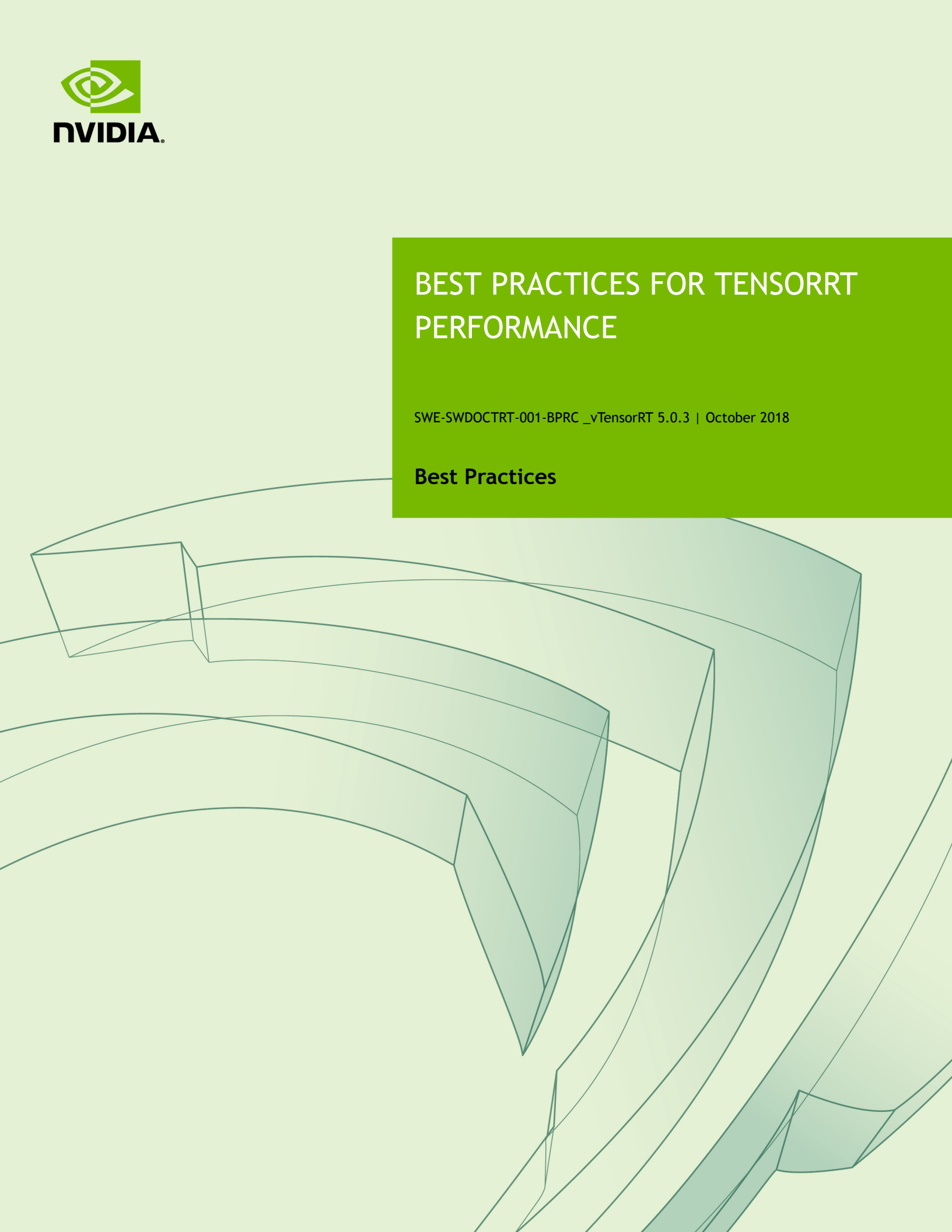

BEST PRACTICES FOR TENSORRT PERFORMANCE

SWE-SWDOCTRT-001-BPRC _vTensorRT 5.0.3 | October 2018

Best Practices

TABLE OF CONTENTS

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

The simplest 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

The next most important measurement is how many inferences can be completed in a fixed unit of time. This is the *throughput* of the network. Higher throughputs are better. Higher throughputs indicate 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 itself; starting from input data that is already present on the GPU, until all network outputs are available on the GPU.

