NVIDIA TensorRT Performance

Best Practices | NVIDIA Docs

BPG-09173-001 _v8.0.1 | July 2021

Table of Contents

List of Figures

Chapter 1. How Do I Measure Performance?

Before starting any optimization effort with TensorRT, it's essential to determine what should be measured. Without measurements, it's impossible to make reliable progress or measure whether success has been achieved.

Latency

A performance measurement for network inference is how much time elapses from an input being presented to the network until an output is available. This is the *latency* of a single input. Lower latencies are better. In some applications, low latency is a critical safety requirement. In other applications, latency is directly visible to users as a quality of service issue. For larger bulk processing, latency may not be important at all.

Throughput

Another performance measurement is how many inferences can be completed in a fixed unit of time. This is the *throughput* of the network. Higher throughput is better. Higher throughputs indicate a more efficient utilization of fixed compute resources. For bulk processing, the total time taken will be determined by the throughput of the network.

Before we can start measuring latency and throughput, we need to choose the exact points at which to start and stop timing. Depending on the network and application, it might make sense to choose different points. In many applications, there is a processing pipeline.

The overall system performance can be measured by the latency and throughput of the entire processing pipeline. Because the pre and post-processing steps depend so strongly on the particular application, in this section, we will mostly consider the latency and throughput of the network inference, excluding the data pre and post-processing overhead.

Another way of looking at latency and throughput is to fix the maximum latency and measure throughput at that latency. This is a type of quality-of-service measurement. A measurement like this can be a reasonable compromise between the user experience and system efficiency.

1.1. Tools

If you have a model saved as an ONNX file, or if you have a network description in a Caffe prototxt format, you can use the `trtexec` tool to test the performance of running inference on your network using TensorRT. The `trtexec` tool has many options such as specifying inputs and outputs, iterations and runs for performance timing, precisions allowed, and other options.

For more information about `trtexec`, see trtexec.

If you have a saved serialized engine file, you can use 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.

1.2. CPU Timing

C++11 provides high precision timers in the `<chrono>` standard library. For example, `std::chrono::system_clock` represents wall-clock time, and `std::chrono::high_resolution_clock` measures time in the highest precision available. Every operating system also provides mechanisms for measuring time in high precision.

For example:

Linux

`gettimeofday`

Windows

`QueryPerformanceCounter and QueryPerformanceFrequency`

These mechanisms measure wall-clock time from the host side. 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. When measuring times with asynchronous operations, ensure you add an explicit CUDA stream or device synchronization to wait for results to become available. Alternatively, convert calls from `IExecutionContext::enqueue` to `IExecutionContext::execute` to force the calls to be synchronous.

The following example code snippet shows measuring a network inference execution host time:

```
#include <chrono>

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

These types of wall-clock times can be useful for measuring overall throughput and latency of the application, and for placing inference times in context within a larger system.

1.3. CUDA Events

One problem with timing on the host exclusively is that it requires host/device synchronization. Optimized applications may have many inferences running in parallel on the device with overlapping data movement. In addition, the synchronization itself adds some amount of noise to timing measurements.

To help with these issues, CUDA provides an Event API. This API allows you to place events into CUDA streams that will be time-stamped by the GPU as they are encountered. Differences in timestamps can then tell you how long different operations took.

The following example code snippet shows computing the time between two CUDA events:

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

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

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

TensorRT also includes an optional CUDA event in the method `IExecutionContext::enqueue` that will be signaled once the input buffers are free to be reused. This allows the application to immediately start refilling the input buffer region for the next inference in parallel with finishing the current inference. For example:

```
cudaEvent_t inputReady;
cudaEventCreate(&inputReady);

