NVIDIA cuDNN

Developer Guide | NVIDIA Docs

PG-06702-001_v8.8.0 | April 2024

Table of Contents

List of Figures

List of Tables

Chapter 1. Overview

NVIDIA® CUDA® Deep Neural Network LIbrary (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of operations arising frequently in DNN applications:

- Convolution forward and backward, including cross-correlation
- Matrix multiplication
- Pooling forward and backward
- Softmax forward and backward
- Neuron activations forward and backward: `relu`, `tanh`, `sigmoid`, `elu`, `gelu`, `softplus`, `swish`
- Arithmetic, mathematical, relational, and logical pointwise operations (including various flavors of forward and backward neuron activations)
- Tensor transformation functions
- LRN, LCN, batch normalization, instance normalization, and layer normalization forward and backward

Beyond just providing performant implementations of individual operations, the library also supports a flexible set of multi-operation fusion patterns for further optimization. The goal is to achieve the best available performance on NVIDIA GPUs for important deep learning use cases.

In cuDNN version 7 and older, the API was designed to support a fixed set of operations and fusion patterns. We informally call this the “legacy API”. Starting in cuDNN version 8, to address the quickly expanding set of popular fusion patterns, we added a graph API, which allows the user to express a computation by defining an operation graph, rather than by selecting from a fixed set of API calls. This offers better flexibility versus the legacy API, and for most use cases, is the recommended way to use cuDNN.

Note that while the cuDNN library exposes a C API, we also provide an open source C++ layer which wraps the C API and is considered more convenient for most users. It is, however, limited to just the graph API, and does not support the legacy API.

Chapter 2. Core Concepts

Before we discuss the details of the graph and legacy APIs, this section introduces the key concepts that are common to both.

2.1. cuDNN Handle

The cuDNN library exposes a host API but assumes that for operations using the GPU, the necessary data is directly accessible from the device.

An application using cuDNN must initialize a handle to the library context by calling `cudnnCreate()`. This handle is explicitly passed to every subsequent library function that operates on GPU data. Once the application finishes using cuDNN, it can release the resources associated with the library handle using `cudnnDestroy()`. This approach allows the user to explicitly control the library's functioning when using multiple host threads, GPUs, and CUDA streams.

For example, an application can use `cudaSetDevice` (prior to creating a cuDNN handle) to associate different devices with different host threads, and in each of those host threads, create a unique cuDNN handle that directs the subsequent library calls to the device associated with it. In this case, the cuDNN library calls made with different handles would automatically run on different devices.

The device associated with a particular cuDNN context is assumed to remain unchanged between the corresponding `cudnnCreate()` and `cudnnDestroy()` calls. In order for the cuDNN library to use a different device within the same host thread, the application must set the new device to be used by calling `cudaSetDevice()` and then create another cuDNN context, which will be associated with the new device, by calling `cudnnCreate()`.

2.2. Tensors and Layouts

Whether using the graph API or the legacy API, cuDNN operations take tensors as input and produce tensors as output.

2.2.1. Tensor Descriptor

The cuDNN library describes data with a generic n-D tensor descriptor defined with the following parameters:

- a number of dimensions from 3 to 8
- a data type (32-bit floating-point, 64 bit-floating point, 16-bit floating-point...)
- an integer array defining the size of each dimension
- an integer array defining the stride of each dimension (for example, the number of elements to add to reach the next element from the same dimension)

This tensor definition allows, for example, to have some dimensions overlapping each other within the same tensor by having the stride of one dimension smaller than the product of the dimension and the stride of the next dimension. In cuDNN, unless specified otherwise, all routines will support tensors with overlapping dimensions for forward-pass input tensors, however, dimensions of the output tensors cannot overlap. Even though this tensor format supports negative strides (which can be useful for data mirroring), cuDNN routines do not support tensors with negative strides unless specified otherwise.

2.2.1.1. WXYZ Tensor Descriptor

Tensor descriptor formats are identified using acronyms, with each letter referencing a corresponding dimension. In this document, the usage of this terminology implies:

- all the strides are strictly positive
- the dimensions referenced by the letters are sorted in decreasing order of their respective strides

2.2.1.2. 3-D Tensor Descriptor

A 3-D tensor is commonly used for matrix multiplications, with three letters: B, M, and N. B represents the batch size (for batch GEMM, set to 1 for single GEMM), M represents the number of rows, and N represents the number of columns. Refer to the `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` operation for more information.

2.2.1.3. 4-D Tensor Descriptor

A 4-D tensor descriptor is used to define the format for batches of 2D images with 4 letters: `N,C,H,W` for respectively the batch size, the number of feature maps, the height and the width. The letters are sorted in decreasing order of the strides. The commonly used 4-D tensor formats are:

- `NCHW`
- `NHWC`
- `CHWN`

2.2.1.4. 5-D Tensor Descriptor

A 5-D tensor descriptor is used to define the format of the batch of 3D images with 5 letters: `N,C,D,H,W` for respectively the batch size, the number of feature maps, the depth, the height, and the width. The letters are sorted in decreasing order of the strides. The commonly used 5-D tensor formats are called:

- NCDHW
- NDHWC
- CDHWN

2.2.1.5. Fully-Packed Tensors

A tensor is defined as `XYZ-fully-packed` if, and only if:

- the number of tensor dimensions is equal to the number of letters preceding the `fully-packed` suffix
- the stride of the i-th dimension is equal to the product of the (i+1)-th dimension by the (i+1)-th stride
- the stride of the last dimension is 1

2.2.1.6. Partially-Packed Tensors

The partially `XYZ-packed` terminology only applies in the context of a tensor format described with a superset of the letters used to define a partially-packed tensor. A `WXYZ` tensor is defined as `XYZ-packed` if, and only if:

- the strides of all dimensions NOT referenced in the `-packed` suffix are greater or equal to the product of the next dimension by the next stride
- the stride of each dimension referenced in the `-packed` suffix in position `i` is equal to the product of the (`i+1`)-st dimension by the (`i+1`)-st stride
- if the last tensor's dimension is present in the `-packed` suffix, its stride is `1`

For example, an `NHWC` tensor WC-packed means that the `c_stride` is equal to 1 and `w_stride` is equal to `c_dim x c_stride`. In practice, the `-packed` suffix is usually applied to the minor dimensions of a tensor but can be applied to only the major dimensions; for example, an `NCHW` tensor that is only N-packed.

2.2.1.7. Spatially Packed Tensors

Spatially-packed tensors are defined as partially-packed in spatial dimensions. For example, a spatially-packed 4D tensor would mean that the tensor is either NCHW HW-packed or CNHW HW-packed.

2.2.1.8. Overlapping Tensors

A tensor is defined to be overlapping if iterating over a full range of dimensions produces the same address more than once. In practice an overlapped tensor will have `stride[i-1] < stride[i]*dim[i]` for some of the `i` from `[1,nbDims]` interval.

2.2.2. Data Layout Formats

This section describes how cuDNN tensors are arranged in memory according to several data layout formats.

The recommended way to specify the layout format of a tensor is by setting its strides accordingly. For compatibility with the v7 API, a subset of the layout formats can also be configured through the `cudnnTensorFormat_t` enum. The enum is only supplied for legacy reasons and is deprecated.

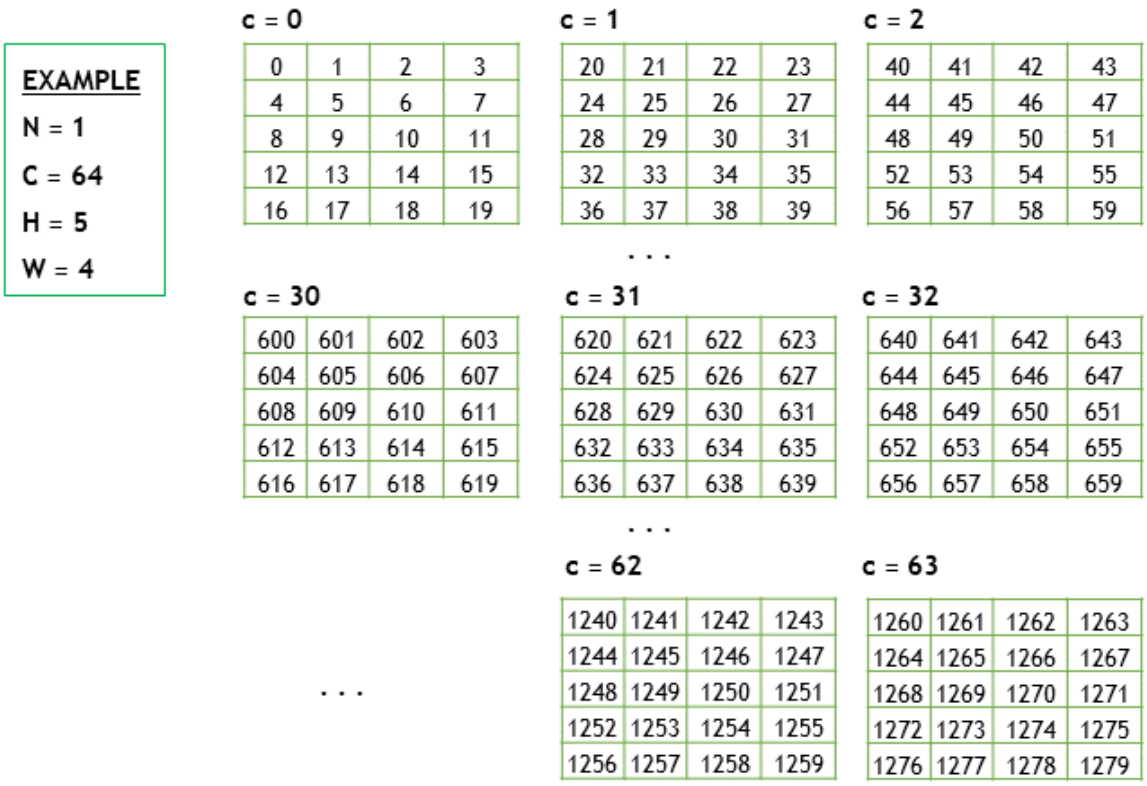
2.2.2.1. Example Tensor

Consider a batch of images with the following dimensions:

- N is the batch size; 1
- C is the number of feature maps (that is,, number of channels); 64
- H is the image height; 5
- W is the image width; 4

To keep the example simple, the image pixel elements are expressed as a sequence of integers, 0, 1, 2, 3, and so on. Refer to Figure 1.

Figure 1. Example with N=1, C=64, H=5, W=4



In the following subsections, we'll use the above example to demonstrate the different layout formats.

2.2.2.2. Convolution Layouts

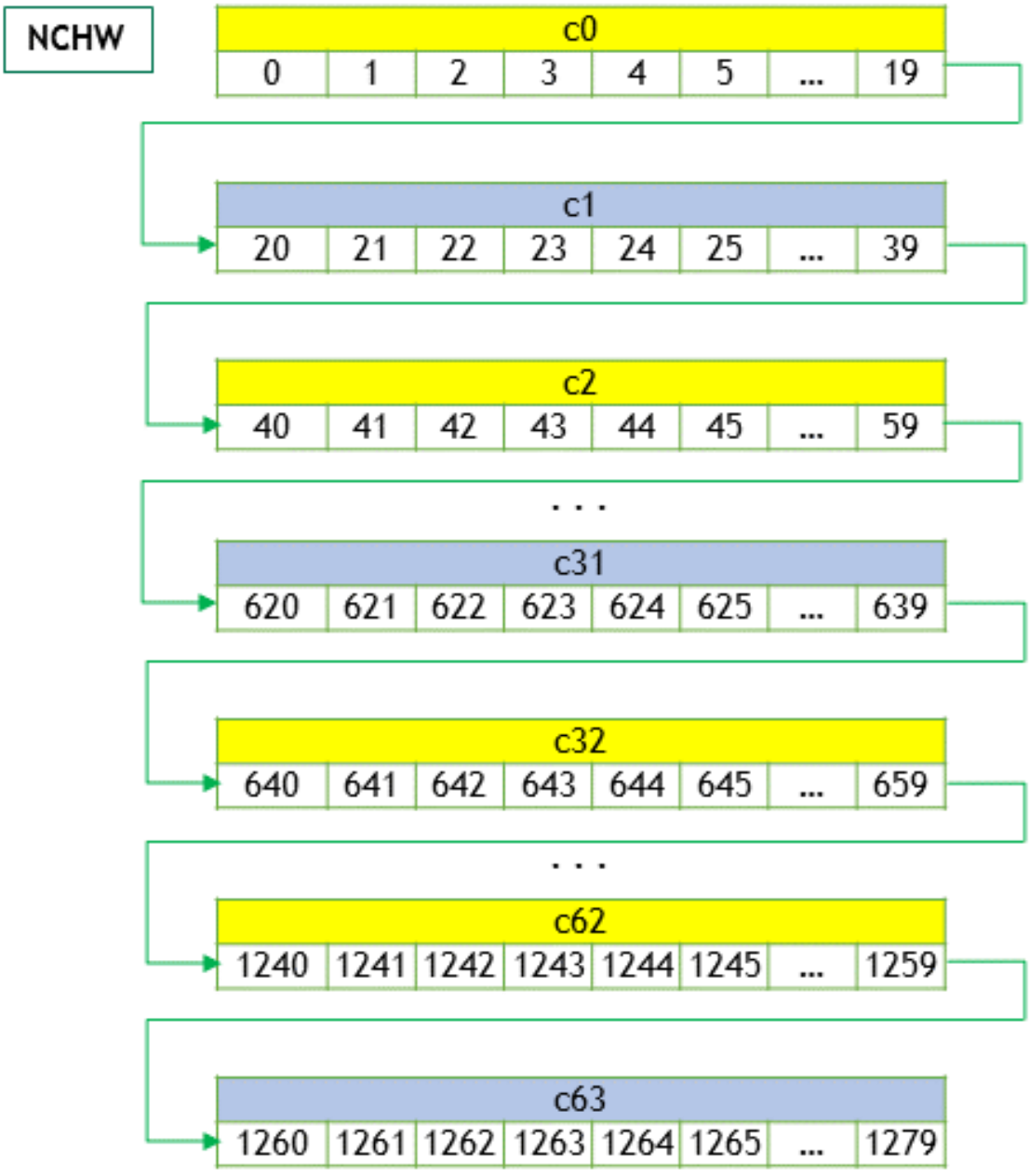
cuDNN supports several layouts for convolution, as described in the following sections.

