default, the GPU clock frequency is floating, meaning it sits idle when there is no active workload and boosts the boost clock frequency when the workload starts. This is usually the desired behavior since it allows the GPU to generate less heat at idle and to run at maximum speed when there is an 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 `sudo nvidia-smi -q -d SUPPORTED_CLOCKS` command can find the supported clock frequencies. After the clock frequency is locked, it should stay at that frequency unless power or thermal throttling occurs, which will be explained in the next sections. When the throttling kicks in, the device behaves like the clock floats.

Running TensorRT workloads with floating clocks or with throttling taking place can lead to more nondeterminism in tactic selections and unstable performance measurements across inferences because every CUDA kernel can run at slightly different clock frequencies, depending on what 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. Still, 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 what clock frequency to lock the GPU while running TensorRT workloads. It depends on whether the deterministic and stable performance or the best average performance is desired.

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 can lead to unstable performance measurements if the measurements are taken within a short 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 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 to stabilize the performance numbers. However, the average performance numbers will be lower than those with floating clocks or the clock locked at a higher frequency, even though power throttling would happen in this case.

Another issue with power throttling is that it can 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 when the GPU is idle between the inferences. The gaps cause the GPU to consume less power on average, so the clock is throttled less, and the GPU can

run at higher clock frequencies on average. However, the throughput numbers measured this way are inaccurate 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 notice 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 depend on the activation values, causing different input performance measurements. For example, if all the network input values are set to zeros or NaNs, the GPU consumes less power than when the inputs are normal values because of fewer bit-flips in DRAM and the 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 using the `--loadInputs` flag. For more information, refer to the *trtexec* section.

GPU Temperature and Thermal Throttling

Thermal throttling happens when the GPU temperature reaches a predefined threshold of 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 runs until it reaches ~85C and the clock frequency drops.

If thermal throttling happens on actively cooled GPUs like Quadro A8000, then it is possible that the fans on the GPU are broken or obstacles are 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 the wrong numbers of GPUs into the server. In some cases, the air flows through the “easy path” (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 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, the environment’s temperature around the server, does not usually affect GPU performance as long as the GPUs are properly cooled, except for GPUs with lower power limits whose performance can be slightly affected.

H2D/D2H Data Transfers and PCIe Bandwidth

On dGPU systems, the input data must often be copied from the host memory to the device memory (H2D) before an inference starts, and the output data must be copied back from the device memory to the 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 parallel to the GPU executions of other inferences so that the GPU does not sit idle when the H2D/D2H copies occur. This can

be done by running multiple inferences in parallel streams or 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 an example of the latter implementation.

When the H2D/D2H copies run 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 to determine whether 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 what generation (such as Gen3 or Gen4) and how many lanes (such as x8 or x16) are used. Next, reduce the amount of data that must be transferred using the PCIe bus. For example, suppose the input images have high resolutions, and the H2D copies become the bottleneck. In that case, you can transmit JPEG-compressed images over the PCIe bus and decode the image on the GPUs before the inference workflow instead of transmitting raw pixels. Finally, 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 machine's NUMA (Non-Uniform Memory Access) configurations with the `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 where the GPU that will receive the copied data resides. Also, pin the CPU threads that trigger the H2D/D2H copies on that specific NUMA node.

Note that the host and the device share the same memory on mobile platforms, so the H2D/D2H data transfers are not required if the host memory is allocated using CUDA APIs and is pinned instead of pageable.

By default, the `trtexec` tool excludes the latencies of the H2D/D2H data transfers. You can add the `--includeDataTransfers` flag to include the latencies of the H2D/D2H data transfers to see if the H2D/D2H copies can bottleneck the TensorRT workload.

GPU Driver Mode

Windows: TCC Mode (recommended)

On Windows, configure the GPU to TCC (Tesla Compute Cluster) mode for best inference performance:

```
nvidia-smi -dm 1
```

In TCC mode, the GPU focuses on computation work, and graphics support like OpenGL or monitor display is disabled. This is the recommended mode for GPUs that run TensorRT inference workloads.

> **Warning**
>
> A GPU connected to a display must not be configured into TCC mode.

For more information, refer to the TCC mode documentation.

Windows: WDDM Mode

WDDM (Windows Display Driver Model) mode supports both graphics and compute but tends to cause GPUs to have worse and unstable performance results when running inference workloads using TensorRT. Use TCC mode when possible for dedicated inference GPUs.

Linux

Linux does not have a TCC/WDDM mode distinction. The GPU driver is always in compute-capable mode. No driver mode configuration is needed for TensorRT inference on Linux.

Enqueue-Bound Workloads and CUDA Graphs

The `enqueueV3()` 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 `enqueueV3()` time can take longer than the actual GPU executions, causing the latency of `enqueueV3()` calls to become the performance bottleneck. We say that this type of workload is "enqueue-bound." Two reasons can cause a workload to be enqueue-bound.

First, if the workload is very tiny in terms of the number 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 increasing the computation per CUDA kernel by increasing the batch size. You can also use *CUDA Graphs* to capture the kernel launches into a graph and launch the graph instead of calling `enqueueV3()`.

Second, it is naturally queue-bound if the workload contains operations requiring device synchronizations, such as loops or if-else conditions. Increasing the batch size can help improve the throughput without increasing the latency.

In `trtexec`, CUDA graphs are enabled by default. You can add the `--noCudaGraph` flag to disable the use of CUDA graphs and check if a workload is enqueue-bound by checking whether the reported `Enqueue Time` is close to or longer than the reported `GPU Compute Time`.

`BlockingSync` and `SpinWait` Synchronization Modes

If performance is measured with `cudaStreamSynchronize()` or `cudaEventSynchronize()`, synchronization overhead variations can lead to performance measurement variations. This section describes the causes of the variations and how to avoid them.

When `cudaStreamSynchronize()` is called, there are two ways that the driver waits until the stream is completed. If the `cudaDeviceScheduleBlockingSync` flag has been set with `cudaSetDeviceFlags()` calls, then the `cudaStreamSynchronize()` uses the blocking-sync mechanism. Otherwise, it uses the spin-wait mechanism.