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

// At this point we can refill the input buffers, but output buffers may not be done
```

1.4. Built-In TensorRT Profiling

To dig deeper into the performance of inference, it requires more fine-grained timing measurements within the optimized network. The `IExecutionContext` interface class provides a method called `setProfiler` that allows you to write a custom class implementing the `IProfiler` interface.

When called, the network will run in a profiling mode. After finishing inference, the profiler object of your class is called to report the timing for each layer in the network. These timings can be used to locate bottlenecks, compare different versions of a serialized engine, and debug performance issues.

Layers inside a loop compile into a single monolithic layer, therefore, separate timings for those layers are not available.

Profiling is currently only enabled for the synchronous `execute` mode when `setProfiler` is called. There is a slight impact on performance when profiling is enabled, therefore, it should only be set up when needed.

An example showing how to use the `IProfiler` interface is provided in the common sample code (`common.h`), and then used in Neural Machine Translation (NMT) Using A Sequence To Sequence (seq2seq) Model (sampleNMT) located in the GitHub repository.

1.5. CUDA Profiling

The recommended CUDA profilers are NVIDIA Nsight Compute and NVIDIA Nsight Systems. Some CUDA developers may be more familiar with nvprof and nvvp, however, these are being deprecated. In any case, these profilers can be used on any CUDA program to report timing information about the kernels launched during execution, data movement between host and device, and CUDA API calls used.

NVIDIA Nsight Systems can be configured in various ways to report timing information for only a portion of the execution of the program or to also report traditional CPU sampling profile information together with GPU information.

Enabling NVIDIA Tools Extension SDK (NVTX) tracing allows NVIDIA Nsight Compute and Nsight Systems to collect data generated by TensorRT applications. NVTX is a C-based API for marking events and ranges in your applications.

Note: In TensorRT, each layer may launch one or more kernels to perform its operations. The exact kernels launched depends on the optimized network and the hardware present. Depending on the choices of the builder, there may be multiple additional operations that reorder data interspersed with layer computations; these reformat operations may be implemented as either device-to-device memory copies or as custom kernels.

Decoding the kernel names back to layers in the original network can be complicated. Because of this, TensorRT uses NVTX to mark a range for each layer, which then allows the CUDA profilers to correlate each layer with the kernels called to implement it. In TensorRT, NVTX helps to correlate the runtime engine layer execution with CUDA kernel calls. Nsight Systems supports collecting and visualizing these events and ranges on the timeline. Nsight Compute also supports collecting and displaying the state of all active NVTX domains and ranges in a given thread when the application is suspended.

For example, the following screenshots are from Nsight Systems.

Figure 1. The layer execution and the kernel being launched on the CPU side.

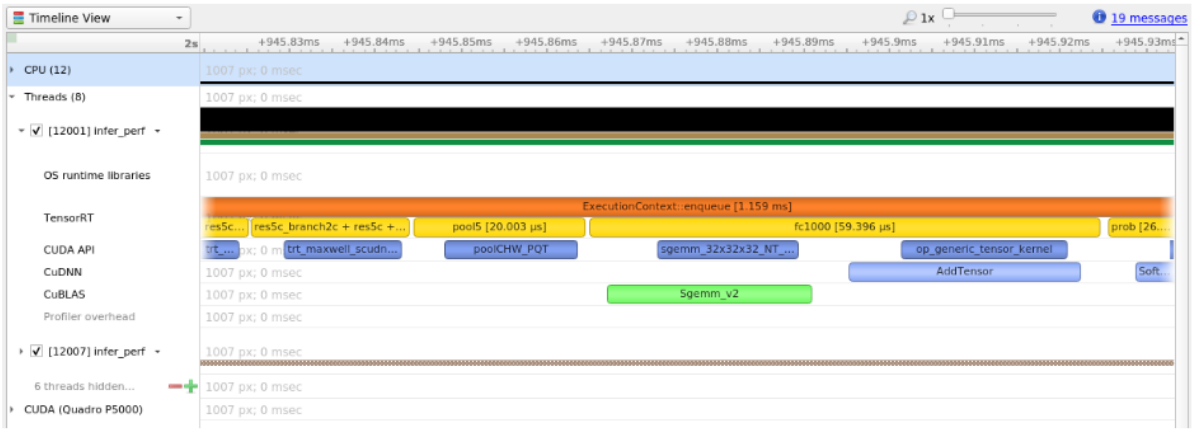

Figure 2. The kernels actually run on the GPU, in other words, it shows the correlation between the layer execution and kernel launch on the CPU side and their execution on the GPU side.

When profiling a TensorRT application, it is recommended to enable profiling only after the engine has been built. During the build phase, all possible tactics are tried and timed. Profiling this portion of the execution will not show any meaningful performance measurements and will include all possible kernels, not the ones actually selected for inference. One way to limit the scope of profiling is to:

First phase
Structure the application to build and then serialize the engines in one phase.

Second phase
Load the serialized engines and run inference in a second phase.

Third phase
Profile this second phase only.

WARNING:

Layers inside loops may be greatly fused and show up as `__myln_k_bb[n]_[m]` where n and m are integers.

1.6. Memory

Tracking memory usage can be as important as execution performance. Usually, the memory will be more constrained on the device than on the host. To keep track of device memory, the recommended mechanism is to create a simple custom GPU allocator that internally keeps some statistics then uses the regular CUDA memory allocation functions `cudaMalloc` and `cudaFree`.

A custom GPU allocator can be set for the builder `IBuilder` for network optimizations, and for `IRuntime` when deserializing engines. One idea for the custom allocator is to keep track of the current amount of memory allocated, and to push an allocation event with a timestamp and other information onto a global list of allocation events. Looking through the list of allocation events allows profiling memory usage over time. For guidance on how to determine the amount of memory a model will use, see FAQs, question *How do I determine how much device memory will be required by my network?*.

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 scrip 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
```

Chapter 2. How Do I Optimize My TensorRT Performance?