2.2.2.2.1. NCHW Memory Layout

The above 4D tensor is laid out in the memory in the NCHW format as below:

1. Beginning with the first channel (c=0), the elements are arranged contiguously in row-major order.
2. Continue with second and subsequent channels until the elements of all the channels are laid out. Refer to Figure 2.
3. Proceed to the next batch (if N is > 1).

Figure 2. NCHW Memory Layout



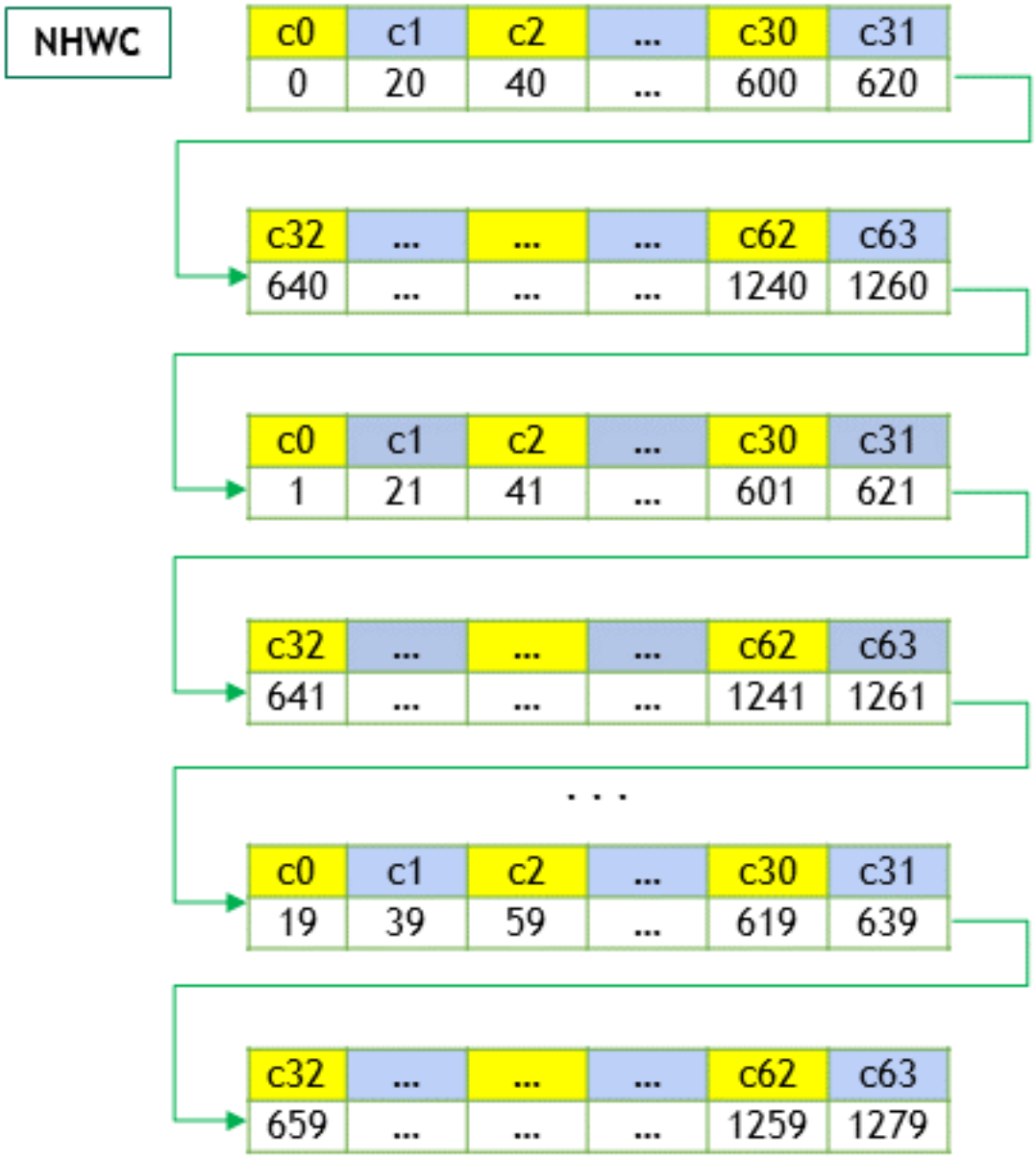
2.2.2.2.2. NHWC Memory Layout

For the NHWC memory layout, the corresponding elements in all the C channels are laid out first, as below:

1. Begin with the first element of channel 0, then proceed to the first element of channel 1, and so on, until the first elements of all the C channels are laid out.

2. Next, select the second element of channel 0, then proceed to the second element of channel 1, and so on, until the second element of all the channels are laid out.
3. Follow the row-major order of channel 0 and complete all the elements. Refer to Figure 3.
4. Proceed to the next batch (if N is > 1).

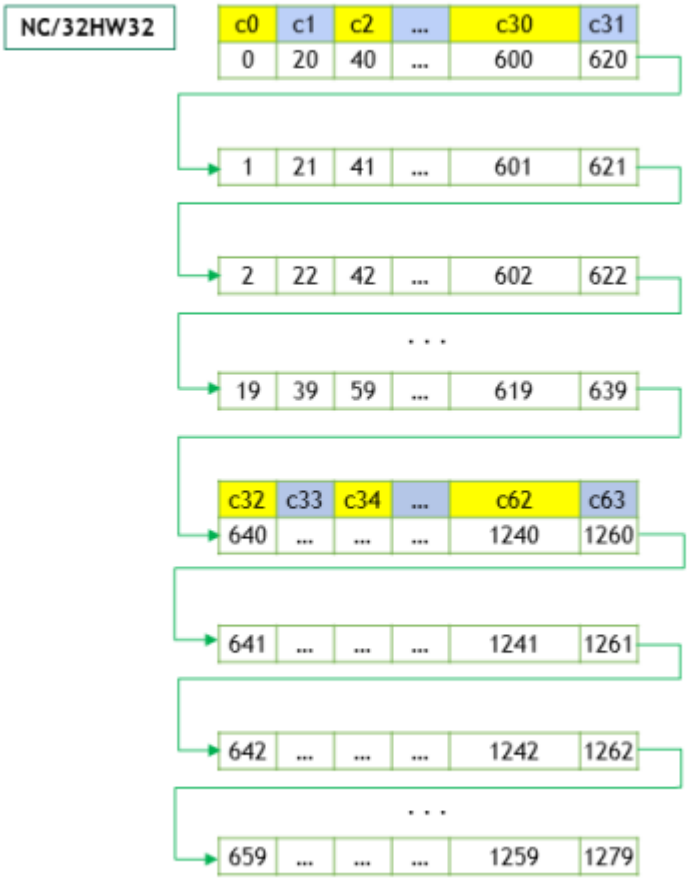
Figure 3. NHWC Memory Layout



2.2.2.2.3. NC/32HW32 Memory Layout

The NC/32HW32 is similar to NHWC, with a key difference. For the NC/32HW32 memory layout, the 64 channels are grouped into two groups of 32 channels each - first group consisting of channels `c0` through `c31`, and the second group consisting of channels `c32` through `c63`. Then each group is laid out using the NHWC format. Refer to Figure 4.

Figure 4. NC/32HW32 Memory Layout



For the generalized NC/xHWx layout format, the following observations apply:

- Only the channel dimension, `C`, is grouped into `x` channels each.
- When `x = 1`, each group has only one channel. Hence, the elements of one channel (that is, one group) are arranged contiguously (in the row-major order), before proceeding to the next group (that is, next channel). This is the same as NCHW format.

- When `x = C`, then NC/xHWx is identical to NHWC, that is, the entire channel depth `C` is considered as a single group. The case `x = C` can be thought of as vectorizing the entire `C` dimension as one big vector, laying out all the `C`s, followed by the remaining dimensions, just like NHWC.
- The tensor format `cudnnTensorFormat_t` can also be interpreted in the following way: The NCHW INT8x32 format is really `N x (C/32) x H x W x 32` (32 `C`s for every `W`), just as the NCHW INT8x4 format is `N x (C/4) x H x W x 4` (4 `C`s for every `W`). Hence the `VECT_C` name - each `W` is a vector (4 or 32) of `C`s.

2.2.2.3. MatMul Layouts

As discussed in 3-D Tensor Descriptor, matmul uses 3D tensors, described using BMN dimensions. The layout can be specified through the following strides. The following are two examples of recommended layouts:

- Packed Row-major: dim [B,M,N] with stride [MN, N, 1], or
- Packed Column-major: dim [B,M,N] with stride [MN, 1, M]

Unpacked layouts for 3-D tensors are supported as well, but their support surface is more ragged.

2.3. Tensor Core Operations

The cuDNN v7 library introduced the acceleration of compute-intensive routines using Tensor Core hardware on supported GPU SM versions. Tensor Core operations are supported beginning with the NVIDIA Volta GPU.

Tensor Core operations accelerate matrix math operations; cuDNN uses Tensor Core operations that accumulate into FP16, FP32, and INT32 values. Setting the math mode to `CUDNN_TENSOR_OP_MATH` via the `cudnnMathType_t` enumerator indicates that the library will use Tensor Core operations. This enumerator specifies the available options to enable the Tensor Core and should be applied on a per-routine basis.

The default math mode is `CUDNN_DEFAULT_MATH`, which indicates that the Tensor Core operations will be avoided by the library. Because the `CUDNN_TENSOR_OP_MATH` mode uses the Tensor Cores, it is possible that these two modes generate slightly different numerical results due to different sequencing of the floating-point operations.

For example, the result of multiplying two matrices using Tensor Core operations is very close, but not always identical, to the result achieved using a sequence of scalar floating-point operations. For this reason, the cuDNN library requires an explicit user opt-in before enabling the use of Tensor Core operations.

However, experiments with training common deep learning models show negligible differences between using Tensor Core operations and scalar floating point paths, as measured by both the final network accuracy and the iteration count to convergence. Consequently, the cuDNN library treats both modes of operation as functionally indistinguishable and allows for the scalar paths to serve as legitimate fallbacks for cases in which the use of Tensor Core operations is unsuitable.

Kernels using Tensor Core operations are available for:

- Convolutions
- RNNs
- Multi-Head Attention

For more information, refer to NVIDIA Training with Mixed Precision.

For a deep learning compiler, the following are the key guidelines:

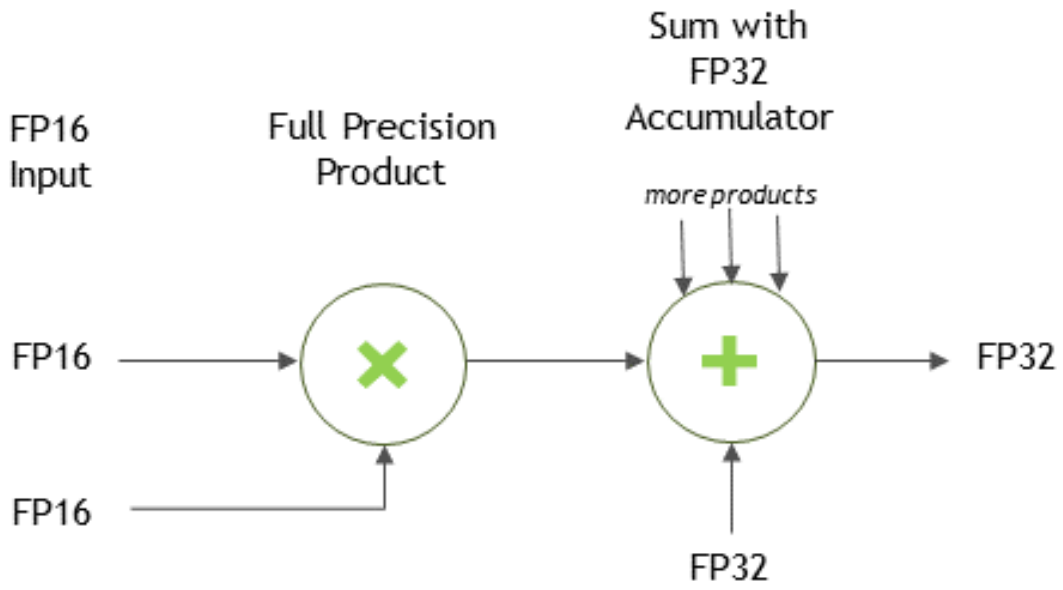
- Make sure that the convolution operation is eligible for Tensor Cores by avoiding any combinations of large padding and large filters.
- Transform the inputs and filters to NHWC, pre-pad channel and batch size to be a multiple of 8.
- Make sure that all user-provided tensors, workspace, and reserve space are aligned to 128-bit boundaries. Note that 1024-bit alignment may deliver better performance.

2.3.1. Notes on Tensor Core Precision

For FP16 data, Tensor Cores operate on FP16 input, output in FP16, and may accumulate in FP16 or FP32. The FP16 multiply leads to a full-precision result that is accumulated in FP32 operations with the other products in a given dot product for a matrix with `m x n x k` dimensions. Refer to Figure 5.

For an FP32 accumulation, with FP16 output, the output of the accumulator is down-converted to FP16. Generally, the accumulation type is of greater or equal precision to the output type.