A 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. 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, leading 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. Still, 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 do not want the stream/event synchronizations to consume CPU resources (such as 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 in spin-wait mode. You can add the `--blockingSync` flag to enable synchronizations using the blocking-sync mode for less CPU utilizations and less power consumption at the cost of more unstable latency measurements.

16.2.2. Optimizing TensorRT Performance

The following sections focus on the general inference flow on GPUs and some general strategies to improve performance. These ideas apply to most CUDA programmers but cannot be as obvious to developers from other backgrounds.

What You Will Learn

- How batching, CUDA graphs, and multi-streaming improve throughput
- How TensorRT's layer fusion and pointwise fusion work, and which patterns are supported
- How to optimize specific layer types and target Tensor Core acceleration
- Advanced techniques for deterministic tactic selection, Python performance, and accuracy/performance tradeoffs
- How to reduce engine build time with timing caches and builder optimization levels

See also

Performance Benchmarking
Establish a measurement baseline before applying these optimizations so you can quantify the impact of each change.

16.2.2.1 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 similarly. Therefore, each instance can be trivially computed in parallel.

Each network layer will have some 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 fully connected layer with `V` inputs and `K` outputs can be implemented for one batch instance as a matrix multiplied by a `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 should be avoided if possible. But

generally, increasing the batch size improves total throughput. In addition, when the network contains MatrixMultiply 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 or later GPUs, decreasing the batch size can improve the throughput significantly if the smaller batch sizes help the GPU cache the input/output values in the L2 cache. Therefore, various batch sizes should be tried to find the batch size that provides optimal performance.

Sometimes, batching inference work is impossible due to the application's organization. In some common applications, such as a server that makes inferences per request, it is possible to implement opportunistic batching. For each incoming request, wait for a time T. If other requests come in, batch them together. Otherwise, continue with a single-instance inference. This strategy adds fixed latency to each request but can greatly improve the system's maximum throughput.

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 by adding optimization profiles. For more information, refer to the *Working with Dynamic Shapes* section.

16.2.2.2 Inference with CUDA Graphs

The CPU has to launch every CUDA kernel that an inference fans out, and each launch costs roughly 5-15 microseconds of host time. For models with many small kernels, that launch overhead can exceed the actual GPU work and become the bottleneck (the *Enqueue-Bound Workloads and CUDA Graphs* section in the benchmarking chapter explains how to detect this). CUDA Graphs collapse the entire sequence of kernels into a single launchable object, so the host pays the per-launch cost once at capture time, then reuses the recorded graph on every subsequent inference.

The TensorRT runtime supports CUDA graph capture out of the box, but capture has constraints you need to know before you wire it into your inference loop. The subsections below cover them in order.

- *Using CUDA Graphs with TensorRT Execution Context* - the C++ and Python recipes for capturing an `enqueueV3()` call as a graph and replaying it using `cudaGraphLaunch`.
- *Limitations of CUDA Graphs* - operators and patterns that fail capture (loops, conditionals, data-dependent shapes), the two-phase shape-update protocol, and why you should keep one execution context per captured graph.
- *Concurrent CUDA Activities with CUDA Graph Capture* - using auxiliary streams, the `cudaStreamNonBlocking` flag, and the async CUDA APIs (`cudaMemcpyAsync`) so other GPU work does not break the capture.

16.2.2.2.1 Using CUDA Graphs with TensorRT Execution Context

TensorRT's `enqueueV3()` method supports CUDA graph capture for models requiring no mid-pipeline CPU interaction. For example:

C++

```
// Call enqueueV3() once after an input shape change to update internal state.
context->enqueueV3(stream);

// Capture a CUDA graph instance
```

(continues on next page)

(continued from previous page)

```
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)
```

16.2.2.2.2 Limitations of CUDA Graphs

CUDA graphs cannot handle some operations, so graph capturing can fail if the execution context contains such operations. Typical deep learning operators unsupported by CUDA graphs include loops, conditionals, and layers requiring data-dependent shapes. In these cases, `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. Refer to the CUDA Programming Guide to learn more about the limitations of CUDA graphs.

Also, when capturing a graph, it is important to account for the two-phase execution strategy used in the presence of dynamic shapes.

1. Update the model's internal state to account for any changes in input size.
2. Stream work to the GPU.

The first phase requires no per-invocation work for models where input size is fixed at build time. Otherwise, if the input sizes have changed since the last invocation, some work can be required to update derived properties.

The first phase of work is not designed to be captured, and even if the capture is successful, it can 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.

Danger

Graphs captured with TensorRT are specific to the input size and the state of the execution context. Modifying the context that the graph was captured from will result in undefined behavior when executing the graph. In particular, if the application is providing its memory for activations using `createExecutionContextWithoutDeviceMemory()`, the memory address is also captured as part of the graph. Locations of input and output buffers 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 TensorRT engine you built is compatible with CUDA graph capture. For more information, refer to the *trtexec* section.

16.2.2.2.3 Concurrent CUDA Activities with CUDA Graph Capture

Launching a CUDA kernel on the CUDA legacy default stream or calling synchronous CUDA APIs like `cudaMemcpy()` while capturing a CUDA graph fails because these CUDA activities implicitly synchronize the CUDA streams used by TensorRT execution contexts.

To avoid breaking the CUDA graph capture, ensure other CUDA kernels are launched on non-default CUDA streams and use the asynchronous version of CUDA APIs, like `cudaMemcpyAsync()`.

Alternatively, a CUDA stream can be created with the `cudaStreamNonBlocking` flag to capture the CUDA graph for an execution context. If the execution context uses auxiliary streams, make sure you also call the `setAuxStreams()` API using streams created with the `cudaStreamNonBlocking` flag. Refer to the *Within-Inference Multi-Streaming* section about how to set auxiliary streams in TensorRT execution contexts.