The following sections focus on the general inference flow on GPUs and some of the general strategies to improve performance. These ideas are applicable to most CUDA programmers but may not be as obvious to developers coming from other backgrounds.

2.1. Mixed Precision

TensorRT supports Mixed Precision Inference with FP32, FP16, or INT8 as supported precisions. Depending on the hardware support, you can choose to enable either of the above precision to accelerate inference.

FP32 precision is the default precision if no specific precision mode is enabled. When FP16 precision mode is enabled, a layer can either execute in FP32 or FP16 based on fastest execution time. Similarly, if INT8 precision mode is enabled, a layer can either execute in FP32 or INT8 based on fastest execution time.

For the best performance, you can choose to enable all three precisions by enabling FP16 and INT8 precision mode explicitly. You can also choose to execute `trtexec` with the "`--best`" option directly, which would enable all supported precisions for inference resulting in best performance.

For more information, refer to the TensorRT Support Matrix.

2.2. Batching

The most important optimization is to compute as many results in parallel as possible using batching. In TensorRT, a *batch* is a collection of inputs that can all be processed uniformly. Each instance in the batch has the same shape and flows through the network in exactly the same way. Each instance can, therefore, be trivially computed in parallel.

Each layer of the network will have some amount of overhead and synchronization required to compute forward inference. By computing more results in parallel, this overhead is paid off more efficiently. In addition, many layers are performance-limited by the smallest dimension in the input. If the batch size is one or small, this size can often be the performance limiting dimension. For example, the FullyConnected layer with `V` inputs and `K` outputs can be implemented for one batch instance as a matrix multiply of an `1xV` matrix with a `VxK` weight

matrix. If `N` instances are batched, this becomes an `NxV` multiplied by `VxK` matrix. The vector-matrix multiplier becomes a matrix-matrix multiplier, which is much more efficient.

Larger batch sizes are almost always more efficient on the GPU. Extremely large batches, such as `N > 2^16`, can sometimes require extended index computation and so should be avoided if possible. But generally, increasing the batchsize improves total throughput. In addition, when the network contains MatrixMultiply layers or FullyConnected layers, batch sizes of multiples of 32 tend to have the best performance for FP16 and INT8 inference because of the utilization of Tensor Cores, if the hardware supports them.

Sometimes batching inference work is not possible due to the organization of the application. In some common applications, such as a server that does inference per request, it can be possible to implement opportunistic batching. For each incoming request, wait for a time `T`. If other requests come in during that time, batch them together. Otherwise, continue with a single instance inference. This type of strategy adds fixed latency to each request but can improve the maximum throughput of the system by orders of magnitude.

Using batching

The C++ and Python APIs are designed for batch input. The `IExecutionContext::execute` (`IExecutionContext.execute` in Python) and `IExecutionContext::enqueue` (`IExecutionContext.execute_async` in Python) methods take an explicit batch size parameter. The maximum batch size should also be set for the builder when building the optimized network with `IBuilder::setMaxBatchSize` (`Builder.max_batch_size` in Python). When calling `IExecutionContext::execute` or `enqueue`, the bindings passed as the `bindings` parameter are organized per tensor and not per instance. In other words, the data for one input instance is not grouped together into one contiguous region of memory. Instead, each tensor binding is an array of instance data for that tensor.

Another consideration is that building the optimized network optimizes for the given maximum batch size. The final result will be tuned for the maximum batch size but will still work correctly for any smaller batch size. It is possible to run multiple build operations to create multiple optimized engines for different batch sizes, then choose which engine to use based on the actual batch size at runtime.

2.3. Streaming

In general, CUDA programming streams are a way of organizing asynchronous work. Asynchronous commands put into a stream are guaranteed to run in sequence but may execute out of order with respect to other streams. In particular, asynchronous commands in two streams may be scheduled to run concurrently (subject to hardware limitations).

In the context of TensorRT and inference, each layer of the optimized final network will require work on the GPU. However, not all layers will be able to fully utilize the computation capabilities of the hardware. Scheduling requests in separate streams allows work to be scheduled immediately as the hardware becomes available without unnecessary synchronization. Even if only some layers can be overlapped, overall performance will improve.

Using streaming

1. Identify the batches of inferences that are independent.
2. Create a single engine for the network.
3. Create a CUDA stream using `cudaStreamCreate` for each independent batch and an `IExecutionContext` for each independent batch.
4. Launch inference work by requesting asynchronous results using `IExecutionContext::enqueue` from the appropriate `IExecutionContext` and passing in the appropriate stream.
5. After all the work has been launched, synchronize with all the streams to wait for results. The execution contexts and streams can be reused for later batches of independent work.