Figure 5. Tensor operation with FP16 inputs. The accumulation is in FP32, which could be the input for other kernel features (for example, activation/bias, beta blending, etc). The final output in this example would be FP16.



Chapter 3. Graph API

The cuDNN library provides a declarative programming model for describing computation as a graph of operations. This *graph API* was introduced in cuDNN 8.0 to provide a more flexible API, especially with the growing importance of operation fusion.

At a high level, the user is describing a dataflow graph of operations on tensors. Given a *finalized* graph, the user then selects and configures an engine that can execute that graph. There are several methods for selecting and configuring engines, which have tradeoffs with respect to ease-of-use, runtime overhead, and engine performance.

The graph API has two entry points:

- NVIDIA cuDNN Backend API (lowest level entry point into the graph API)
- NVIDIA cuDNN Frontend API (convenience layer on top of the C backend API)

We expect that most users prefer the cuDNN frontend API because:

- It is less verbose without loss of control - all functionality accessible through the backend API is also accessible through the frontend API.
- It adds functionality on top of the backend API, like errata filters and autotuning.
- It is open source.

In either case (that is, the backend or frontend API), the high level concepts are the same.

3.1. Key Concepts

As mentioned previously, the key concepts in the graph API are:

- Operations and Operation Graphs
- Engines and Engine Configurations
- Heuristics

3.1.1. Operations and Operation Graphs

An operation graph is a dataflow graph of operations on tensors. It is meant to be a mathematical specification and is decoupled from the underlying *engines* that can implement it, as there may be more than one engine available for a given graph.

I/O tensors connect the operations implicitly, for example, an operation A may produce a tensor X, which is then consumed by operation B, implying that operation B depends on operation A.

3.1.2. Engines and Engine Configurations

For a given operation graph, there are some number of engines that are candidates for implementing that graph. The typical way to query for a list of candidate engines is through a heuristics query, covered below.

An engine has knobs for configuring properties of the engine, like tile size (refer to `cudnnBackendKnobType_t`).

3.1.3. Heuristics

A *heuristic* is a way to get a list of engine configurations that are intended to be sorted from the most performant to least performant for the given operation graph. There are three modes:

`CUDNN_HEUR_MODE_A`
: Intended to be fast and be able to handle most operation graph patterns. It returns a list of engine configs ranked by the expected performance.

`CUDNN_HEUR_MODE_B`
: Intended to be more generally accurate than mode A, but with the tradeoff of higher CPU latency to return the list of engine configs. The underlying implementation may fall back to the mode A heuristic in cases where we know mode A can do better.

`CUDNN_HEUR_MODE_FALLBACK`
: Intended to be fast and provide functional fallbacks without expectation of optimal performance.

The recommended workflow is to query either mode A or B and check for support. The first engine config with support is expected to have the best performance.

You can “auto-tune”, that is, iterate over the list and time for each engine config and choose the best one for a particular problem on a particular device. The cuDNN frontend API provides a convenient function, `cudnnFindPlan()`, which does this.

If all the engine configs are not supported, then use the mode fallback to find the functional fallbacks.

Expert users may also want to filter engine configs based on properties of the engine, such as numerical notes, behavior notes, or adjustable knobs. Numerical notes inform the user about the numerical properties of the engine such as whether it does datatype down conversion at the input or during output reduction. The behavior notes can signal something about the underlying implementation like whether or not it uses runtime compilation. The adjustable knobs allow fine grained control of the engine’s behavior and performance.

3.2. Graph API Example with Operation Fusion

The following example implements a fusion of convolution, bias, and activation.

3.2.1. Creating Operation and Tensor Descriptors to Specify the Graph Dataflow

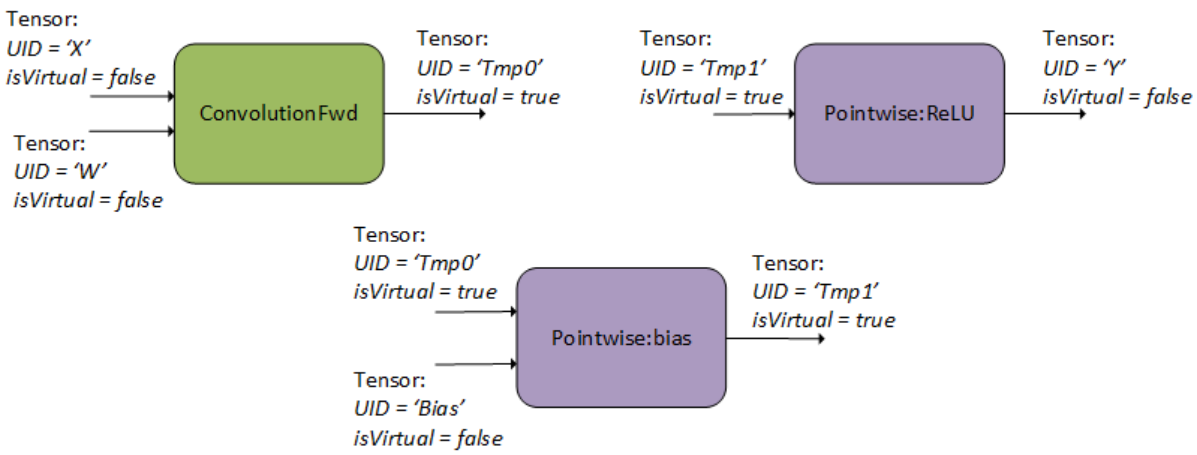
First, create three cuDNN backend operation descriptors.

As can be seen in Figure 6, the user specified one forward convolution operation (using `CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR`), a pointwise operation for the bias addition (using `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_ADD`), and a pointwise operation for the ReLU activation (using `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_RELU_FWD`). Refer to the NVIDIA cuDNN Backend API for more details on setting the attributes of these descriptors. For an example of how a forward convolution can be set up, refer to the Setting Up An Operation Graph For A Grouped Convolution use case in the cuDNN backend API.

You should also create tensor descriptors for the inputs and outputs of all of the operations in the graph. The graph dataflow is implied by the assignment of tensors (refer to Figure 6), for example, by specifying the backend tensor *Tmp0* as both the output of the convolution operation and the input of the bias operation, cuDNN infers that the dataflow runs from the convolution into the bias. The same applies to tensor *Tmp1*. If the user doesn't need the intermediate results *Tmp0* and *Tmp1* for any other use, then the user can specify them to be virtual tensors, so the memory I/Os can later be optimized out.

- Note that graphs with more than one operation node do not support in-place operations (that is, where any of the input UIDs matches any of the output UIDs). Such in-place operations are considered cyclic in later graph analysis and deemed unsupported. In-place operations are supported for single-node graphs.
- Also note that the operation descriptors can be created and passed into cuDNN in any order, as the tensor UIDs are enough to determine the dependencies in the graph.

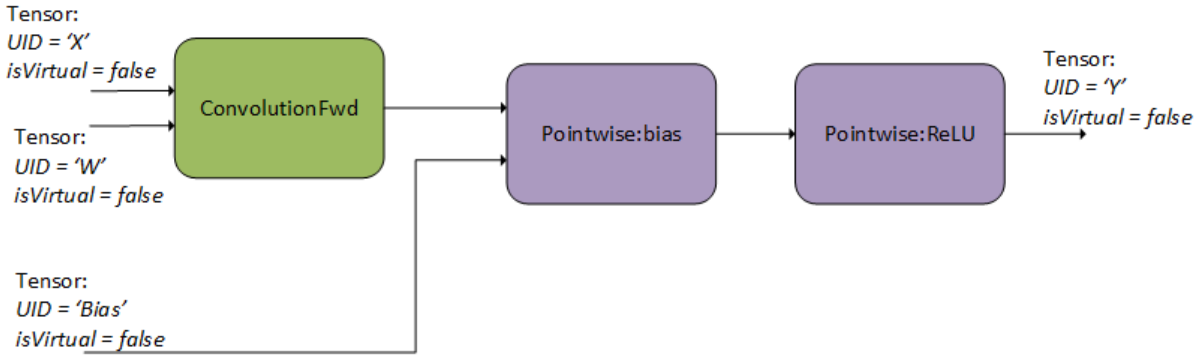
Figure 6. A set of operation descriptors the user passes to the operation graph



3.2.2. Finalizing The Operation Graph

Second, the user finalizes the operation graph. As part of finalization, cuDNN performs the dataflow analysis to establish the dependency relationship between operations and connect the edges, as illustrated in the following figure. In this step, cuDNN performs various checks to confirm the validity of the graph.

Figure 7. The operation graph after finalization



3.2.3. Configuring An Engine That Can Execute The Operation Graph

Third, given the finalized operation graph, the user must select and configure an engine to execute that graph, which results in an execution plan. As mentioned in Heuristics, the typical way to do this is:

1. Query heuristics mode A or B.

2. Look for the first engine config with functional support (or auto-tune all the engine configs with functional support).
3. If no engine config was found in (2), try querying the fallback heuristic for more options.

3.2.4. Executing The Engine

Finally, with the execution plan constructed and when it comes time to run it, the user should construct the backend variant pack by providing the workspace pointer, an array of UIDs, and an array of device pointers. The UIDs and the pointers should be in the corresponding order. With the handle, the execution plan and variant pack, the execution API can be called and the computation is carried out on the GPU.

3.3. Supported Graph Patterns

The cuDNN Graph API supports a set of graph patterns. These patterns are supported by a large number of engines, each with their own support surfaces. These engines are grouped into three different classes, as reflected by the following three subsections: pre-compiled single operation engines, runtime fusion engines, and specialized pre-compiled engines.

Since these engines have some overlap in the patterns they support, a given pattern may result in zero, one, or more engines.

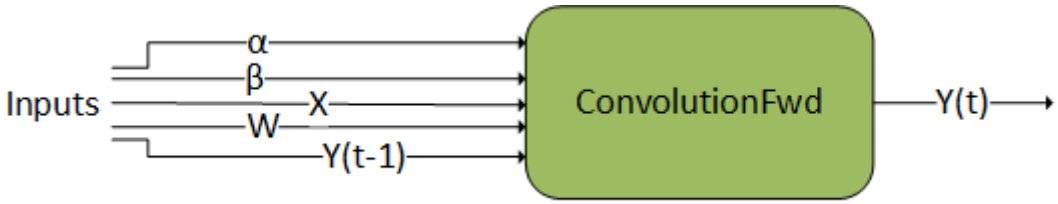
3.3.1. Pre-compiled Single Operation Engines

One basic class of engines includes pre-compiled engines that support an operation graph with just one operation; specifically: `ConvolutionFwd`, `ConvolutionBwFilter`, `ConvolutionBwData`, or `ConvolutionBwBias`. Their more precise support surface can be found in the NVIDIA cuDNN API Reference.

3.3.1.1. `ConvolutionFwd`

`ConvolutionFwd` computes the convolution of X with filter data W. In addition, it uses scaling factors α and β to blend this result with the previous output. This graph operation is similar to `cudnnConvolutionForward()`.

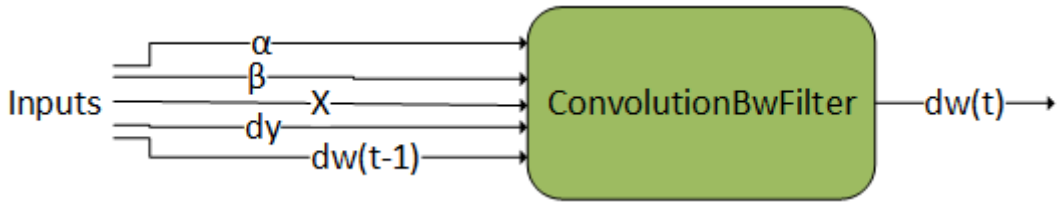
Figure 8. `ConvolutionFwd` Engine



3.3.1.2. `ConvolutionBwFilter`

`ConvolutionBwFilter` computes the convolution filter gradient of the tensor dy. In addition, it uses scaling factors α and β to blend this result with the previous output. This graph operation is similar to `cudnnConvolutionBackwardFilter()`.

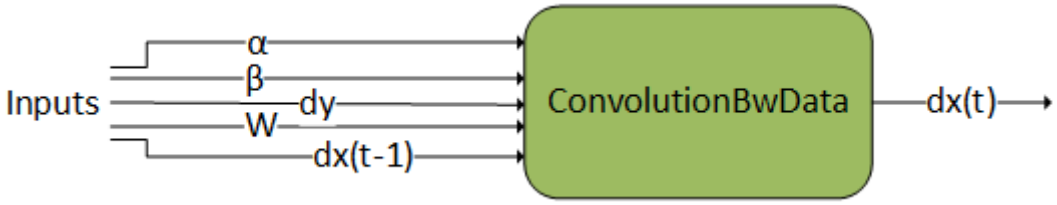
Figure 9. `ConvolutionBwFilter` Engine



3.3.1.3. `ConvolutionBwData`

`ConvolutionBwData` computes the convolution data gradient of the tensor dy. In addition, it uses scaling factors α and β to blend this result with the previous output. This graph operation is similar to `cudnnConvolutionBackwardData()`.

Figure 10. `ConvolutionBwData` Engine



3.3.2. Runtime Fusion Engine

The engines documented in the previous section support single-op patterns. Of course, for fusion to be interesting, the graph needs to support multiple operations. And ideally, we want the supported patterns to be flexible to cover a diverse set of use cases. To accomplish this generality, cuDNN has a runtime fusion engine that generates the kernel (or kernels) at runtime based on the graph pattern. This section outlines the patterns supported by these runtime fusion engines (that is, engines with `CUDNN_BEHAVIOR_NOTE_RUNTIME_COMPILATION` behavioral note).