16.2.2.3 Inference with Multiple CUDA Streams

A CUDA stream is a queue of GPU work that executes in order, but work in *different* streams can overlap whenever the hardware has spare compute, memory bandwidth, or copy engines. TensorRT exposes that latent parallelism in two complementary ways. You can split the work *within* a single inference across auxiliary streams to drive its latency down, and you can run multiple inferences *across* their own streams to drive aggregate throughput up. The two techniques also stack, so a server can run several execution contexts in parallel with each context itself fanning work across auxiliary streams.

The tradeoff in both cases is contention. Concurrent streams share SMs, register files, L2, and DRAM bandwidth, so kernels TensorRT picked while profiling against a fully available GPU may no longer be optimal at runtime. The third subsection below addresses that.

- *Within-Inference Multi-Streaming* - split the layers of one engine across auxiliary streams to lower single-inference latency when one inference does not saturate the GPU.
- *Cross-Inference Multi-Streaming* - run multiple execution contexts on their own streams to raise aggregate throughput when you can keep the GPU fed with independent requests.
- *Limiting Compute Resources* - tell the builder to profile against a smaller share of the GPU so the tactics it picks remain optimal under the contention that concurrent streams introduce.

Note

Multi-streaming is orthogonal to *CUDA Graphs*. Streams give you parallelism between work; CUDA graphs reduce the per-launch overhead *of* that work. Enqueue-bound workloads usually want both.

16.2.2.3.1 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 can execute out of order concerning other streams. In particular, asynchronous commands in two streams can 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 can fully use the hardware's computation capabilities. 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.

Use the `IBuilderConfig::setMaxAuxStreams()` API to set the maximum number of auxiliary streams TensorRT can use to run multiple layers in parallel. The auxiliary streams contrast the "mainstream" provided in the `enqueueV3()` call. If enabled, TensorRT will run some layers on the auxiliary streams parallel to those 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 can use 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 the following:

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 mainstream provided using enqueueV3() call and the auxiliary streams:

- At the beginning of the enqueueV3() call, TensorRT will ensure that all the auxiliary streams wait on the activities on the mainstream.
- At the end of the enqueueV3() call, TensorRT will ensure that the mainstream waits for the activities on all the auxiliary streams.

Enabling auxiliary streams can increase memory consumption because some activation buffers can no longer be reused.

16.2.2.3.2 Cross-Inference Multi-Streaming

In addition to the within-inference streaming, you can 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 several streams sharing compute resources simultaneously. This means the network can have fewer compute resources available during inference than when the TensorRT engine was optimized. This difference in resource availability can cause TensorRT to choose a suboptimal kernel for the actual runtime conditions. To mitigate this effect, you can limit the amount of available compute resources during engine creation to resemble actual runtime conditions more closely. This approach generally promotes throughput at the expense of latency. For more information, refer to the *Limiting Compute Resources* section.

It is also possible to use multiple host threads with streams. A common pattern is incoming requests dispatched to a pool of worker threads waiting for work. In this case, the pool of worker threads will each have one execution context and CUDA stream. Each thread will request work in its stream as the work becomes available. Each thread will synchronize with its stream to wait for results without blocking other worker threads.

16.2.2.3.3 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 fewer resources (note that the recommended approach is to build the engine on the GPU that will be used for inference, but this cannot 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 computing 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.

16.2.2.4 Enabling Layer Fusion

Layer fusion is one of the highest-impact optimizations the TensorRT builder performs. When the builder recognizes a supported pattern (for example, a Convolution feeding a ReLU, or a chain of element-wise operations), it collapses those layers into a single optimized kernel. The fused kernel:

- **Eliminates kernel-launch overhead** for every layer it absorbs (typically 5–15 µs per launch on the host), which can dominate runtime in enqueue-bound networks.
- **Avoids materializing intermediate tensors** in DRAM, removing both the allocation pressure and the round-trip memory traffic between layers.
- **Unlocks specialized implementations** (such as fused Conv+ReLU, fused Conv+Bias+Activation, or pointwise super-kernels) that no individual layer could use on its own.

Fusion happens automatically at build time (there is no runtime API to “turn it on”), but the patterns the builder is willing to fuse are constrained. The remaining sections describe what those patterns are so you can build (or rewrite) networks that fuse well:

- *Layer Fusion* - introduces the mechanism, explains how the builder logs fusion decisions, and describes the fused-layer naming convention you will see in profiles.
- *Types of Fusions* - catalog of supported pattern fusions (for example, Convolution + Activation, Shuffle + Reduce, Padding + Convolution) and reduction-op fusions (GELU, L1Norm, L2Norm, LogSumExp).
- *Pointwise Fusion* - covers a broader fusion that collapses chains of adjacent pointwise ops (Activation, ElementWise, Unary, Scale, single-value Constant) into one kernel.

- *Q/DQ Fusion* - dedicated guidance for INT8 and FP8 networks containing `QuantizeLinear` and `DequantizeLinear` layers.

Tip

To see exactly the fusions that the builder applied to your network, set the `ILogger` callback to surface `kINFO` messages during the build, or run `trtexec` with `--profilingVerbosity=detailed --dumpLayerInfo`. Layers whose names contain a `+` (such as `ip1 + relu1`) or that are wrapped in `fusedPointwiseNode(...)` are the result of fusion.

16.2.2.4.1 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 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 an internal options set.

Consider the common case of a convolution followed by ReLU activation. Creating a network with these operations involves adding a Convolution layer with `addConvolutionNd` and 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 implementation supports the operations, then fuse them into one layer.

To investigate the fusions that have occurred, the builder logs its operations to the logger object provided during construction. Optimization steps are at the `kINFO` log level. To view 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 that were fused. For example, a MatrixMultiply layer (InnerProduct) named `ip1` is fused with a ReLU Activation layer named `relu1` to create a new layer named `ip1 + relu1`.

16.2.2.4.2 Types of Fusions

The following list describes the types of supported fusions.