With the help of streaming, the Triton Inference Server helps to manage multiple execution instances of a model. For more information about how Triton Inference Server does this, see Instance Groups.

It is also possible to use multiple host threads with streams. A common pattern is incoming requests dispatched to a pool of waiting for worker threads. In this case, the pool of worker threads will each have one execution context and CUDA stream. Each thread will request work in its own stream as the work becomes available. Each thread will synchronize with its stream to wait for results without blocking other worker threads.

2.4. CUDA Graphs

Sometimes, inference performance is not bottlenecked by GPU computations, but instead is bottlenecked by the duration of the `enqueue()`/`enqueueV2()` calls of the TensorRT execution context. This happens when the workload is running with small batch sizes or when the network contains many layers with short kernel execution time, causing the kernel launch time to dominate the inference latency.

This usually manifests as `trtexec` reporting a longer Enqueue Time than GPU Compute Time, or Nsight Systems profiles showing that the duration of the kernel launches is longer than that of the kernel executions.

In this case, CUDA Graphs become a useful feature to shorten the `enqueue()`/`enqueueV2()` duration and speed-up inference. The following example code shows how to capture a CUDA graph instance and run inference with CUDA graphs.

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

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

Note that the binding locations cannot be altered after the graph has been captured, unless the graph is re-captured again with new bindings. Also, not all TensorRT engines support CUDA graph capturing, especially those containing loops or recurrent parts. `trtexec` provides example code about checking whether the built TensorRT engine can be compatible with CUDA graph capture at runtime. When CUDA graph capture fails, the inference will fallback to be launched without using CUDA graph launch.

You can execute `trtexec` with the `--useCudaGraph` argument to get the inference performance of your workload with CUDA graphs. This argument may be ignored when the built TensorRT engine contains operations that are not permitted under CUDA graph capture mode.

2.5. Thread Safety

The TensorRT builder may only be used by one thread at a time. If you need to run multiple builds simultaneously, you will need to create multiple builders.

The TensorRT runtime can be used by multiple threads simultaneously, so long as each object uses a different execution context.

Note: Plugins are shared at the engine level, not the execution context level, and thus plugins which may be used simultaneously by multiple threads need to manage their resources in a thread-safe manner. This is however not required for plugins based on `IPluginV2Ext` and derivative interfaces since we clone these plugins when `ExecutionContext` is created.

The TensorRT library pointer to the logger is a singleton within the library. If using multiple builder or runtime objects, use the same logger, and ensure that it is thread-safe.

2.6. Initializing The Engine

In general, creating an engine from scratch is an expensive operation. The builder optimizes the given network in various ways, then performs timing tests to choose the highest performance implementation for each layer specific to the actual GPU in the system. As the number of layers in the network increases, the number of possible configurations increases and the time taken to choose the optimal one also increases.

The builder layer timing cache helps to reduce the time taken in the builder phase. The caching should work in all cases and even better than non-caching in some cases, however, there can be cases where turning it off may give you marginally better performance.

More complicated deployment scenarios can involve multiple networks for the same application or even multiple applications running at the same time. The recommended strategy in these scenarios is to create engines and serialize them before they are needed. An engine can be deserialized relatively quickly. One engine can then be used to create multiple `IExecutionContext` objects.

2.7. Enabling Fusion

The following sections discuss the different options for enabling fusion.

2.7.1. Layer Fusion

TensorRT attempts to perform many different types of optimizations in a network during the build phase. In the first phase, layers are fused together whenever possible. Fusions transform the network into a simpler form but preserve the same overall behavior. Internally, many layer implementations have extra parameters and options that are not directly accessible when creating the network. Instead, the fusion optimization step detects supported patterns of operations and fuses multiple layers into one layer with internal options set.

Consider the common case of a convolution followed by ReLU activation. To create a network with these operations, it involves adding a Convolution layer with `addConvolution`, following it with an Activation layer using `addActivation` with an `ActivationType` of `kRELU`. The unoptimized graph will contain separate layers for convolution and activation. The internal implementation of convolution supports computing the ReLU function on the output in one step directly from the convolution kernel without requiring a second kernel call. The fusion optimization step will detect the convolution followed by ReLU, verify that the operations are supported by the implementation, then fuse them into one layer.

To investigate which fusions have happened, or has not happened, the builder logs its operations to the logger object provided during construction. Optimization steps are at the `kINFO` log level. To see these messages, ensure you log them in the `ILogger` callback.

Fusions are normally handled by creating a new layer with a name containing the names of both of the layers which were fused. For example, in MNIST, a FullyConnected layer (InnerProduct) named `ip1` is fused with a ReLU Activation layer named `relu1;` to create a new layer named `ip1 + relu1`.