We can think of the support surface as covering the follwing generic patterns:

1. `ConvolutionFwd` fusions

   $$g_2\left(\mathrm{Y} = \mathrm{convolutionFwd}\left(\mathrm{X} = g_1\left(\mathrm{inputs}\right), \mathrm{W}\right), \mathrm{inputs}\right)$$

2. `ConvolutionBwFilter` fusions

$g_2(\text{dw} = \text{convolutionBwFilter}(\text{dy} = g_1(\text{inputs}), \text{X}), \text{inputs})$

3. `ConvolutionBwData` fusions

   $g_2(\text{dx} = \text{convolutionBwData}(\text{dy}, \text{W}), \text{inputs})$

4. `MatMul` fusions

   $g_2(\text{C} = \text{matmul}(\text{A} = g_1(\text{inputs}), \text{B}), \text{inputs})$

5. Pointwise fusions

   $g_2(\text{inputs})$

6. `Mha-Fprop` fusions

   $\text{O} = \text{matmul}(\text{S} = g_4(\text{P} = \text{matmul}(\text{Q}, g_2(\text{K})), \text{V}))$

7. `Mha-Bprop` fusions

   This pattern is executed in a fused pattern in a single kernel.

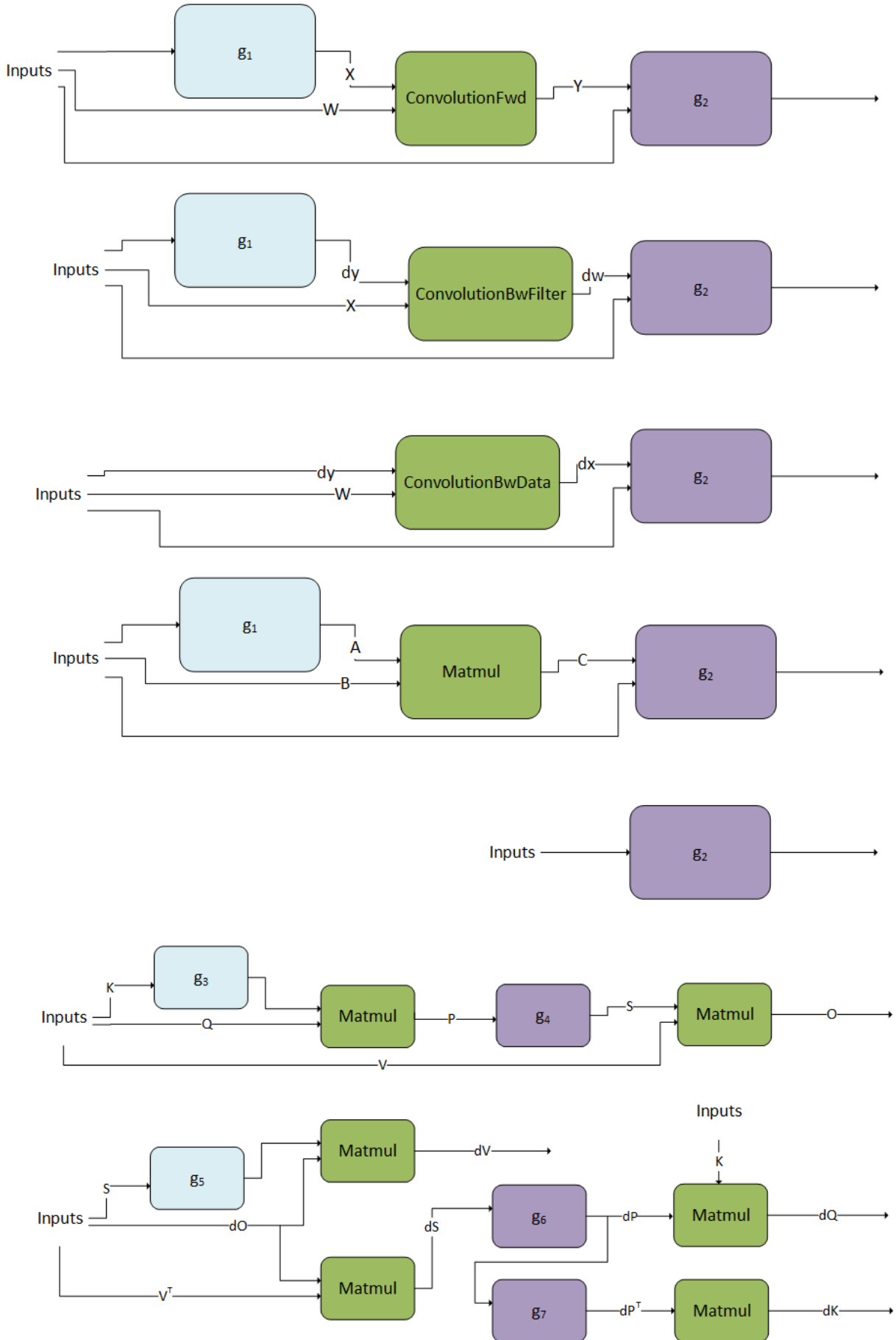
   $\text{dV} = \text{matmul}(g_5(\text{S}), \text{dO})$

   $\text{dS} = \text{matmul}(\text{dO}, V^T)$

   $\text{dQ} = \text{matmul}(g_6(\text{dS}), \text{K})$

   $\text{dK} = \text{matmul}(\text{Q}, g_7(\text{dS}))$

Figure 11. Graphical Representation of the Generic Patterns Supported by the Runtime Fusion Engine



g_1 is a directed acyclic graph (DAG) that can consist of zero or any number of the following operation:

- `CUDNN_BACKEND_OPERATION_CONCAT_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_SIGNAL_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR`

g_2 is a DAG that can consist of zero or any number of the following operations:

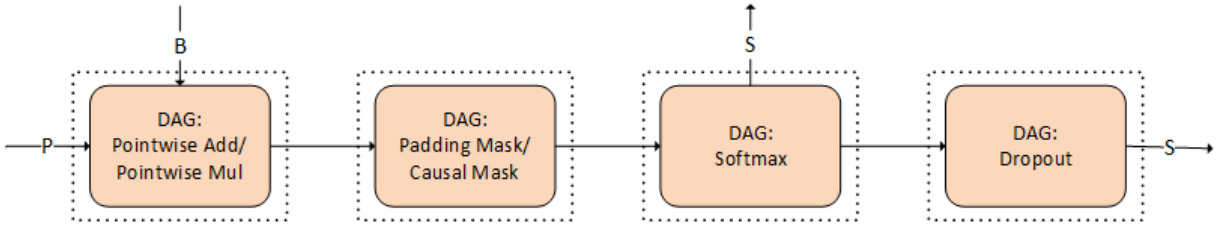
- `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR`
- `CUDNN_BACKEND_OPERATION_SIGNAL_DESCRIPTOR`

`MatMul-MatMul` has been added to the runtime fusion engine to serve patterns that are commonly used in multihead attention. g_3 and g_4 are limited to a certain few DAGs of operations.

g_3 can be a empty graph or a single scale operation with the scale being a scalar value (`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR with mode CUDNN_POINTWISE_MUL`).

g_4 can be empty or the combination of the following DAGs of cuDNN operations. Each of these DAGs is optional, as shown by the dotted line.

Figure 12. DAGs of cuDNN operations

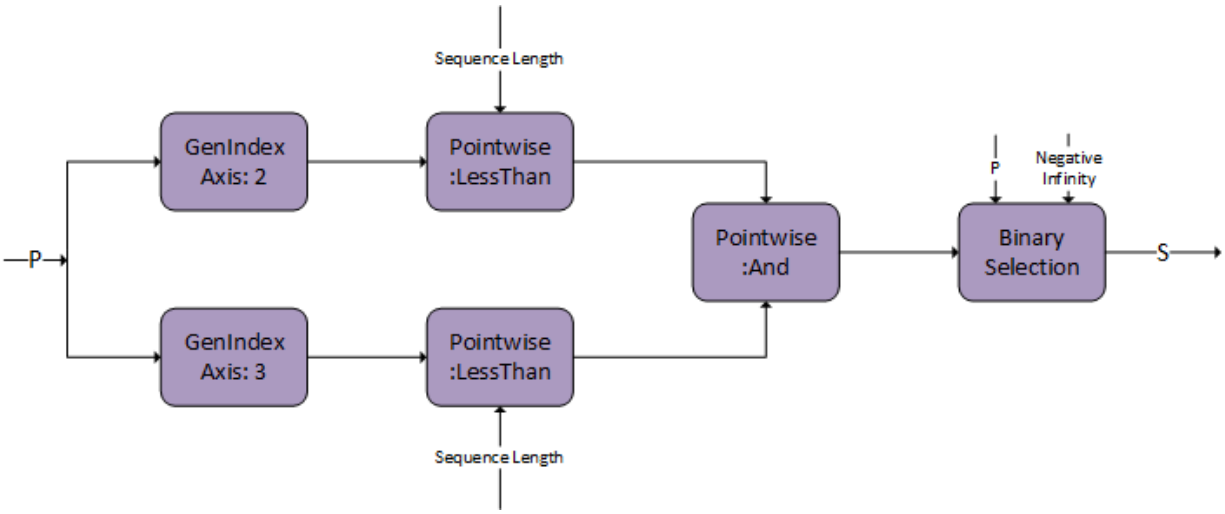


The combination has to obey the order in which we present them. For example: if you want to use the padding mask and softmax, the padding mask has to appear before softmax.

These operations are commonly used in multihead attention. In the following diagram, we depict how to create a DAG for each of the operations. In later versions, we will be expanding the possible DAGs for g_3 and g_4.

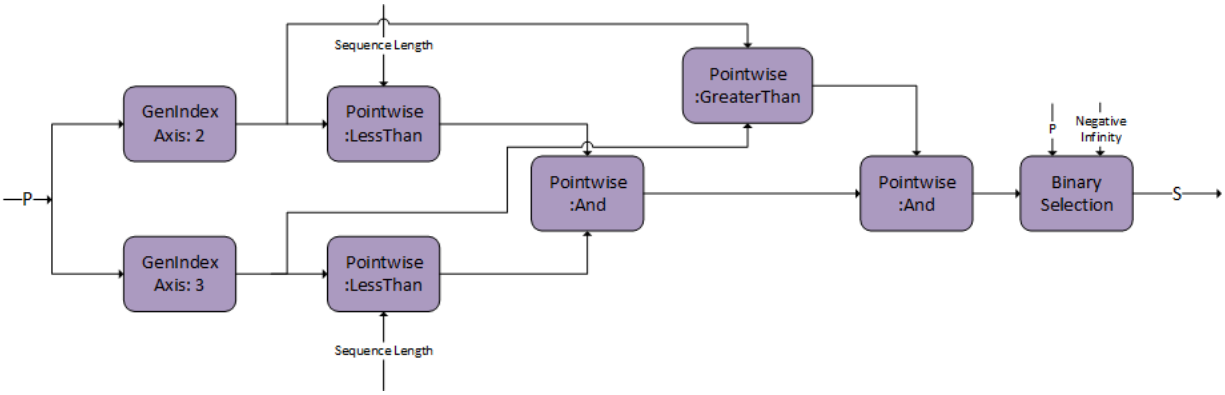
Padding Mask

Figure 13. cuDNN graph depicting DAG:Padding Mask



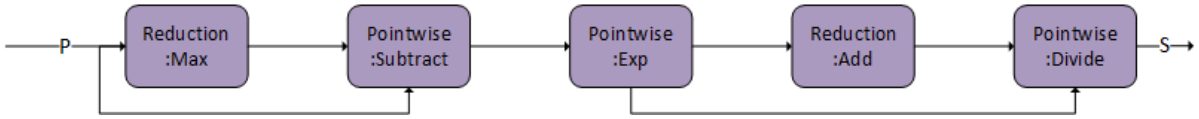
Causal Mask

Figure 14. cuDNN graph depicting DAG:Causal Mask



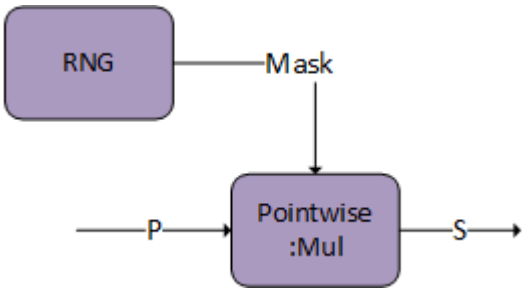
Softmax

Figure 15. cuDNN graph depicting DAG:Softmax



Dropout

Figure 16. cuDNN graph depicting DAG:Dropout

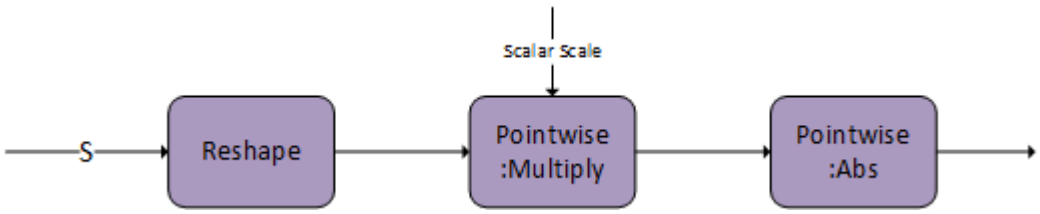


g_4 is capable of storing an intermediate tensor to global memory marked as S, which can be used for pattern 7. Both DAG:Softmax and DAG:Dropout have this capability. You should set S as the output from the last DAG in the graph.

The tensor descriptor marked as S needs to have the `CUDNN_ATTR_TENSOR_REORDERING_MODE` set to `CUDNN_TENSOR_REORDERING_F16x16`. This is because the tensor is stored in a special format and can only be consumed by pattern 7.