Supported Layer Fusions

- **ReLU Activation**: A single activation layer will replace an Activation layer performing ReLU followed by an activation performing ReLU.
- **Convolution and ReLU Activation**: The Convolution layer can be of any type, and values are not restricted. The Activation layer must be of the ReLU type.
- **Convolution and GELU Activation**: The input and output precision should be the same, with both of them FP16 or INT8. The Activation layer must be GELU type. TensorRT should run on an NVIDIA Turing or later with CUDA version 10.0.
- **Convolution and Clip Activation**: The Convolution layer can be any type, and values are not restricted. 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**: If all the padding sizes are non-negative, padding followed by a Convolution or Deconvolution can be fused into a single Convolution/Deconvolution layer.
- **Shuffle and Reduce**: A Shuffle layer without reshaping, 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 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**: Adjusting the convolution weights can fuse a convolution layer followed by a Scale layer that is `kUNIFORM` or `kCHANNEL` into a single convolution. 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 output can be called a generic activation layer. A convolution layer followed by a generic activation layer can be fused into a single convolution layer.
- **Reduce**: It performs average pooling, which a Pooling layer will replace. The Reduce layer must have a `keepDimensions` set and be reduced across `H` and `W` dimensions from the 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 can already have a fused activation operation from a previous fusion.
- **Depthwise Separable Convolution**: A depthwise convolution with activation followed by a convolution with activation can sometimes be fused into a single optimized `DepSepConvolution` layer. The precision of both convolutions must be INT8, and the device's computation capability must be 7.2 or later.
- **Softmax and Log**: If it has not already been fused with a previous log operation, it can be fused into a single Softmax layer.
- **Softmax and TopK**: It can be fused into a single layer. The Softmax can optionally include a Log operation.

Supported Reduction Operation Fusions

- **GELU**: A group of Unary and ElementWise layers representing the following equations can be fused into a single `GELU` reduction operation.

  $0.5x \times \left(1 + \tanh\left(\frac{2}{\pi}\left(x + 0.044715x^3\right)\right)\right)$

  Or the alternative representation:

  $0.5x \times \left(1 + erf\left(\frac{x}{\sqrt{2}}\right)\right)$

- **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 operation.

16.2.2.4.3 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, so some layers cannot be fused.

Fusion creates a new layer with a name consisting of both fused layers. For example, an ElementWise layer named `add1` is fused with a ReLU Activation layer named `relu1`, creating a new layer named `fusedPointwiseNode(add1, relu1)`.

16.2.2.4.4 Q/DQ Fusion

Refer to the *Explicit Quantization* section for suggestions on optimizing INT8 and FP8 networks containing `QuantizeLinear` and `DequantizeLinear` layers.

16.2.2.5 Optimizing Layer Performance

While *Enabling Layer Fusion* covers the optimizations TensorRT applies *across* layers, this section covers what makes individual layers themselves run well: both the choices you make when authoring the network and the hardware features the builder targets when it picks an implementation. The subsections below address the two highest-impact areas:

- *Optimizing for Tensor Cores* - alignment and dimension rules for the GPU hardware path that delivers the largest speedups for MatrixMultiply, Convolution, and Deconvolution.
- *Optimizing Plugins* - guidance for custom layers, where the kernel is your code and standard CUDA performance practices apply.

Before diving into the subsections, the following two lists capture the most common per-layer guidance. The first list is **action items when authoring the network**; the second is **automatic optimizations** to be aware of when reading profiles, but which require no action on your part.

When authoring your network

- **Gather**: Use an axis of 0 to maximize the performance of a Gather layer. There are no fusions available for a Gather layer.

- **Reduce**: 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.
- **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 access. Reductions along multiple dimensions at once can be simulated using a Shuffle layer to reshape the data and then appropriately reinterpret the index values.

Automatic optimizations TensorRT applies

- **RNN**: The loop-based API provides a much more flexible mechanism for using general layers within recurrence. 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 layer are properly combined to maximize machine utilization after loop unrolling along the sequence dimension. This works best if you can avoid a MatrixMultiply layer with a recurrent data dependence along the sequence dimension.
- **Shuffle**: Shuffle operations 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.

16.2.2.5.1 Optimizing for Tensor Cores

Tensor Core is a key technology for delivering high-performance inference on NVIDIA GPUs. In TensorRT, Tensor Core operations are supported by all compute-intensive layers: MatrixMultiply, Convolution, and Deconvolution.

Tensor Core layers tend to achieve better performance if the I/O tensor dimensions are aligned to a certain minimum granularity:

- The alignment requirement is on the I/O channel dimension in the Convolution and Deconvolution layers.
- In the MatrixMultiply layer, 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 16.3: 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 unmet, 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, it cannot use a Tensor Core implementation even if it is 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 using Nsight Systems to profile TensorRT applications.

It is impractical to expect a CUDA kernel to reach 100% Tensor Core usage since there are other overheads such as DRAM reads/writes, instruction stalls, and other computation units. The more computation-intensive an operation is, the higher the Tensor Core usage rate the CUDA kernel can achieve.

16.2.2.5.2 Optimizing Plugins

TensorRT provides a mechanism for registering custom plugins that perform layer operations. After a plugin creator is registered, you can search the registry to find the creator and add the corresponding plugin object to the network during serialization/deserialization.

After the plugin library is loaded, all TensorRT plugins are automatically registered. For more information about custom plugins, refer to *Extending TensorRT With Custom Layers*.

Plugin performance depends on the CUDA code performing the plugin operation. Standard CUDA Best Practices apply. When developing plugins, starting with simple standalone CUDA applications that perform the plugin operation and verify correctness can be helpful. 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.

Supporting as many formats as possible in the plugin is important to get the best performance possible. 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.