2.7.2. Types Of Fusions

The following list describes the types of supported fusions.

Supported Layer Fusions

ReLU ReLU Activation

An Activation layer performing ReLU followed by an activation performing ReLU will be replaced by a single activation layer.

Convolution and ReLU Activation

The Convolution layer can be of any type and there are no restrictions on values. The Activation layer must be ReLU type.

Convolution and GELU Activation

The precision of input and output should be the same; with both of them FP16 or INT8. The Activation layer must be GELU type. TensorRT should be running on a Turing or later device with CUDA version 10.0 or later.

Convolution and Clip Activation

The Convolution layer can be any type and there are no restrictions on values. The Activation layer must be Clip type.

Scale and Activation

The Scale layer followed by an Activation layer can be fused into a single Activation layer.

Convolution And ElementWise Operation

A Convolution layer followed by a simple sum, min, or max in an ElementWise layer can be fused into the Convolution layer. The sum must not use broadcasting, unless the broadcasting is across the batch size.

Padding and Convolution/Deconvolution

Padding followed by a Convolution or Deconvolution can be fused into a single Convolution/Deconvolution layer if all the padding sizes are non-negative.

Shuffle and Reduce

A Shuffle layer without reshape, followed by a Reduce layer can be fused into a single Reduce layer. The Shuffle layer can perform permutations but cannot perform any reshape operation. The Reduce layer must have `keepDimensions` set of dimensions.

Shuffle and Shuffle

Each Shuffle layer consists of a transpose, a reshape, and a second transpose. A Shuffle layer followed by another Shuffle layer can be replaced by a single Shuffle (or nothing). If both Shuffle layers perform reshape operations, this fusion is only allowed if the second transpose of the first shuffle is the inverse of the first transpose of the second shuffle.

Scale

A Scale layer that adds `0`, multiplied by `1`, or computes powers to the `1` can be erased.

Convolution and Scale

A Convolution layer followed by a Scale layer that is `kUNIFORM` or `kCHANNEL` can be fused into a single convolution by adjusting the convolution weights. This fusion is disabled if the scale has a non-constant `power` parameter.

Reduce

A Reduce layer that performs average pooling will be replaced by a Pooling layer. The Reduce layer must have `keepDimensions` set, reduced across `H` and `W` dimensions from `CHW` input format before batching, using the `kAVG` operation.

Convolution and Pooling

The Convolution and Pooling layers must have the same precision. The Convolution layer may already have a fused activation operation from a previous fusion.

Depthwise Separable Convolution

A depthwise convolution with activation followed by a convolution with activation may sometimes be fused into a single optimized DepSepConvolution layer. The precision of both convolutions must be INT8 and the device computes capability must be 7.2 or later.

SoftMax and Log

Can be fused into a single Softmax layer if the SoftMax has not already been fused with a previous log operation.

SoftMax and TopK

It can be fused into a single layer. The SoftMax may or may not include a Log operation.

FullyConnected

The FullyConnected layer will be converted into the Convolution layer, all fusions for convolution will take effect.

Supported Reduction Operation Fusions

GELU

A group of Unary layer and ElementWise layer which represent the following equations can be fused into a single GELU reduction operation.

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

Or the alternative representation.

$$0.5x \times \left(1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right)$$

L1Norm

A Unary layer `kABS` operation followed by a Reduce layer `kSUM` operation can be fused into a single L1Norm reduction operation.

Sum of Squares

A product ElementWise layer with the same input (square operation) followed by a `kSUM` reduction can be fused into a single square Sum reduction operation.

L2Norm

A sum of squares operation followed by a `kSQRT` UnaryOperation can be fused into a single L2Norm reduction operation.

LogSum

A Reduce layer `kSUM` followed by a `kLOG` UnaryOperation can be fused into a single LogSum reduction operation.

LogSumExp

A Unary `kEXP` ElementWise operation followed by a LogSum fusion can be fused into a single LogSumExp reduction.

2.7.3. MLP Fusions

Multilayer Perceptron (MLP) networks can be described as stacked layers of FullyConnected or MatrixMultiply layers interleaved with Activation layer functions. To improve the performance of Multilayer Perceptron networks, different types of fusions are possible.

The initial creation of a dedicated MLP layer comes from a MatrixMultiply layer fused with an Activation layer. The MatrixMultiply layer must be a 2D multiplication. The size of the matrices must be small enough to use hardware shared memory to store temporary weights; for the untransposed case, this means the product of the widths of both matrices must be limited (heights if transposed).

Other patterns supported for the creation of the initial MLP layer are fusing a MatrixMultiply with an ElementWise `kSUM` operation with a constant, for example bias, and fusing two MatrixMultiply layers together with no intermediate computation.