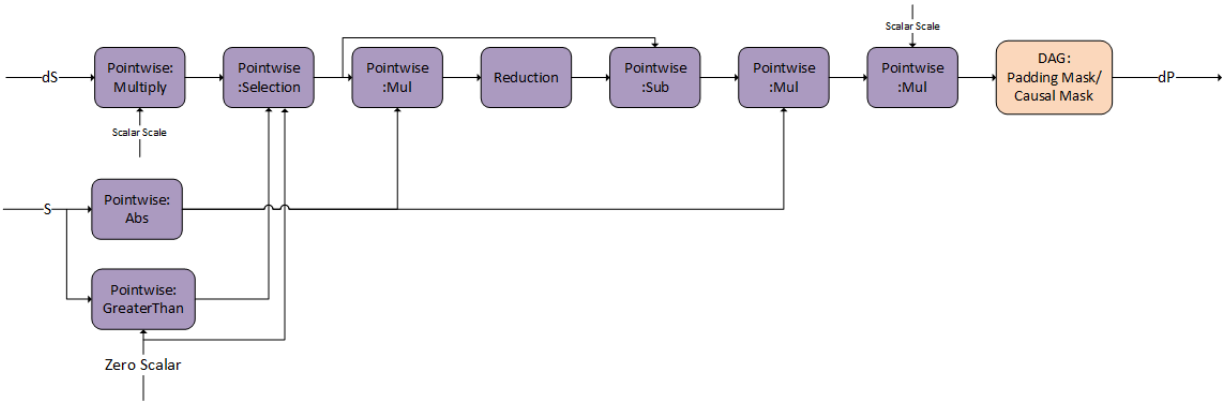
g_5, g_6, and g_7 can only support a fixed DAG. We are working towards generalizing these graphs.

Figure 17. cuDNN graph depicting g_5



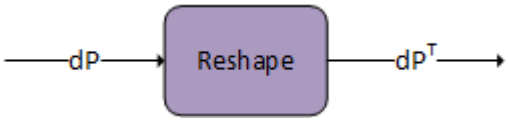
g6 represents the backward pass of Softmax and masking, to get dP.

Figure 18. cuDNN graph depicting g_6



g7 is the transpose of dP the output of g6.

Figure 19. cuDNN graph depicting g_7



Note:

- The arrow going into g_2 can go into any of g_2's nodes and does not necessarily need to feed into a root node.
- The abbreviated notations for operations are used in the diagrams and throughout the text for visualization purposes.

3.3.2.1. Limitations

While the generic patterns listed previously are widely applicable, there are some cases where we do not have full support.

Limitations Common to all Generic Patterns

Limitations to g_1:

- Concatenation or signaling operations, if present, should be before any pointwise operations.
- For compute capability < 8.0, g_1 is not supported.

Limitations to g_2:

- As specified in the previous section, g_2 can include only `Pointwise` operations, `ResampleFwd`, `ResampleBwd`, `GenStats`, and `Reduction`.

- The I/O (that is, non-virtual) tensor data type can be any of `{FP32, FP16, BF16, INT8, packed-BOOLEAN}`.
- For pointwise operations, non-virtual tensors need to be either all NCHW (or row-major), or all NHWC (or column-major).
- The intermediate virtual tensor data type can be any of `{FP32, FP16, BF16, INT8, BOOLEAN}`, and this intermediate storage type is obeyed by the code-generator. Generally, FP32 is recommended.
- The input tensor to a `ResampleFwd` or `ResampleBwd` operation should not be produced by another operation within this graph, but should come from global memory. The two operations cannot be used in the `ConvolutionBwFilter`, `ConvolutionBwData`, and `MatMul` fusion patterns.
- There can be at most one reduction operation, and it needs to be at the final node of g_2.
- Signaling operations, if present, must be the final nodes in g_2. Hence, signaling operations cannot be used in conjunction with reduction operations.
- For `ResampleFwd` or `ResampleBwd` operations, when the tensor format is NCHW/NCDHW, there are some limitations.
  - Upsampling is not supported
  - `Int64_t` indices is not supported
  - Only support symmetric padding using the prepadding backend API
  - X, Y, and DY are required when max pooling.

Limitations per Generic Pattern

Table 1. Limitations to g_1

	Limitations to g_1
`ConvolutionFwd` fusions	- Fusion operations on input tensors can be only a chain of three specific pointwise operations, in this exact order: `Pointwise:mul`, `Pointwise:add`, and `Pointwise:ReLU`. This specific support is added to realize convolution batch norm fusion use cases. - All tensors involved can only be FP16. - `Pointwise:mul` can only be with a tensor of scalars per channel. - `Pointwise:add` can only be a column broadcast.

	Limitations to g_1
`ConvolutionBwFilter` fusions	Same limitations specified for `ConvolutionFwd` fusions apply here.
`ConvolutionBwData` fusions	No fusion on input tensors for backward data convolution is supported.
`MatMul` fusions	▶ Can be any combination of pointwise operations. ▶ Only fusible with operand A, not with B. ▶ Operand A should have an FP16 data type. ▶ Broadcasted input can have any data type. ▶ Compute type is FP32 only.
Pointwise fusions	Not Applicable

Table 2. Limitations to Mha-Fprop fusions

	Limitations to Mha-Fprop fusions
`MatMul`	▶ Compute type for both MatMul ops needs to be float ▶ Input tensors need to have datatype FP16 or BF16 ▶ Output tensors need to have datatype FP16, BF16, or FP32 (TF32) datatype
Pointwise operations in g_3 and g_4	Compute type needs to be FP32 (TF32)
Reduction operations in g_3 and g_4	I/O types and compute type needs to be FP32 (TF32)
`RNG` operation in g_3 and g_4	▶ Data type of `yTensor` needs to be FP32 (TF32) ▶ The `CUDNN_TYPE_RNG_DISTRIBUTION` needs to be `CUDNN_RNG_DISTRIBUTION_BERNOULLI`

Table 3. Limitations to Mha-Bprop fusions

	Limitations to Mha-Bprop fusions
`MatMul`	▶ Compute type for all MatMul ops needs to be float

	Limitations to Mha-Bprop fusions
	▶ Input tensors need to have datatype FP16 or BF16 ▶ Output tensors need to have datatype FP16, BF16, or FP32 (TF32) datatype
Pointwise operations in g_5 , g_6 and g_7	Compute type needs to be FP32 (TF32)
Reduction operations in g_5 , g_6 and g_7	I/O types and compute type needs to be FP32 (TF32)

Tensor Layout Requirements

Lastly, there are some layout requirements to the I/O tensors involved in fusion graphs. For more information, refer to the Tensor Descriptor and Data Layout Formats sections. The following table describes the requirements per fusion pattern:

Table 4. Layout Requirements per Pattern

Pattern	Layout Requirement
`ConvolutionFwd`, `ConvolutionBwFilter`, `ConvolutionBwData` fusions	▶ All tensors are fully packed NHWC.
`MatMul` fusions	▶ Input operands can have either row-major or all column-major. ▶ In g_1, the tensor operating with Matrix A (dim[B, M, K]) can be either a scalar with dim[1, 1, 1], a row vector with dim[B, M, 1], a column vector with dim[B, 1, K], or a full matrix with dim[B, M, K]. ▶ In g_2, all I/O tensors should be either all row-major or all column-major.
Pointwise fusions	▶ If all tensors are 3D, the same layout requirements as matmul g_2. ▶ If all tensors are 4D or 5D, the same requirements as `ConvolutionFwd`, `ConvolutionBwFilter`, `ConvolutionBwData` layout.
`Mha-Fprop` fusions	▶ All I/O tensors need to have 4 dimensions, with the first two denoting the batch dimensions. The usage of rank-4 tensors in

Pattern	Layout Requirement
	matmul ops can be read from the NVIDIA cuDNN Backend API documentation. ▶ The contracting dimension (dimension K) for the first matmul needs to be 64. ▶ The non contracting dimension (dimensions M and N) for the first matmul needs to be equal and less than or equal to 512. ▶ The last dimension (corresponding to hidden dimensions) in Q, V and O is expected to have stride 1. ▶ For the K tensor, the stride is expected to be 1 for the 2nd last dimension. ▶ The S tensor is expected to have the `CUDNN_ATTR_TENSOR_REORDERING_MODE` set to `CUDNN_TENSOR_REORDERING_F16x16`
`Mha-Bprop` fusions	▶ All I/O tensors need to have 4 dimensions, with the first two denoting the batch dimensions. The usage of rank-4 tensors in matmul ops can be read from the NVIDIA cuDNN Backend API documentation. ▶ The contracting dimension (dimension K) for the second matmul needs to be 64. ▶ The contracting dimension (dimension K) for the 1st, 2nd and 3rd matmul needs to be equal and less than or equal to 512. ▶ The last dimension (corresponding to hidden dimensions) in Q, K, V, O and dO is expected to have stride 1. ▶ The S tensor and dP tensor is expected to have the `CUDNN_ATTR_TENSOR_REORDERING_MODE` set to `CUDNN_TENSOR_REORDERING_F16x16`

3.3.2.2. Examples of Supported Patterns

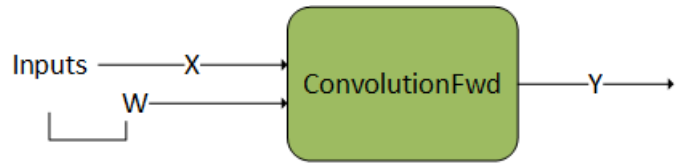
The following sections provide examples of supported patterns, in order of increasing complexity. We employ the same color scheme as in the overall pattern to aid in identifying the structure of g_1 (blue) and g_2 (purple).

For illustration purposes, we abbreviated the operations used. For a full mapping to the actual backend descriptors, refer to the Mapping with Backend Descriptors.

3.3.2.2.1. Single Operation

The following example illustrates a convolution operation without any operations before or after it. This means, g_1 and g_2, are empty graphs.

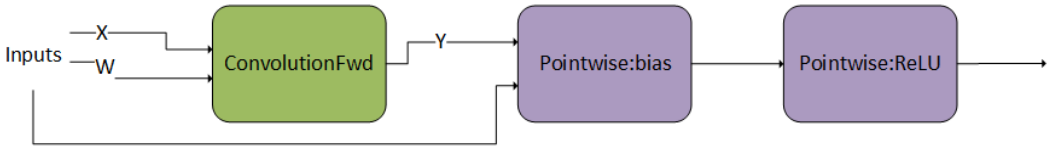
Figure 20. This example illustrates the Runtime Fusion Engine with a Single Operation



3.3.2.2.2. Pointwise Operations After Convolution 1

In this example, g_2 consists of a sequential set of two pointwise operations after the convolution.

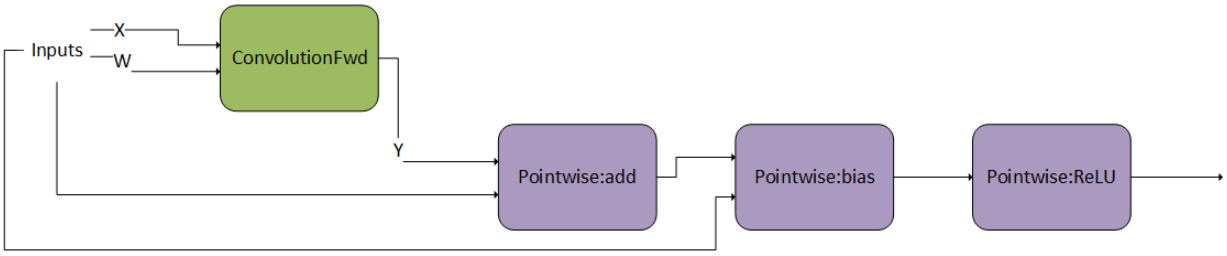
Figure 21. `ConvolutionFwd` Followed by a DAG with Two Operations



3.3.2.2.3. Pointwise Operations After Convolution 2

Similar to the previous example, g_2 consists of a sequential set of multiple pointwise operations.

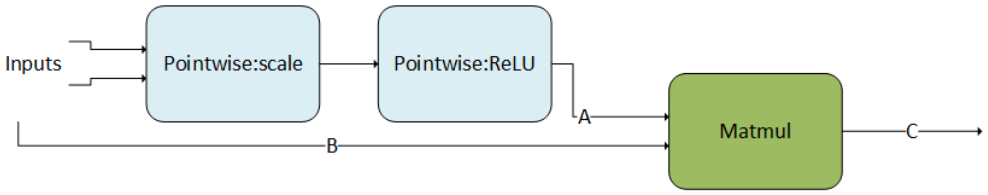
Figure 22. `ConvolutionFwd` Followed by a DAG with Three Operations



3.3.2.2.4. Pointwise Operations Before Matrix Multiplication

Pointwise operations can also precede a convolution or matrix multiplication, that is, g_1 is composed of pointwise operations.

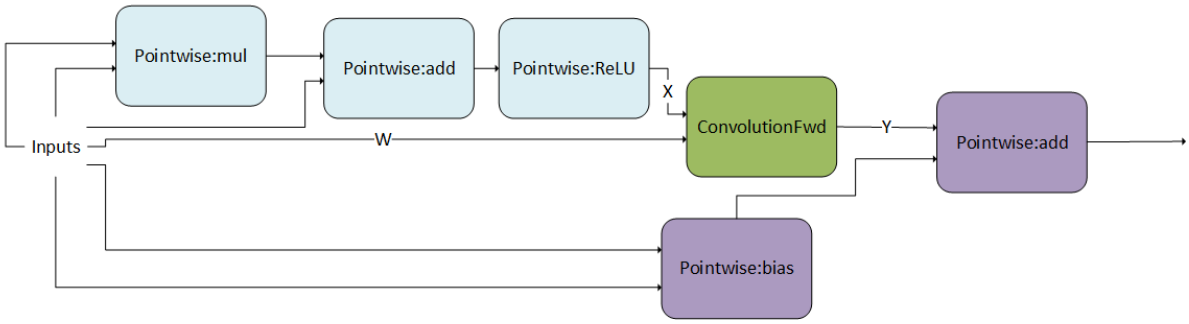
Figure 23. MatMul Preceded by a DAG with Two Operations



3.3.2.2.5. Convolution Producer Node in Middle of DAG

The following pattern shows g_1 as a DAG of pointwise operations feeding into a convolution. In addition, g_2 is a DAG consisting of two pointwise operations. Note that the convolution is being consumed in the middle of g_2 as opposed to g_2's first node. This is a valid pattern.