16.2.2.6 Advancing Performance Optimization Techniques

Once batching, CUDA graphs, multi-streaming, fusion, and per-layer tuning have taken your engine as far as the obvious knobs allow, a second tier of issues tends to surface. These are situational rather than universal: most networks will not hit all of them, but every production deployment hits at least one. This section collects the techniques that address them:

- *Overhead of Shape Change and Optimization Profile Switching* - explains the one-time cost paid by the *first* `enqueueV3()` call after a dynamic-shape change or profile switch. Important if your application's tail latency matters or if you switch profiles frequently.
- *Deterministic Tactic Selection* - covers how to make the builder pick the *same* tactics across rebuilds (clock locking, more averaging iterations, timing-cache reuse). Important for A/B testing, regression triage, and any workflow where bit-for-bit reproducibility matters.
- *Optimizing Python Performance* - addresses the Python-specific overheads around input buffer setup; inference itself is already on par with C++.
- *Tradeoffs Between Accuracy and Performance* - the playbook for when reduced precision (FP16, BF16, FP8, INT8) or aggressive tactic selection has degraded accuracy. Covers per-layer precision overrides, calibration, the editable timing cache, and the precision-debug workflow.

16.2.2.6.1 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 `enqueueV3()` call after a shape/profile change can be longer than the subsequent `enqueueV3()` calls.

16.2.2.6.2 Deterministic Tactic Selection

TensorRT runs through all the possible tactics in the engine-building phase and selects the fastest ones. Since the selection is based on the tactics' latency measurements, TensorRT can select different tactics across different runs if some have similar latencies. As a result, different engines built from the same `INetworkDefinition` can behave slightly differently regarding output values and performance. You can inspect the selected tactics of an engine by using the *engine inspector APIs* or by turning on verbose logging while building the engine.

If deterministic tactic selection is desired, the following lists a few suggestions that can help improve the determinism of tactic selection.

Locking GPU Clock Frequency

By default, the GPU's clock frequency 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 can 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 building a TensorRT engine can 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 can 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 can 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.

16.2.2.6.3 Optimizing Python Performance

Most of the same performance considerations apply when using the Python API. 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 comes from. Internally, `pycuda` supports the Python Buffer Protocol, allowing efficient access to memory regions. This means that if the input data is available in a suitable format in `numpy` arrays or another type with support for the buffer protocol, it allows efficient access and transfer to the GPU. For even better performance, allocate a page-locked buffer using `pycuda` and write your final preprocessed input.

For more information about using the Python API, refer to the *Python API documentation*.

16.2.2.6.4 Tradeoffs Between Accuracy and Performance

Depending on the builder configuration, TensorRT can execute a layer in FP32, FP16, BF16, FP8, or INT8 precision. 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 higher-precision layer helps improve accuracy with some performance hits.

There are several steps that we can take to improve model accuracy:

1. Validate layer outputs:
    1. Use Polygraphy to dump layer outputs and verify no NaNs or Infs. The `--validate` option can check for NaNs and Infs. Also, we can compare layer outputs with golden values from, such as ONNX runtime.
    2. For FP16 and BF16, a model might require retraining to ensure that intermediate layer output can be represented in FP16/BF16 precision without overflow or underflow.
    3. For INT8, consider recalibrating with a more representative calibration data set. If your model comes from PyTorch, we also provide the NVIDIA Model Optimizer 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:
    1. Sometimes, running a layer with a certain precision results in incorrect output. This can be due to inherent layer constraints (such as `LayerNorm` output should not be INT8) or model constraints (output gets diverged, resulting in poor accuracy).
    2. You can control layer execution precision and output precision.
    3. An experimental debug precision tool can help automatically find layers to run with high precision.
3. Use the Editable Timing Cache to select a proper tactic.
    1. When accuracy changes between two built engines for the same model, it might be due to a bad tactic being selected for a layer.
    2. Use Editable Timing Cache to dump available tactics. Update the cache with a proper one.

Accuracy from run-to-run variation should not change; after the engine is built for a specific GPU, it should result in bit-accurate outputs in multiple runs. If not, file a *TensorRT bug*.

See also

Working with Quantized Types
INT8, FP8, and FP4 quantization techniques and calibration workflows.

Troubleshooting
Diagnosing accuracy issues and common error patterns.

16.2.2.7 Reducing Engine Build Time

Every other section in this chapter has been about making *inference* faster. This one is about making the *builder* faster (the offline step that profiles each layer's available tactics, picks the best one, and emits a serialized engine). Build time can range from seconds for small networks to tens of minutes (or longer) for large models with complex topology, and that cost shows up wherever builds happen often, such as in CI pipelines, model-iteration loops during development, fleet redeployments after a model update, and per-device builds when shipping to heterogeneous hardware.

There is a fundamental tradeoff to be aware of: the more time the builder spends searching, the better the tactics it can find, and therefore the faster the resulting engine runs. The two subsections below give you two distinct levers to tune that tradeoff:

- *Timing Cache* - *reuse* tactic-timing measurements from previous builds so the builder does not re-profile layers it has already seen. Effectively free inference performance, since the cached timings represent the same exhaustive search the original build performed. Use this on every build pipeline you control.
- *Builder Optimization Level* - *reduce* how much tactic search the builder performs in the first place. Trades inference performance for faster builds. Use this when you are iterating on a model and need quick feedback, and dial it back up before shipping the production engine.

The two compose well: a timing cache shared across CI runs cuts the cost of every build, and a lowered optimization level cuts the cost of any uncached layers during development.

16.2.2.7.1 Timing Cache

TensorRT creates a layer-timing cache to reduce builder time and 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`.

The TensorRT builder skips profiling and reuses the cached result for the repeated layers if other layers have the same IO tensor configuration and layer parameters. 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);
```

Due to cache misses, the timing cache can be augmented with more information during the build. After the build, it can be serialized for use with another builder.

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

If a builder does not have a timing cache attached, it creates its 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);
```

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.

16.2.2.7.2 Builder Optimization Level

Set the optimization level in the 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 engine's performance can be worse. On the other hand, setting it to a larger value will increase the engine building time, but the resulting engine can perform better if TensorRT finds better tactics.

For example, to set the optimization level to `0` (the fastest):

C++

```
config->setBuilderOptimizationLevel(0);
```

Python

```
config.builder_optimization_level = 0
```

Chapter 17. Troubleshooting

This guide helps you diagnose and resolve common issues when working with TensorRT. It provides solutions to frequently encountered problems, explains error messages, and offers strategies for debugging TensorRT applications.