It is also possible to create the initial MLP layer from fusing a FullyConnected layer with an Activation layer, fusing a FullyConnected layer with a Scale layer (performing bias only using the `shift` parameter), and fusing two FullyConnected layers with no intermediate computation.

Once an MLP layer is created, it will be reported in the builder log as a `1-layer MLP` layer (or a `2-layer MLP` layer if two MatrixMultiply or FullyConnected layers were merged). This layer can then also be fused with more layers to create deeper MLP fusions.

MLP layers can be fused with subsequent MatrixMultiply, FullyConnected, Activation, ElementWise sums, and Scale layers. The general restrictions are that:

- MatrixMultiply must be strictly 2D
- ElementWise must be a `kSUM` with a constant
- Scale must be a bias using the `shift` parameter

All activations are supported. The size of matrices being multiplied must allow shared memory for weight reuse as described for initial MLP layer creation.

Two MLP layer nodes can also be fused into one larger MLP layer. The total number of layers is limited to 31. The last layer of the first MLP layer must match the input of the second MLP layer.

Because these fusions are general, sometimes networks not designed as strictly as Multilayer Perceptron networks will use MLP layers as an automatic optimization. For example, the MNIST sample contains an InnerProduct layer followed by a ReLU activation, followed by another InnerProduct layer. InnerProduct from Caffe is parsed as a FullyConnected layer in TensorRT. The `ip1` and `relu1` layers are fused into `ip1 + relu1` as described previously. This layer is then fused with `ip2` into a new layer named `2-layer MLP`.

2.7.4. 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
All `ActivationType` is supported.

Constant
Only constant with single value (size == 1).

ElementWise
All `ElementWiseOperation` are supported.

PointWise
`PointWise` itself is also a PointWise layer.

Scale
Only support `ScaleMode::kUNIFORM`.

Unary
All `UnaryOperation` are supported.

Safe mode does not support PointWise fusion.

The size of the fused PointWise layer is not unlimited, therefore, some PointWise layers may not be fused.

Fusion will create a new layer with a name consisting of both of the layers which were fused. For example, an ElementWise layer named `add1` is fused with a ReLU Activation layer named `relu1` with a new layer name: `fusedPointwiseNode(add1, relu1)`.

2.7.5. Q/DQ Fusion

Quantized INT8 graphs generated from QAT tools like NVIDIA's Quantization Toolkit for PyTorch consists of `onnx::QuantizeLinear` and `onnx::DequantizeLinear` pair of nodes (Q/DQ) with scales and zero-points. Starting in TensorRT 7.0, it's required that `zero_point` is `0`.

Q/DQ nodes help convert from FP32 values to INT8 and vice-versa. Such a graph would still have weights and bias in FP32 precision.

Weights are followed by a Q/DQ node pair so that they can be quantized/dequantize if required. Bias quantization is performed using scales from activations and weights, thus no extra Q/DQ node pair is required for bias input. Assumption for bias quantization is that `S_weights * S_input = S_bias`.

Fusions related to Q/DQ nodes include quantizing/dequantizing weights, commutating Q/DQ nodes without changing the mathematical equivalence of the model, and erasing redundant Q/DQ nodes. After applying Q/DQ fusions, the rest of the builder optimizations would be applied to the graph.

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

If we have a

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

(`[DQ, DQ] > Node > Q`) sequence, then it is fused to the quantized node (`QNode`).

Supporting Q/DQ nodes pair for `weights` requires weighted nodes to support more than one input. Thus we support adding second input (for weights tensor) and third input (for bias tensor). Additional inputs can be set using `setInput(index, tensor)` API for Convolution, Deconvolution and FullyConnected layers where index = 2 for weights tensor and index = 3 for bias tensor.

During fusion with weighted nodes, we would quantize FP32 weights to INT8 and fuse it with the corresponding weighted node. Similarly, FP32 bias would be quantized to INT32 and fused.

Fuse Q/DQ with non-weighted node

If we have a `DequantizeLinear > Node > QuantizeLinear` (`DQ > Node > Q`) sequence, then it is fused to the quantized node (`QNode`).

Commutate Q/DQ nodes

`DequantizeLinear` commutation is allowed when `Φ(DQ(x)) == DQ(Φ(x))`.
`QuantizeLinear` commutation is allowed when `Q(Φ(x)) == Φ(Q(x))`.

Also, commutation logic also accounts for available kernel implementations such that mathematical equivalence is guaranteed.

Insert missing Q/DQ nodes

If a node has a missing Q/DQ nodes pair, and `max(abs(Φ(x)) == max(abs(x))` (for example, MaxPool), missing Q/DQ pairs would be inserted to run more node with INT8 precision.