Figure 24. This example illustrates fusion of operations before and after the `ConvolutionFwd` operation. In addition we observe that the output of `ConvolutionFwd` can feed anywhere in g_2.



3.3.2.3. Operation specific Constraints for the Runtime Fusion Engine

Every operation in the supported generic patterns of the runtime fusion engine is subject to a few specific constraints regarding their parameter surface. The following subsections document these.

Note that these constraints are in addition to (1) any constraints mentioned in the NVIDIA cuDNN Backend API, and (2) limitations in relation to other operations in the directed acyclic graph (DAG), as mentioned in the Limitations section.

3.3.2.3.1. Convolutions

There are three operation nodes that represent different types of convolutions namely:

ConvolutionFwd
This operation represents forward convolution, that is, computing the response tensor of image tensor convoluted with filter tensor. For complete details on the interface, as well as general constraints, refer to the CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR section.

ConvolutionBwFilter
This operation represents convolution backward filters, that is, computing filter gradients from a response and an image tensor. For complete details on the interface, as well as general constraints, refer to the CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_FILTER_DESCRIPTOR section.

ConvolutionBwData
This operation represents convolution backward data, that is, computing input data gradients from a response and a filter tensor. For complete details on the interface, as well as general constraints, refer to the CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_DATA_DESCRIPTOR section.

Table 5. Tensor Attributes for all Three Operations

	Input Tensor Attribute Name	Output Tensor Attribute Name
ConvolutionFwd	CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_X CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_W	CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_
ConvolutionBwFilter	CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_DX CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_DY	CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA
ConvolutionBwData	CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_DW CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_DY	CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FIL

The following tables list the constraints for all three operations, in addition to any constraints mentioned in the NVIDIA cuDNN Backend API, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when these operations are used in the runtime fusion engine.

Table 6. Constraints for all Three Operations

Attribute	Support
CUDNN_ATTR_CONVOLUTION_MODE	CUDNN_CROSS_CORRELATION
CUDNN_ATTR_CONVOLUTION_COMP_TYPE	▶ For ConvolutionFwd CUDNN_DATA_HALF,CUDNN_DATA_INT32, andCUDNN_DATA_FLOAT

Attribute	Support
	▶ For ConvolutionBwDataand ConvolutionBwFilter ▶ Only CUDNN_DATA_FLOAT
CUDNN_ATTR_CONVOLUTION_SPATIAL_DIMS	2 or 3
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_ALPHA	1.0f
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_BETA	0.0f

Table 7. I/O Tensors Alignment Requirements

Tensor Data Type	Number of input and output channels for NVIDIA Hopper Architecture	Number of input and output channels for NVIDIA Ampere and Ada Lovelace	Number of input and output channels for NVIDIA Volta/Turing Architecture
INT8	Multiple of 4	Multiple of 4	Multiple of 16
FP8	Multiple of 16	N/A	N/A
FP16/BF16	Multiple of 2	Multiple of 2	Multiple of 8
FP32(TF32)	Any value	Any value	Multiple of 4

Lastly, there are some batch size requirements per operation:

Table 8. Batch Size Requirements Per Operation

Operation	Batch size for FP8 data type on NVIDIA Hopper Architecture	Batch size for other data types
ConvolutionFwd	Any	Any
ConvolutionBwFilter	Multiple of 16	Any
ConvolutionBwData	Multiple of 16	Any

The FP8 data type since Hopper architecture has two variants; CUDNN_DATA_FP8_E4M3 and CUDNN_DATA_FP8_E5M2 as I/O data types. It also has two possible compute types; CUDNN_DATA_FLOAT and CUDNN_DATA_FAST_FLOAT_FOR_FP8, which is a faster, but less accurate option for FP8 Tensor Core operations. It is sufficiently accurate for inference or the forward pass of training. However, for FP8 training backward pass computations (that is, computing weight and activation gradients), we recommend choosing the more accurate CUDNN_DATA_FLOAT compute type to preserve a higher level of accuracy which can be necessary for some models.

Table 9. Recommended compute type for FP8 tensor computations for Hopper architecture

Operation	Recommended I/O type	Recommended compute type
ConvolutionFwd	CUDNN_DATA_FP8_E4M3	▶ CUDNN_DATA_FAST_FLOAT_FOR_FP8 ▶ CUDNN_DATA_FLOAT
ConvolutionBwData	CUDNN_DATA_FP8_E4M3	CUDNN_DATA_FLOAT
BatchNorm	CUDNN_DATA_FP8_E4M3	CUDNN_DATA_FLOAT
Pooling	▶ CUDNN_DATA_FP8_E4M3 ▶ CUDNN_DATA_FP8_E5M2	CUDNN_DATA_FLOAT
Pointwise	▶ CUDNN_DATA_FP8_E4M3 ▶ CUDNN_DATA_FP8_E5M2	CUDNN_DATA_FLOAT

3.3.2.3.2. MatMul

This operation represents matrix-matrix multiplication: A * B = C. For complete details on the interface, refer to the CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR section.

The following two tables list the constraints for MatMul operations, in addition to any general constraints as listed in the NVIDIA cuDNN Backend API, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when MatMul is used in the runtime fusion engine.

Table 10. Constraints for MatMul Operations

Attribute	Support
CUDNN_ATTR_MATMUL_COMP_TYPE	CUDNN_DATA_HALF,CUDNN_DATA_INT32, and CUDNN_DATA_FLOAT

Table 11. MatMul Alignment Requirements

Tensor Data Type	Innermost dimension for NVIDIA Ampere Architecture and later	Innermost dimension for NVIDIA Volta/Turing Architecture
INT8	Multiple of 4	Multiple of 16
FP16/BF16	Multiple of 2	Multiple of 8
FP32(TF32)	Any value	Multiple of 4

3.3.2.3.3. Pointwise

Represents a pointwise operation that implements the equation `Y = op (alpha1 * X)` or `Y = op (alpha1 * X, alpha2 * B)`. Refer to the `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` and `CUDNN_BACKEND_POINTWISE_DESCRIPTOR` sections for more information and general constraints.

The following table lists the constraints for pointwise operations, in addition to the general constraints listed above, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when these operations are used in the runtime fusion engine.

Table 12. Constraints for Pointwise Operations

Attribute	Requirement
Tensor data type for `CUDNN_ATTR_OPERATION_POINTWISE_XDESC`, `CUDNN_ATTR_OPERATION_POINTWISE_YDESC` and, if applicable, `CUDNN_ATTR_OPERATION_POINTWISE_BDESC`	▶ For any of the logical operators (`CUDNN_POINTWISE_LOGICAL_AND`, `CUDNN_POINTWISE_LOGICAL_OR`, and `CUDNN_POINTWISE_LOGICAL_NOT`), data type can be any of `CUDNN_DATA_INT32`, `CUDNN_DATA_INT8`, or `CUDNN_DATA_BOOLEAN`. ▶ For all other operators, all data types are supported.
`CUDNN_ATTR_POINTWISE_MATH_PREC`	▶ For any of the logical operators (`CUDNN_POINTWISE_LOGICAL_AND`, `CUDNN_POINTWISE_LOGICAL_OR`, and `CUDNN_POINTWISE_LOGICAL_NOT`), math precision needs to be `CUDNN_DATA_BOOLEAN`. ▶ For all other operators, only `CUDNN_DATA_FLOAT` is supported.
`CUDNN_ATTR_OPERATION_POINTWISE_ALPHA1`	`1.0f`
`CUDNN_ATTR_OPERATION_POINTWISE_ALPHA2`	`1.0f`

3.3.2.3.4. GenStats

Represents an operation that generates per-channel statistics. Refer to the `CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR` section for more information and general constraints.

The following table lists the constraints for GenStats operations, in addition to the general constraints listed above, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when GenStats operations are used in the runtime fusion engine.

Table 13. Constraints for GenStats Operations

Attribute	Requirement
Tensor data type for CUDNN_ATTR_OPERATION_GENSTATS_XDESC	▶ Prior to the NVIDIA Ampere architecture GPU: CUDNN_DATA_HALF ▶ On NVIDIA Ampere architecture and later: CUDNN_DATA_HALF and CUDNN_DATA_FLOAT
Tensor shape for CUDNN_ATTR_OPERATION_GENSTATS_SUMDESC and CUDNN_ATTR_OPERATION_GENSTATS_SQSUMDESC	Both should be of shape [1, C, 1, 1] for 2D conv or [1, C, 1, 1, 1] for 3D conv.
Tensor data type for CUDNN_ATTR_OPERATION_GENSTATS_SUMDESC and CUDNN_ATTR_OPERATION_GENSTATS_SQSUMDESC	CUDNN_DATA_FLOAT
CUDNN_ATTR_POINTWISE_MATH_PREC	CUDNN_DATA_FLOAT
Tensor layout for CUDNN_ATTR_OPERATION_GENSTATS_XDESC, CUDNN_ATTR_OPERATION_GENSTATS_SUMDESC and CUDNN_ATTR_OPERATION_GENSTATS_SQSUMDESC	NHWC fully packed

3.3.2.3.5. Reduction

This operation represents reducing values of a tensor in one or more dimensions. Refer to the CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR section for more information and general constraints.

The following two tables are constraints for Reduction forward operations, in addition to the general constraints listed above, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when Reduction operations are used in the runtime fusion engine.

Table 14. Constraints for Reduction Operations

Attribute	Requirement
Tensor data type for CUDNN_ATTR_OPERATION_REDUCTION_YDESC	CUDNN_DATA_FLOAT
CUDNN_ATTR_REDUCTION_COMP_TYPE	CUDNN_DATA_FLOAT
Tensor layout for CUDNN_ATTR_OPERATION_REDUCTION_XDESC and CUDNN_ATTR_OPERATION_REDUCTION_YDESC	NHWC/NDHWC/BMN fully packed
CUDNN_ATTR_REDUCTION_OPERATOR	CUDNN_REDUCE_TENSOR_ADD, CUDNN_REDUCE_TENSOR_MIN, and CUDNN_REDUCE_TENSOR_MAX

Table 15. Supported Reduction Patterns

Reduction Operation	Reduction Pattern	
	Input	Output
Standalone reduction operation	[N, C, H, W]	[N, 1, H, W]
		[1, C, 1, 1]
		[1, 1, 1, 1]
Reduction fused after convolution backward filter gradient	[N, K, P, Q]	[N, 1, P, Q]
		[1, K, 1, 1]
		[1, 1, 1, 1]
Reduction fused after convolution backward data gradient	[N, C, H, W]	[N, 1, H, W]
		[1, C, 1, 1]
		[1, 1, 1, 1]
Reduction fused after convolution backward filter gradient	[K, C, R, S]	[K, 1, 1, 1]
		[1, C, R, S]
		[1, 1, 1, 1]
Reduction fused after matrix multiplication operation	[B, M, N]	[B, M, 1]
		[B, 1, N]

3.3.2.3.6. `ResampleFwd`

This operation represents resampling of the spatial dimensions of an image to a desired value. Resampling is supported in both directions, upsampling and downsampling. Downsampling represents the standard operation of pooling, commonly used in convolutional neural networks. Refer to the `CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR` section for more information and general constraints.

The following are constraints for Resample operations, in addition to the general constraints listed above, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when Resample forward operations are used in the runtime fusion engine.

We allow a choice amongst four modes for resample. All modes have the following common support specifications:

- Supported layout: NHWC or NDHWC, NCHW or NCDHW
- Spatial dimensions supported: 2 or 3
- Input dimensions supported: 4 or 5
- If specified, the index tensor dimension should be equal to the response tensor dimension.

There are some mode specific restrictions also. The following tables list the values that are allowed for particular parameters. For the parameters not listed, we allow any value which is mathematically correct.

The following downsampling modes are supported:

- CUDNN_RESAMPLE_AVGPOOL_INCLUDE_PADDING
- CUDNN_RESAMPLE_AVGPOOL_EXCLUDE_PADDING
- CUDNN_RESAMPLE_MAXPOOL

Table 16. Specific Restrictions for the Downsampling Modes

Attribute	Average Pooling	Max Pooling
CUDNN_ATTR_RESAMPLE_PADDING_MODE	CUDNN_ZERO_PAD	CUDNN_NEG_INF_PAD
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_ALPHA	1.0	1.0
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_BETA	0.0	0.0
CUDNN_ATTR_RESAMPLE_COMP_TYPE	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT

For the upsampling modes, CUDNN_RESAMPLE_NEAREST is not supported for any combination of parameters. CUDNN_RESAMPLE_BILINEAR has the following support specifications.

Table 17. Specific Restrictions for Upsampling Mode CUDNN_RESAMPLE_BILINEAR

Attribute	Bilinear
Input dimensions	Equal to 0.5 x output dimensions
CUDNN_ATTR_RESAMPLE_PRE_PADDINGS	0.5
CUDNN_ATTR_RESAMPLE_POST_PADDINGS	1
CUDNN_ATTR_RESAMPLE_STRIDES	0.5
CUDNN_ATTR_RESAMPLE_WINDOW_DIMS	2
Data type for CUDNN_ATTR_OPERATION_RESAMPLE_FWD_XDESC and CUDNN_ATTR_OPERATION_RESAMPLE_FWD_YDESC	CUDNN_DATA_FLOAT
CUDNN_ATTR_RESAMPLE_COMP_TYPE	CUDNN_DATA_FLOAT
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_ALPHA	1.0
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_BETA	0.0
CUDNN_ATTR_RESAMPLE_PADDING_MODE	CUDNN_EDGE_VAL_PAD

3.3.2.3.6.1. Resampling Index Tensor Dump for Training

For max-pooling resampling mode, an index tensor can be provided to be used as a mask for backpropagation.