What You Will Learn

- **Frequently Asked Questions (FAQs)** - Quick answers to common questions about engine building, deployment, and performance
- **Understanding Error Messages** - Explanations of common error messages and how to resolve them
- **Reporting Issues** - How to gather diagnostic information and report bugs effectively

How to Use This Guide

1. **Start with FAQs** - Check if your question has already been answered
2. **Search for Error Messages** - Use Ctrl+F to find specific error messages you are encountering
3. **Follow Systematic Debugging** - Use the debugging strategies provided for complex issues
4. **Gather Diagnostics** - Collect necessary information before reporting issues

Getting Additional Help

If this guide does not resolve your issue:

- Post questions on the NVIDIA Developer Forum with the `tensorrt` tag
- Open an issue on the TensorRT GitHub repository
- Contact your NVIDIA support engineer if you have enterprise support

17.1. FAQs

This section provides answers to the most frequently asked questions about TensorRT. Questions are organized by topic for easy navigation.

How do I create an optimized engine for several batch sizes?

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 batch sizes, create optimization profiles at the dimensions assigned to `OptProfilerSelector::kOPT`.

Are calibration tables portable across TensorRT versions?

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.

Are engines portable across TensorRT versions?

By default, no. Refer to the *Version Compatibility* section for instructions on configuring engines for forward compatibility.

How do I choose the optimal workspace size?

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. The workspace size can need to be limited to less than the full device memory size if device memory is needed for other purposes during the engine build.

How do I use TensorRT on multiple GPUs?

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 that it was created from. When calling `execute()` or `enqueue()`, ensure that the thread is associated with the correct device by calling `cudaSetDevice()` if necessary.

How do I get the version of TensorRT from the library file?

There is a symbol in the symbol table named `tensorrt_version_#_#_#_#` that 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_#_#_#_#
```

What can I do if my network produces the wrong answer?

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 generates exactly the input format the network requires.
- If you are using reduced precision, run the network in FP32. If it produces the correct result, lower precision can have an insufficient dynamic range for the network.
- Try marking intermediate tensors in the network as outputs and verify if they match your expectations.

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.

How do I implement batch normalization in TensorRT?

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
```

Does TensorRT support INT4 quantization or INT16 quantization?

TensorRT supports INT4 quantization for GEMM weight-only quantization. TensorRT does not support INT16 quantization.

What should I do if the ONNX parser does not support a layer my network requires?

The TensorRT ONNX parser is an open-source project. You can add support for custom operators through TensorRT plugins. UFF and Caffe parsers have been removed; use the ONNX workflow instead.

Can I use multiple TensorRT builders to compile on different targets?

Warning

TensorRT assumes that all resources for the device it is building on are available for optimization purposes. Concurrent use of multiple TensorRT builders (such as multiple `trtexec` instances) to compile on different targets can oversubscribe system resources causing undefined behavior (meaning, inefficient plans, builder failure, or system instability).

Using `trtexec` with the `-saveEngine` argument, it is recommended to compile for different targets 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. 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.

Which layers are accelerated by Tensor Cores?

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 can be faster and be selected instead of a tensor core implementation.

Why are reformatting layers observed, although there is no warning message that no implementation obeys reformatting-free rules?

Reformat-free network I/O does not mean reformatting layers are not inserted into the entire network. Only the input and output network tensors can be configured not to require reformatting layers; in other words, TensorRT can insert reformatting layers for internal tensors to improve performance.

17.2. Understanding Error Messages

If an error occurs during execution, TensorRT reports an error message intended to help debug the problem. The following sections discuss some common error messages that developers can encounter.

ONNX Parser Error Messages

The following table captures the common ONNX parser error messages. For specific ONNX node support information, refer to the Operators' support document.

Table 17.1: Common ONNX Parser Error Messages

Error Message	Description
`<X> must be an initializer!`	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`
`!inputs.at(X).is_weights()`	
`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.

Table 17.2: Common TensorRT Core Library Error Messages

	Error Message	Description
Installation Errors	`CUDA initialization failure with error <code>. Please check the CUDA installation: http://docs.nvidia.com/cuda/cuda-installation-guide-l index.html.`	This error can occur if the CUDA or NVIDIA driver installation corrupts. Refer to the URL for instructions 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::setMemory`	This error 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 have non-zero count but null values <layer-name>: (kernel|bias) weights have zero count but non-null values`	This error occurs when a mismatch between the values and count fields in a Weights data structure is passed to the builder. If the count is 0, 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 a device different from the current device.` | This error can occur if you create an `IBuilder` targeting one GPU, called `cudaSetDevice()`, to target a different GPU and then attempt to use the `IBuilder` to create an engine. Ensure you only use the `IBuilder` when targeting the GPU used to create the `IBuilder`.	
	You can encounter error messages indicating that the tensor dimensions do not match the semantics of the given layer. Carefully verify 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: - Constant tensor with all zero values; not an error. - Activation (ReLU) output with all negative inputs; not an error. - Data distribution is forced to all zero due to a computation error in the previ-

17.3. Code Analysis Tools

17.3.1. Compiler Sanitizers

Google sanitizers are a set of code analysis tools.

17.3.1.1 Issues With dlopen And Address Sanitizer

There is a known issue with sanitizers, which is 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.

17.3.1.2 Issues with dlopen and Thread Sanitizer

The thread sanitizer can list errors when using `dlopen` from multiple threads. 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"
```

17.3.1.3 Issues with CUDA and Address Sanitizer

The address sanitizer has a known issue with CUDA applications, which is documented here. 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.

17.3.1.4 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.

17.3.2. Valgrind

Valgrind is a framework for dynamic analysis tools that can 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
```

(continues on next page)

[1] Evaluating the calibration input or validating the previous layer outputs is recommended.

(continued from previous page)

```
    match-leak-kinds: definite
    ...
    fun:*dlopen*
    ...
}
```

17.3.3. 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 later in CUDA but unavailable on the current platform.