Erase redundant Q/DQ nodes

It's possible that after applying all the optimizations, the graph still has Q/DQ node pairs which are in itself a no-op. Q/DQ node erasure fusion would remove such redundant pairs.

2.8. Structured Sparsity

NVIDIA Ampere GPUs support Structured Sparsity. To make use of this feature to achieve higher inference performance, the convolution kernel weights and/or the fully-connected weights must meet the following requirements:

For each output channel and for each spatial pixel in the kernel weights, every 4 input channels must have at least 2 zeros. In other words, assuming that the kernel weights have the shape `[K, C, R, S]` and `C % 4 == 0`, then the requirement is:

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

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

```
builderConfig.setFlag(BuilderFlag::kSPARSE_WEIGHTS);
```

At the end of the TensorRT logs when the TensorRT engine is built, TensorRT reports which layers contain weights that meet the structures sparsity requirement, and in which layers TensorRT selects tactics that make use of the structured sparsity. In some cases, tactics with structured sparsity can be slower than normal tactics and TensorRT will choose normal

tactics in these cases. The following output shows an example of TensorRT logs showing information about sparsity:

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

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

To get inference performance measurements with structured sparsity using `trtexec`, pass the `--sparsity=<mode>` flag, where `<mode>` can be:

- `disable`: Disable all tactics using structured sparsity. This is the default.
- `enable`: Enable tactics using structured sparsity, but they will only be used if the weights in the ONNX file meet the requirements for structured sparsity.
- `force`: Enable tactics using structured sparsity and allow `trtexec` to overwrite the weights in the ONNX file to enforce them to have structured sparsity patterns. Note that the accuracy is not preserved, so this is to get inference performance only.

Chapter 3. How Do I Optimize My Layer Performance?

The following descriptions detail how you can optimize the listed layers.

Concatenation Layer

The main consideration with the Concatenation layer is that if multiple outputs are concatenated together, they can not be broadcasted across the batch dimension and must be explicitly copied. Most layers support broadcasting across the batch dimension to avoid copying data unnecessarily, but this will be disabled if the output is concatenated with other tensors.

Gather Layer

To get the maximum performance out of a Gather layer, use an `axis` of `0`. There are no fusions available for a Gather layer.

MatrixMultiply and FullyConnected Layers

A new development is encouraged to use MatrixMultiply in preference to FullyConnected layers for consistency of interface. Matrix multiplication is generally significantly faster in FP16 Tensor Cores compared to FP32.

Tensor dimensions (or the number of input and output channels for FullyConnected layer) 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. Tensor Core kernels for FP16 data require striding between data rows to be multiples of 8 data elements. For example, a MatrixMultiply that is `M x K` times `K x N` requires `M`, `K`, and `N` to be multiple of 8 to use Tensor Core optimized kernels.

Reduce Layer

To get the maximum performance out of a Reduce layer, perform the reduction across the last dimensions (tail reduce). This allows optimal memory to read/write patterns through sequential memory locations. If doing common reduction operations, express the reduction in a way that will be fused to a single operation if possible.

RNN Layer

If possible, opt to use the newer RNNv2 interface in preference to the legacy RNN interface. The newer interface supports variable sequence lengths and variable batch sizes, as well as having a more consistent interface. To get maximum performance, larger batch sizes are better. In general, sizes that are multiples of 64 achieve highest performance. Bidirectional RNN-mode prevents wavefront propagation because of the added dependency, therefore, it tends to be slower.

In addition, the newly introduced ILoop-based API provides a much more flexible mechanism to use general layers within recurrence without being limited to a small set of predefined RNNv2 interface. The ILoop recurrence enables a rich set of automatic loop optimizations, including loop fusion, unrolling, and loop-invariant code motion, to name a few. For example, significant performance gains are often obtained when multiple instances of the same MatrixMultiply or FullyConnected layer are properly combined to maximize machine utilization after loop unrolling along the sequence dimension. This works best if you can avoid a MatrixMultiply or FullyConnected layer with a recurrent data dependence along the sequence dimension.

TopK

To get the maximum performance out of a TopK layer, use small values of `K` reducing the last dimension of data to allow optimal sequential memory accesses. Reductions along multiple dimensions at once can be simulated by using a Shuffle layer to reshape the data, then reinterpreting the index values appropriately.

For more information about layers, see TensorRT Layers

Chapter 4. How Do I Optimize My Plugins?

TensorRT provides a mechanism for registering custom plugins that perform layer operations. After a plugin creator is registered, you can look up the registry to find the creator and add the corresponding plugin object to the network during serialization/deserialization.

All TensorRT plugins are automatically registered once the plugin library is loaded. For more information about custom plugins, see Extending TensorRT With Custom Layers.