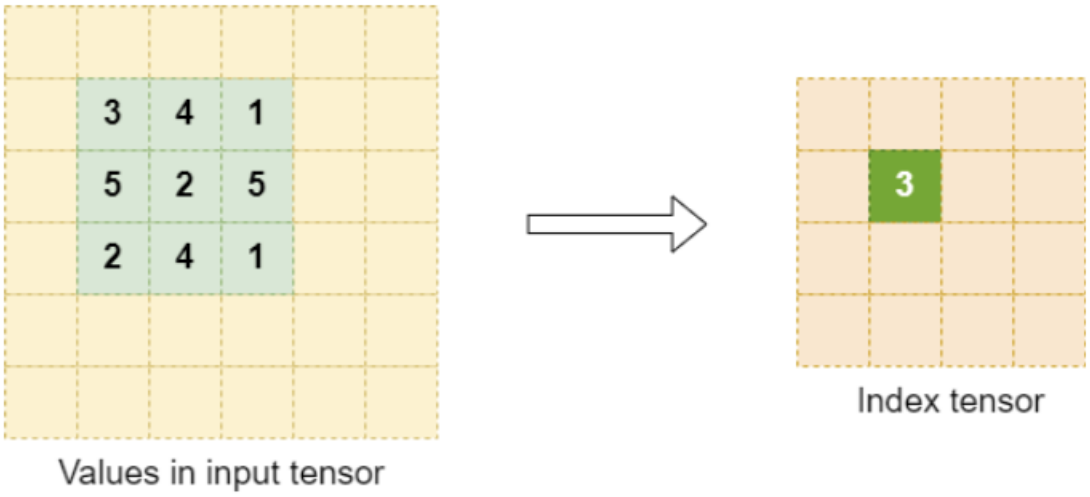
Values in the index tensors are:

- Zero-indexed row-major position of maximum value of input tensor in the resampling window.

- In case of multiple input pixels with maximum value, the first index in a left-to-right top-to-bottom scan is selected.

Example of index element selection:

Figure 25. Values In the Index Tensors



Select an appropriate element size for the index tensor. As a reference, any element size such that the maximum zero-indexed window position fits should be sufficient.

3.3.2.3.7. `ResampleBwd`

This operation represents backward resampling of the spatial dimensions of an output response to a desired value. Resampling is supported in both directions, upsampling and downsampling. Backwards downsampling represents the standard operation of backward pooling, commonly used in convolutional neural networks. Refer to the `CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR` section for more information and general constraints.

The following are constraints for Resample backward operations, in addition to the general constraints listed above, and any constraints listed in the Limitations section, in relation to other operations. Note that these additional constraints only apply when Resample backward operations are used in the runtime fusion engine.

We allow a choice amongst four modes for resample. All modes have the following common support specifications:

- Supported layout: NHWC or NDHWC, NCHW or NCDHW
- Spatial dimensions supported: 2 or 3
- Input dimensions supported: 4 or 5
- The index tensor dimensions should be equal to the input gradient tensor dimensions.

Index tensor should be provided for only max pooling mode, and should adhere to the format described in the resampling forward index dump section.

There are some mode specific restrictions also. The following tables list the values that are allowed for particular parameters. For the parameters not listed, we allow any value which is mathematically correct.

The following backward downsampling modes are supported:

- `CUDNN_RESAMPLE_AVGPOOL_INCLUDE_PADDING`
- `CUDNN_RESAMPLE_AVGPOOL_EXCLUDE_PADDING`
- `CUDNN_RESAMPLE_MAXPOOL`

Table 18. Specific Restrictions for the Backwards Downsampling Modes

Attribute	Average Pooling	Max Pooling
`CUDNN_ATTR_RESAMPLE_PADDING_MODE`	`CUDNN_ZERO_PAD`	`CUDNN_NEG_INF_PAD`
`CUDNN_ATTR_OPERATION_RESAMPLE_BWD_ALPHA`	`1.0`	`1.0`
`CUDNN_ATTR_OPERATION_RESAMPLE_BWD_BETA`	`0.0`	`0.0`
`CUDNN_ATTR_RESAMPLE_COMP_TYPE`	`CUDNN_DATA_FLOAT`	`CUDNN_DATA_FLOAT`

Backward upsampling modes are currently not supported.

3.3.3. Pre-Compiled Specialized Engines

The pre-compiled specialized engines target and optimize for a specialized graph pattern with a ragged support surface. Because of this targeting, these graphs do not require runtime compilation.

In most cases, the specialized patterns are just special cases of the generic patterns used in the runtime fusion engine, but there are some cases where the specialized pattern does not fit any of the generic patterns. If your graph pattern matches a specialized pattern, you will get at least a pattern matching engine, and you might also get a runtime fusion engine as another option.

Currently, the following patterns are supported by the pattern matching engines. Some nodes are optional. Optional nodes are indicated by dashed outlines.

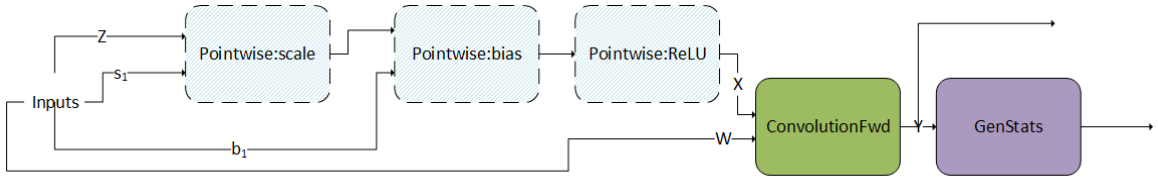
3.3.3.1. ConvBNfprop

In Figure 26, the `ConvBNfprop` pattern is illustrated. Its restrictions and options include:

- The three pointwise nodes scale, bias, and ReLU are optional.
- X, Z, W, s_1, b_1 must all be of FP16 data type.
- Z needs to be of shape [N, C, H, W] with NHWC packed layout.
- W needs to be of shape [K, C, R, S] with KRSC packed layout.

- s_1, b_1 need to be of shape [1, C, 1, 1] with NHWC packed layout.
- Only ReLU activation is supported.
- All of the intermediate tensors need to be virtual, except, Y needs to be non-virtual.
- I/O pointers should be 16 bytes aligned.

Figure 26. The pre-compiled `ConvBNfprop` engine fuses several pointwise operations with `ConvolutionFwd` and `GenStats`.

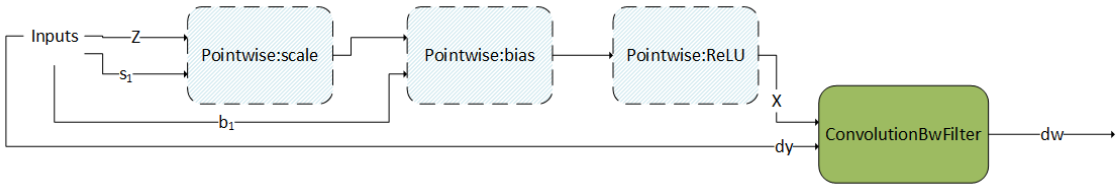


3.3.3.2. ConvBNwgrad

In Figure 27, the `ConvBNwgrad` pattern is illustrated. Its restrictions and options include:

- The three pointwise operations are all optional, as indicated by the dashed outlines.
- Only ReLU activation is supported.
- X, s_1, b_1, and `dy` must all be of FP16 datatype.
- I/O pointers should be 16 bytes aligned.

Figure 27. The `ConvBNwgrad` pre-compiled engine fuses several (optional) pointwise operations with `ConvolutionBwFilter`.



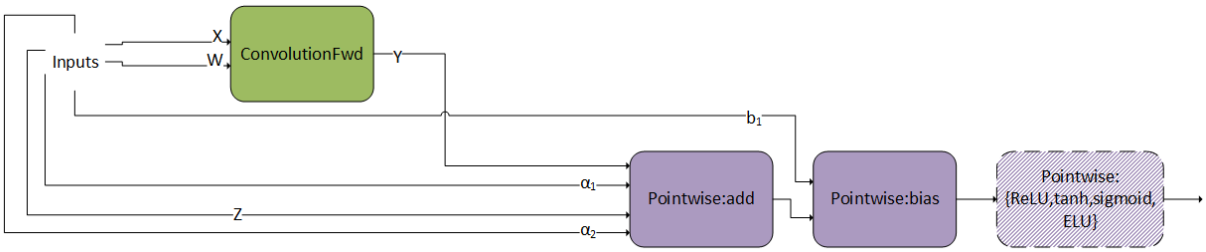
3.3.3.3. ConvBiasAct

In the following figure, the `ConvBiasAct` pattern is illustrated. Its restrictions and options include:

- α_1 and α_2 need to be scalars.
- The activation node is optional.
- The size of the bias tensor should be [1, K, 1, 1].

- Internal conversions are not supported. That is, the virtual output between nodes need to have the same data type as the node's compute type, which should be the same as the epilog type of the convolution node.
- There are some restrictions on the supported combination of data types, which can be found in the API Reference (refer to `cudnnConvolutionBiasActivationForward()`).

Figure 28. `ConvBiasAct`, another pre-compiled engine, fuses `ConvolutionFwd` with several pointwise operations.

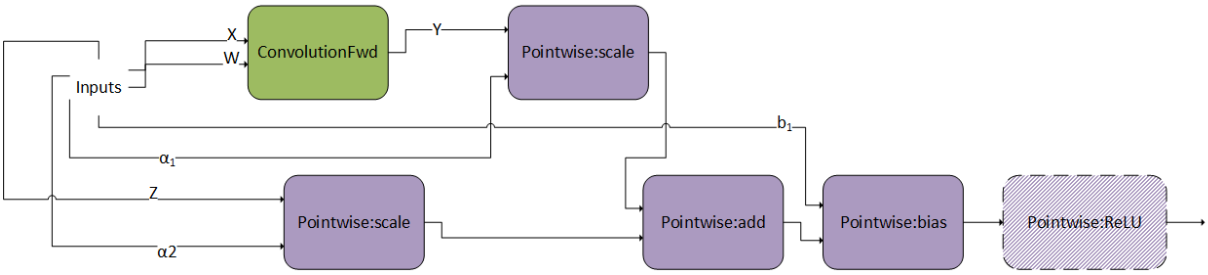


3.3.3.4. `ConvScaleBiasAct`

In the following figure, the `ConvScaleBiasAct` pattern is illustrated. Its restrictions and options include:

- α_1 and α_2 and b_2 should have the same data type/layout and can only be FP32.
- X, W, and Z can only be INT8x4 or INT8x32.
- The size of the bias tensor should be [1, K, 1, 1].
- Internal conversions are not supported. Meaning, "virtual output" between nodes needs to be the same as their compute type.
- Currently, `Pointwise:ReLU` is the only optional pointwise node.

Figure 29. The pre-compiled engine, `ConvScaleBiasAct`



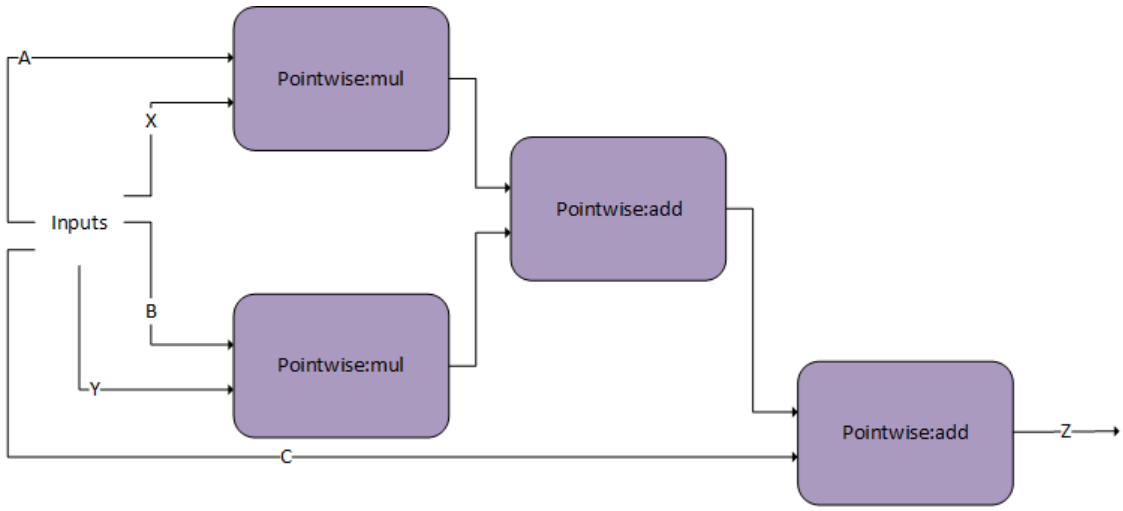
This pattern is very similar as `ConvBiasAct`. The difference is that here, the scales α_1 and α_2 are tensors, not scalars. If they are scalars, this pattern becomes a normal `ConvBiasAct`.

3.3.3.5. dBNapply

In Figure 30, the `dBNapply` pattern is illustrated. Its restrictions and options include:

- One of the inputs to the `mul` nodes and the input to the final add node must be of FP32 datatype (A, B, C).
- The other inputs to the `mul` nodes (X and Y) must be of FP16 data type.
- X, Y and Z are 4D tensors – [N,C,H,W] with NHWC packed layout.
- A, B, C are 1D tensors - [1,C,1,1] with NHWC packed layout.
- Channel C should be a multiple of 16 for all the tensors.
- Tensors A and B should be attached to the B port of the `mul` nodes; tensors X and Y should be attached to the X port.