17.4. 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)
```

Where:

- `Half` indicates that the element type is `DataType::kHALF`, a 16-bit floating point
- `: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 of 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 vector
- `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`

17.5. Reporting TensorRT Issues

If you encounter issues when using TensorRT, 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, because the issue may have been fixed in the latest release.

17.5.1. Channels for TensorRT Issue Reporting

If neither the *FAQs* nor the *Understanding Error Messages* sections help, you can report the issue through the NVIDIA Developer Forum or the TensorRT GitHub Issue page. These channels are constantly monitored to provide feedback on the issues you encounter.

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 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 versions (if any of them are used).
  - TensorRT version
  - NGC TensorRT container version (if TensorRT container is used).
  - Jetson (if used), include OS and hardware versions

- A thorough description of the issue.
- Steps to reproduce the issue:
  - ONNX file (if ONNX is used).
  - A TensorRT API Capture (if ONNX is not used). For more information, refer to the *TensorRT API Capture and Replay* section.
  - 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.

17.5.2. 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 commands to reproduce the issue and a detailed description of the environment. Having more details helps us debug the functional issue faster.

If you are not using ONNX, it is recommended to capture the TensorRT API calls and attach the JSON and BIN files. For more information, refer to the *TensorRT API Capture and Replay* section.

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. Also, ensure 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).

17.5.3. Reporting an Accuracy Issue

When reporting an accuracy issue, provide the scripts and the commands used to calculate the accuracy metrics. Describe the expected accuracy level and share the steps to get the expected results using other frameworks like ONNX-Runtime.

The Polygraphy tool can debug the accuracy issue and produce a minimal failing case. For instructions, refer to the documentation on Debugging TensorRT Accuracy Issues. Having a Polygraphy command that shows the accuracy issue or having a 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 encounter a significant drop in the accuracy metric between TensorRT and other frameworks such as PyTorch, TensorFlow, or ONNX-Runtime, it can 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 can 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 (such as Reduce and Element-Wise Power ops) to FP32.

Polygraphy can help you diagnose common problems by using reduced precision. Refer to Polygraphy's Working with Reduced Precision how-to guide for more information.

17.5.4. 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 --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> --warmUp=0 --duration=0 --
↪iterations=20
```

Refer to the *trtexec* section for more instructions on using the `trtexec` tool and the meaning of these flags.

If you do not use `trtexec` to measure performance, provide the scripts and commands you use to measure it. Compare the performance measurement from your script with that from the `trtexec` tool. If the two numbers differ, your scripts can have some issues with the performance measurement methodology.

Refer to the *Hardware/Software Environment for Performance Measurements* section for some environmental factors affecting performance.

Chapter 18. Glossary

This glossary defines key terms and concepts used throughout the TensorRT documentation. Terms are organized alphabetically for quick reference.

See also

TensorRT's Capabilities
Overview of TensorRT features and programming model.

Data Format Descriptions
Detailed descriptions of tensor layout formats referenced in this glossary.

TensorRT Support Matrix
Platform, GPU, and precision support for the current release.

How to Use This Glossary:

- Use Ctrl+F (or Cmd+F on Mac) to search for specific terms
- Terms are organized by their first letter
- Related terms are cross-referenced with italics
- Technical API references are linked where applicable

—

A–B

Activation - The output tensor of a layer in a neural network. Activations are intermediate results that flow between layers during inference.

ASIL - Automotive Safety Integrity Level. A risk classification defined by ISO 26262 for functional safety in road vehicles. Levels are A (lowest stringency), B, C, and D (highest stringency); items not safety-related under ISO 26262 are labeled ASIL QM ("Quality Managed"). The TensorRT *Safety Runtime* on NVIDIA DriveOS is developed to ASIL D as a *SEooC*.

Batch - 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 similarly. All instances can, therefore, be computed in parallel.

Builder - TensorRT's model optimizer. The *builder* takes a *network definition* as input, performs device-independent and device-specific optimizations, and creates an *engine*.

C–D

Calibration - The process of determining optimal quantization scaling factors for INT8 inference. Calibration uses representative input data to minimize accuracy loss when converting from higher precision to INT8.

CUDA Graph - A CUDA feature that allows the GPU to launch a sequence of kernels without CPU involvement, reducing launch overhead and improving performance for inference workloads.

Data-Dependent Shape - A tensor shape with a dynamic dimension not calculated solely from network input dimensions and shape tensors.

Device - A specific GPU. Two GPUs are considered identical devices if they have the same model name and configuration.

DLA - Deep Learning Accelerator. A dedicated inference processor on NVIDIA embedded platforms designed for efficient execution of neural networks with lower power consumption than GPU execution. *(DLA is not available in TensorRT 11.0.1 for NVIDIA DriveOS 7.2.5; the DLA-format constants and APIs in libnvinfer remain documented for completeness.)*

Dynamic batch - A mode of inference deployment where the batch size is unknown until runtime. Historically, TensorRT treated batch size as a special dimension and the only configurable dimension at runtime. TensorRT 6 and later allow *engines* to be built such that all dimensions of inputs can be adjusted at runtime.

Dynamic Shape - A tensor dimension whose value is not known until runtime. Dynamic shapes allow a single engine to handle inputs of varying sizes. Refer to *Optimization Profile*.

E–F

Engine - A representation of a model optimized by the TensorRT *builder*. An engine contains the optimized network, kernel selections, and memory management strategy. Engines can be serialized to a *plan* file for later use.

Execution Context - A runtime instance that maintains state for executing inference with a TensorRT *engine*. Multiple contexts can share the same engine, enabling concurrent inference on the same model.

Explicitly Data-Dependent Shape - A tensor shape that depends on the dimensions of an output of `INonZeroLayer` or `INMSLayer`.

Framework Integration - Integration of TensorRT into deep learning frameworks such as PyTorch or TensorFlow, allowing model optimization and inference within the framework. Examples include Torch-TensorRT for PyTorch.

Fusion - An optimization technique where TensorRT combines multiple layers into a single operation to reduce memory bandwidth and kernel launch overhead. For example, convolution, bias, and activation layers might be fused into a single kernel.