The performance of plugins depends on the CUDA code performing the plugin operation. Standard CUDA best practices apply. When developing plugins, it can be helpful to start with simple standalone CUDA applications that perform the plugin operation and verify correctness. The plugin program can then be extended with performance measurements, more unit testing, and alternate implementations. After the code is working and optimized, it can be integrated as a plugin into TensorRT.

To get the best performance possible in FP16 mode, it is important to support as many formats as possible in the plugin. This removes the need for internal reformat operations during the execution of the network. Currently, plugins can support:

- FP32 `NCHW`
- FP16 `NCHW`
- FP16 `N(C/2)HW2` (Half2 format)
- FP16 `NHWC8` format (8-element packed channels; `C` is a multiple of 8)

For more information, see Data Format Descriptions.

Chapter 5. How Do I Optimize My Python Performance?

When using the Python API, most of the same performance considerations apply. When building engines, the builder optimization phase will normally be the performance bottleneck; not API calls to construct the network. Inference time should be nearly identical between the Python API and C++ API.

Setting up the input buffers in the Python API involves using `pycuda` or another CUDA Python library, like `cupy`, to transfer the data from the host to device memory. The details of how this works will depend on where the host data is coming from. Internally, `pycuda` supports the Python Buffer Protocol which allows efficient access to memory regions. This means that if the input data is available in a suitable format in `numpy` arrays or another type that also has support for the buffer protocol, this allows efficient access and transfer to the GPU. For even better performance, ensure that you allocate a page-locked buffer using `pycuda` and write your final preprocessed input there.

For more information about using the Python API, see Working With TensorRT Using The Python API.

Chapter 6. How Do I Improve My Model Accuracy?

TensorRT can execute a layer in FP32, FP16, or INT8 precision depending on the builder configuration. By default, TensorRT chooses to run a layer in a precision which results in optimal performance. Sometimes this can result in poor accuracy due to various reasons. Generally, running a layer in higher precision helps improve accuracy with some performance hit.

There are several steps we can take to improve model accuracy:

1. Validate layer outputs:
    a). Use Polygraphy to dump layer outputs and verify there are no NaNs or Infs. The `--validate` option can check for NaNs and Infs. Also, we can compare layer outputs with golden values from, for example, ONNX runtime.
    b). For FP16, it is possible that a model might require retraining to ensure that intermediate layer output can be represented in FP16 precision without overflow/ underflow.
    c). For INT8, consider recalibrating with a more representative calibration data set. If your model comes from PyTorch, we also provide NVIDIA's Quantization Toolkit for PyTorch for QAT in the framework besides PTQ in TensorRT. You can try both approaches and choose the one with more accuracy.
2. Manipulate layer precision:
    a). Sometimes running a layer in certain precision results in incorrect output. This can be due to inherent layer constraints (for example, LayerNorm output should not be INT8), model constraints (output gets diverged resulting in poor accuracy), or report a TensorRT bug.
    b). You can control layer execution precision and output precision using this API.
    c). An experimental debug precision tool can help automatically find layers to run in high precision.
3. Use an algorithm selector to disable flaky tactics:
    a). When accuracy changes between from build+run to build+run, it might be due to a selection of a bad tactic for a layer.
    b). Use an algorithm selector to dump tactics from both good and bad runs. Configure the algorithm selector to allow only a subset of tactics (i.e. just allow tactics from a good run etc).

c). You can use Polygraphy to automate this process.

Accuracy from run-to-run variation should not change; once the engine is built for a specific GPU, it should result in bit accurate outputs in multiple runs. If not, file a TensorRT bug.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

VESA DisplayPort

DisplayPort and DisplayPort Compliance Logo, DisplayPort Compliance Logo for Dual-mode Sources, and DisplayPort Compliance Logo for Active Cables are trademarks owned by the Video Electronics Standards Association in the United States and other countries.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ARM

ARM, AMBA and ARM Powered are registered trademarks of ARM Limited. Cortex, MPCore and Mali are trademarks of ARM Limited. All other brands or product names are the property of their respective holders. "ARM" is used to represent ARM Holdings plc; its operating company ARM Limited; and the regional subsidiaries ARM Inc.; ARM KK; ARM Korea Limited.; ARM Taiwan Limited; ARM France SAS; ARM Consulting (Shanghai) Co. Ltd.; ARM Germany GmbH; ARM Embedded Technologies Pvt. Ltd.; ARM Norway, AS and ARM Sweden AB.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, CUDA Toolkit, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, GPU, JetPack, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NVCaffe, NVIDIA Ampere GPU architecture, NVIDIA Deep Learning SDK, NVIDIA Developer Program, NVIDIA GPU Cloud, NVLink, NVSHMEM, PerfWorks, Pascal, SDK Manager, T4, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, Triton Inference Server, Turing, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com