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 a UFF 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 for specifying inputs and outputs, iterations and runs for performance timing, precisions allowed, and other options.

If you have a saved serialized engine file, you can use the yais tool to run the engine with multiple execution contexts from multiple threads in a fully pipelined asynchronous way to test parallel inference performance.

For more information about **trtexec**, see Command Line Wrapper.

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
 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. Alternately, 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->enqueue(batchSize, &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->enqueue(batchSize, &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 signalled 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->enqueue(batchSize, &buffers[0], stream, &inputReady);
cudaEventSynchronize(inputReady);

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

1.4. 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.

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 setup when needed.

An example showing how to use the **`IProfiler`** interface is provided in the common sample code (**`common.h`**), and then used in sampleNMT.

1.5. NVProf

CUDA provides a command line profiler called **`nvprof`**. This profiler 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. It can be configured in various ways to report only timing information for a portion of execution of the program, or to also report traditional CPU sampling profile information together with GPU information.

The **`nvprof`** tool can be run with TensorRT applications. It is recommended to turn on as much debug information as possible in the application to provide the most information in the profiler output.

For 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 many additional operations that reorder data interspersed with layer computations. Some reformat operations may be implemented as device-to-device memory copies, others with custom kernels.

Decoding the kernel names back to layers in the original network can be complicated. Some kernels are templated with many types and sizes as arguments. Some names include internal block sizes and other configuration information. When interpreting results from the profiler, it is recommended to start with the **`IProfiler`** interface to get per-layer timing information before using **`nvprof`** to get per-kernel timing information.

When running **`nvprof`**, it is recommended to only enable it after the engine has been built. During the build phase, all possible tactics are tried and timed. Using **`nvprof`** for 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 **`nvprof`** 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
Run **`nvprof`** on this second phase only.

1.6. Memory

Tracking memory usage can be as important as execution performance. Usually, 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?*.

You can use the yais tool to track the base pointer and size of a memory allocation (see yais/Memory.h. For more information about using the default **`cudaMallocManagedRuntime`** that saves weights to **`cudaMallocManaged`** memory, see yais/TensorRT.cc.

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. 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 multiply becomes a matrix-matrix multiply, 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. Often the time taken to compute results for batch size **`N=1`** is almost identical to batch sizes up to **`N=16`** or **`N=32`**. In this case, increasing the batch size from **`N=1`** to **`N=32`** would dramatically improve total throughput with only a small effect on latency. 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 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 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.

2.2. 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 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.

For an example using streaming, see yais/inference.cc.

It is also possible to use multiple host threads with streams. A common pattern is incoming requests dispatched to a pool of waiting 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.

2.3. 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.

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.

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.

2.4. 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.

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.

2.5. Enabling Fusion

2.5.1. Layer Fusion

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

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

To investigate which fusions have happened, or 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`**.

2.5.2. Types Of Fusions

Supported Layer Fusions

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.

FullyConnected and ReLU Activation

The FullyConnected layer has no restrictions. The Activation layer must be ReLU type.

Scale and Activation

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

Convolution And ElementWise Sum

A Convolution layer followed by a simple Sum 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.

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.

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, reduce across **`H`** and **`W`** dimensions from **`CHW`** input format before batching, using the **`kAVG`** operation.

Supported Reduction Operation Fusions

L1Norm

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

Sum of Square

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.

For more information about layers, see TensorRT Layers

2.5.3. MLP Fusions

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

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 is 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`**.

Chapter 3. HOW DO I OPTIMIZE MY LAYER PERFORMANCE?

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 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
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. FullyConnected supports INT8 Tensor Cores format but MatrixMultiply does not. Therefore, performance improvements may be possible by switching to FullyConnected when using INT8 datatypes on hardware that supports INT8 Tensor Cores.

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.

Reduce Layer
To get maximum performance out of a Reduce layer, perform the reduction across the last dimensions (tail reduce). This allows optimal memory 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.

TopK

To get 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.

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 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 when **`execute`** or **`execute_async`** is called through the Python API as opposed to the C++ API.

Setting up the input buffers in the Python API involves using **`pycuda`** 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.

Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do 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 guide. NVIDIA does not accept any 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 guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DALI, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2018 NVIDIA Corporation. All rights reserved.