I–L

Implicitly Data-Dependent Shape - A tensor shape with a dynamic dimension calculated from data other than network input dimensions, network input shape tensors, or an explicitly data-dependent shape (refer to *Explicitly Data-Dependent Shape*). For example, a shape with a dimension calculated from data output by a convolution.

INT8 - An 8-bit integer data type used for quantized inference. INT8 operations can significantly improve inference performance while maintaining acceptable accuracy when properly calibrated.

Kernel - A GPU function that performs a specific computation. TensorRT selects and tunes kernels for each layer during the build phase to optimize performance for the target hardware.

Layer - A single operation in a neural network, such as convolution, activation, or pooling. In TensorRT, layers are represented in the *network definition* and optimized into efficient GPU *kernels*.

N–O

Network Definition - A representation of a model in TensorRT before optimization. A network definition is a graph of tensors and operators that describes the model's structure and connectivity.

NVRM - NVIDIA Resource Manager. A low-level library on NVIDIA DriveOS that arbitrates access to GPU resources (memory, contexts, channels) and provides the foundation that higher-level NVIDIA runtimes build on, including CUDA, the TensorRT *runtime*, and the TensorRT *Safety Runtime*. NVRM configuration and safety integration are documented in the *NVIDIA DriveOS Developer Guide*.

NVSCI - NVIDIA System Communications Interface. A family of NVIDIA DriveOS APIs - notably **NvSciBuf** for cross-process buffer interchange and **NvSciSync** for cross-process synchronization primitives - used to coordinate data and timing between independent processes, including between the TensorRT *Safety Runtime* on QNX Safety and other DriveOS subsystems. NVSCI behavior and per-process initialization are documented in the *NVIDIA DriveOS Developer Guide*.

ONNX - Open Neural Network eXchange. A framework-independent standard for representing machine learning models. TensorRT uses ONNX as the primary format for importing models from training frameworks. For more information, refer to onnx.ai.

ONNX Parser - A parser for creating a TensorRT *network definition* from an ONNX model.

Optimization Profile - A set of dimensions (minimum, optimal, and maximum) specified for each dynamic input dimension. TensorRT uses profiles to optimize *engines* for specific ranges of input shapes.

P–Q

Plan - An optimized inference *engine* in a serialized format. Applications deserialize the model from the plan file to initialize the inference engine. A typical workflow builds an engine once and then serializes it as a plan file for later use.

Platform - A combination of hardware architecture and operating system. Examples include Linux on x86-64 and QNX on AArch64. Platforms with different architectures or operating systems are considered distinct platforms. *Plan* files are generally platform-specific.

Plugin - A custom layer implementation that extends TensorRT's built-in layer support. Plugins allow you to implement operations not natively supported by TensorRT.

Precision - The numerical format used to represent values during computation. TensorRT supports mixed-precision inference with FP32, TF32, FP16, BF16, FP8, INT8, and INT4 data types. The *builder* selects precisions to balance accuracy and performance based on specified constraints.

PTQ - Post-Training Quantization. A method to quantize a trained model to lower precision (typically INT8) without retraining. PTQ uses *calibration* data to determine quantization parameters.

QAT - Quantization-Aware Training. A training technique that simulates quantization during training, allowing the model to learn parameters that are more robust to quantization. QAT typically achieves better accuracy than *PTQ* for INT8 inference.

Quantization - The process of converting a model from higher precision (FP32/FP16) to lower precision (INT8/INT4) to reduce memory usage and improve performance. Refer to *PTQ* and *QAT*.

R–S

Refit - The process of updating weights in an existing TensorRT *engine* without rebuilding the entire engine. Refitting is faster than rebuilding and useful for updating model parameters without changing the network structure.

Runtime - The component of TensorRT that executes inference on a TensorRT *engine*. The runtime API supports synchronous and asynchronous execution, profiling, and management of *execution contexts*.

Safety Proxy - A non-certified host-side runtime that exposes the same API as the TensorRT *Safety Runtime*. The safety proxy lets developers prototype and test safety-certified applications on platforms that are not themselves safety-certified (for example, NVIDIA DriveOS Linux). The proxy is intended for development only; production deployments use the certified *Safety Runtime* on QNX Safety. Also called the *safety proxy runtime*.

Safety Runtime - A version of the TensorRT runtime certified for use in safety-critical applications, particularly in automotive systems with ISO 26262 requirements. On NVIDIA DriveOS, the safety runtime runs on QNX Safety and is developed to *ASIL* D as a *SEooC*. For host-side prototyping on non-certified platforms, refer to *Safety Proxy*.

SEooC - Safety Element out of Context. An ISO 26262 development concept describing a safety-related element (hardware or software) developed independently of a specific vehicle application and later integrated by a customer into their item-level safety case. NVIDIA delivers TensorRT for DriveOS as a SEooC; integration assumptions, restrictions, and certified state are captured in the *NVIDIA Deep Learning Inference SEooC 2.2 Safety Manual*.

Shape Tensor - A tensor whose values represent the dimensions of another tensor. Shape tensors enable dynamic control of tensor shapes during inference.

Strongly Typed Network - A network where tensor types are explicitly specified and strictly enforced, rather than being automatically selected by TensorRT for performance.

T–W

Tactic - A specific implementation choice for a layer, including kernel selection, tile sizes, and algorithm variants. The *builder* evaluates tactics and selects the fastest option for the target hardware.

Tensor - A multi-dimensional array of data. In TensorRT, tensors represent inputs, outputs, and intermediate *activations* in a neural network.

Timing Cache - A cache of measured kernel performance data. Timing caches can be serialized and reused to speed up subsequent engine builds with similar network structures.

Version Compatibility - A TensorRT feature that allows *engines* built with one TensorRT version to run on later versions, providing forward compatibility across TensorRT releases.

Weight - Learned parameters in a neural network, such as convolution filters or fully connected layer matrices. Weights are optimized during training and remain constant during inference.

Workspace - Temporary GPU memory used by TensorRT during *engine* building and inference. The workspace size can be controlled to balance performance and memory usage.

Copyright