Figure 30. The pre-compiled engine, `dBNapply`



The pattern implements a simple linear combination:

- Z = A*X + B*Y + C

3.3.3.6. DualdBNapply

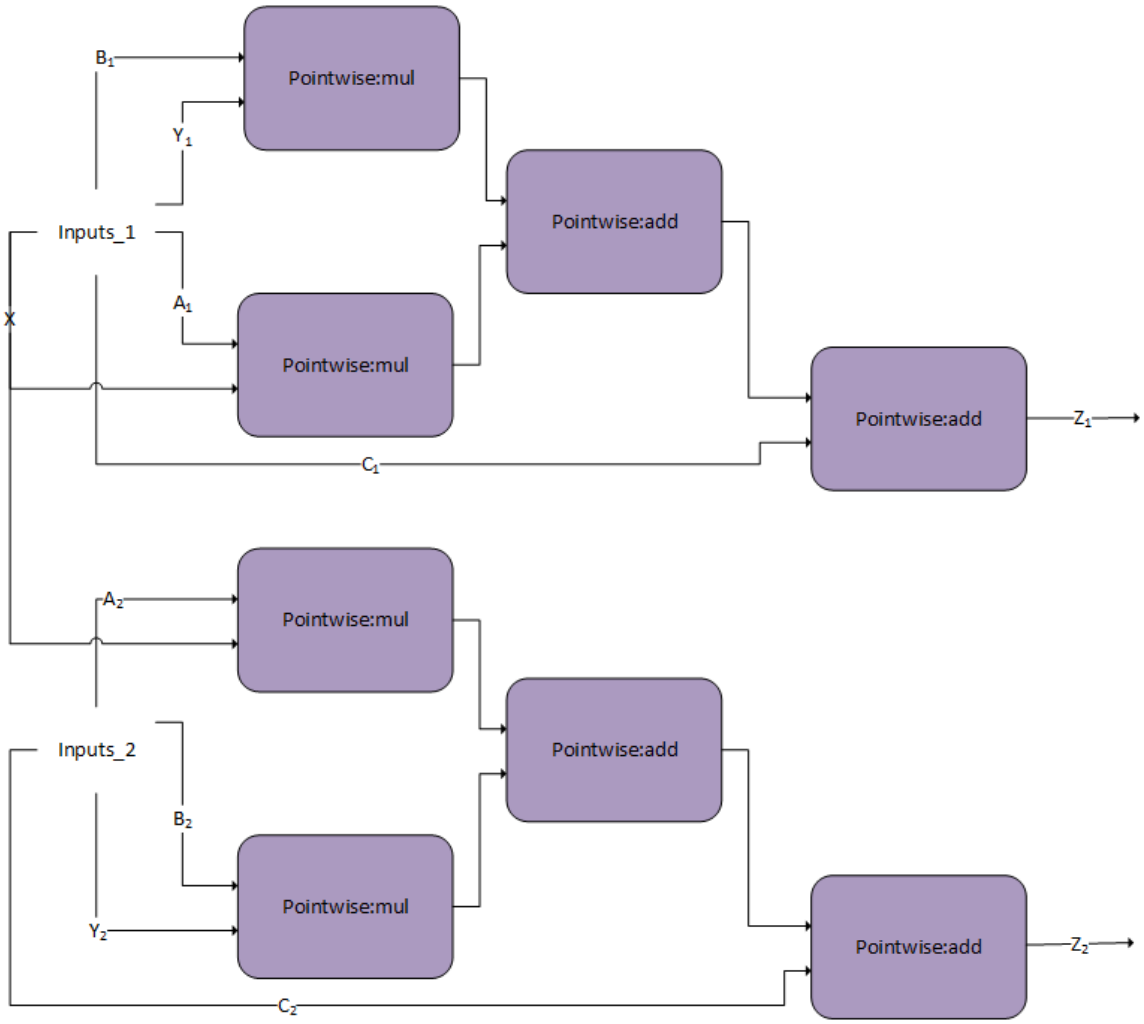
In Figure 26, the `DualdBNapply` pattern is illustrated. Its restrictions and options include:

- One tensor X is shared between the two linear combinations.
- Five tensors, X, Y_1, Y_2, Z_1, Z_2 are 4D tensors [N,C,H,W] with NHWC packed layout.
- Six tensors A_1, A_2, B_1, B_2 ,C_1,C_2 are 1D tensors [1,C,1,1].
- Channel C should be a multiple of 16 for all the tensors.

In essence, `DualdBNapply` runs the previous pattern, `dBNapply` twice, as two subgraphs. However, both subgraphs share one input tensor, X.

Note that for visibility purposes, the Inputs block is split into `Inputs_1` and `Inputs_2`. This has no semantic meaning.

Figure 31. The `DualdBNapply` engine



This pattern implements two linear combinations:

- $Z_1 = A_1 * X + B_1 * Y_1 + C_1$
- $Z_2 = A_2 * X + B_2 * Y_2 + C_2$

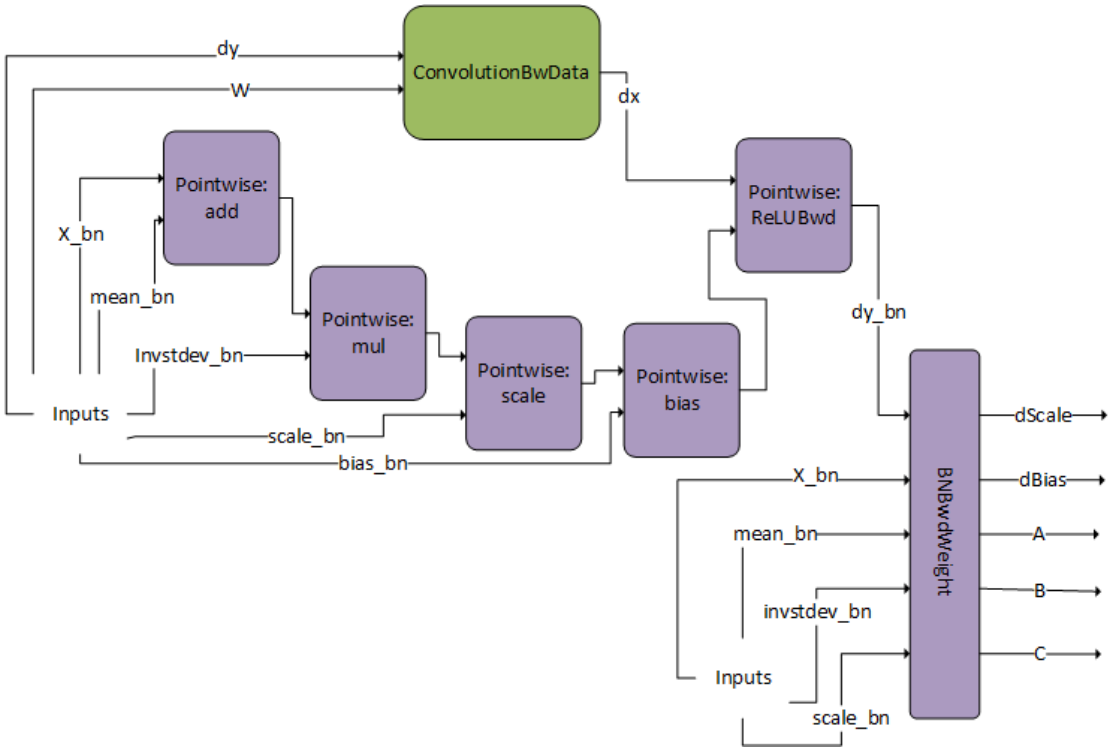
3.3.3.7. `DgradDreluBNBwdWeight`

In Figure 32, the `DgradDreluBNBwdWeight` pattern is illustrated. Its restrictions and options include:

- `Dgrad` input `dy` and W are of FP16 datatypes.
- Batch norm fwd inputs, `X_bn` is of FP16 datatype while the other tensors `mean_bn`, `invstd_dev_bn`, `scale_bn`, and `bias_bn` are FP32.

- Outputs: `dScale`, `dBias`, A,B,C are of FP32 data type.
- All pointers are 16 byte aligned.
- Only supported on NVIDIA Ampere architecture GPUs.

Figure 32. `DgradDreluBNBwdWeight` is a pre-compiled engine that can be used in conjunction with the `dBNApply` pattern to compute the backwards path of batch norm.



The `BNBwdWeight` operation takes in five inputs: `X_bn`, `mean_bn`, `invstddev_bn`, `scale_bn`, and `dy_bn` (that is, the output from the `ReLUBwd` node).

It produces five outputs: gradients of the batch norm scale and bias params, `dScale`, `dBias`, and coefficients A,B,C. Note that for illustration purposes, the inputs are duplicated. The inputs on the left and right are however exactly the same.

This pattern is typically used in the computation of the Batch Norm Backward Pass.

When computing the backward pass of batch norm, `dScale`, `dBias`, and `dX_bn` are needed. The `DgradDreluBnBwdWeight` pattern computes the former two. Using the generated A, B, and C we can use the `dBNApply` pattern above to compute `dX`, the input gradient, as follows `dx_bn = A*dy_bn + B*X_bn +C`.

Note that this pattern is used in combination with the forward pass, the `ConvBNfprop` pattern. Because of performance reasons, the output of batch norm `Y_bn`, which was calculated in `ConvBNfprop` (output of scale-bias), needs to be recalculated by `DgradDreluBnBwdWeight`. The pointwise add node subtracts `mean_bn` from `X_bn`, hence the `alpha2` parameter for that node should be set to `-1`.

3.3.4. Mapping with Backend Descriptors

For readability, the operations used in this section are abbreviated. The mapping with the actual backend descriptors can be found in this table:

Table 19. Notations and Backend Descriptors

Notation used in this section	Backend descriptor
`Pointwise:scale`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_MUL` and with operand B broadcasting into operand X
`Pointwise:bias`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_ADD` and with operand B broadcasting into operand X
`Pointwise:add`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_ADD` and with operand B with same dimensions as X
`Pointwise:mul`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_MUL` and with operand B with same dimensions as X
`Pointwise:ReLU`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_RELU_FWD`
`Pointwise:ReLUBwd`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_RELU_BWD`
`Pointwise:tanh`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_TANH_FWD`
`Pointwise:sigmoid`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_SIGMOID_FWD`
`Pointwise:ELU`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with mode `CUDNN_POINTWISE_ELU_FWD`

`Pointwise:{ReLU,tanh,sigmoid,ELU}`	`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` with one of the following modes:
	`CUDNN_POINTWISE_RELU_FWD,`
	`CUDNN_POINTWISE_TANH_FWD,`
	`CUDNN_POINTWISE_SIGMOID_FWD,`
	`CUDNN_POINTWISE_ELU_FWD`

`MatMul`	`CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR`
`ConvolutionFwd`	`CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR`
`ConvolutionBwFilter`	`CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_FILTER_DES`
`ConvolutionBwData`	`CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_DATA_DESC`
`GenStats`	`CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR`
`ResampleFwd`	`CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR`
`GenStats`	`CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR`
`Reduction`	`CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR`

Notation used in this section	Backend descriptor
BnBwdWeight	CUDNN_BACKEND_OPERATION_BN_BWD_WEIGHTS_DESCRIPTOR
BOOLEAN/packed-BOOLEAN	CUDNN_DATA_BOOLEAN: As described in the NVIDIA cuDNN API Reference, this type implies that eight boolean values are packed in a single byte, with the lowest index on the right (that is, least significant bit). packed-BOOLEAN and BOOLEAN are used interchangeably, where the former is used to emphasize and remind the user about the semantics.
INT8	CUDNN_DATA_INT8
FP8	CUDNN_DATA_FP8_E4M3 or CUDNN_DATA_FP8_E5M2
FP16	CUDNN_DATA_HALF
BF16	CUDNN_DATA_BFLOAT16
FP32	CUDNN_DATA_FLOAT
TF32	A tensor core operation mode used to accelerate floating point convolutions or matmuls. This can be used for an operation with compute type CUDNN_DATA_FLOAT, on NVIDIA Ampere architecture or later and be disabled with NVIDIA_TF32_OVERRIDE=1.

Chapter 4. Legacy API

4.1. Convolution Functions

4.1.1. Prerequisites

For the supported GPUs, the Tensor Core operations will be triggered for convolution functions only when `cudnnSetConvolutionMathType()` is called on the appropriate convolution descriptor by setting the `mathType` to `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`.

4.1.2. Supported Algorithms

When the prerequisite is met, the below convolution functions can be run as Tensor Core operations:

- `cudnnConvolutionForward()`
- `cudnnConvolutionBackwardData()`
- `cudnnConvolutionBackwardFilter()`

Refer to the following table for a list of supported algorithms:

Supported Convolution Function	Supported Algos
`cudnnConvolutionForward`	`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED`
`cudnnConvolutionBackwardData`	`CUDNN_CONVOLUTION_BWD_DATA_ALGO_1` `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED`
`cudnnConvolutionBackwardFilter`	`CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED`

4.1.3. Data and Filter Formats

The cuDNN library may use padding, folding, and NCHW-to-NHWC transformations to call the Tensor Core operations. For more information, refer to Tensor Transformations.

For algorithms other than `*_ALGO_WINOGRAD_NONFUSED`, when the following requirements are met, the cuDNN library will trigger the Tensor Core operations:

- Input, filter, and output descriptors (`xDesc`, `yDesc`, `wDesc`, `dxDesc`, `dyDesc` and `dwDesc` as applicable) are of the `dataType = CUDNN_DATA_HALF` (that is, FP16). For FP32 `dataType`, refer to Conversion Between FP32 and FP16.
- The number of input and output feature maps (that is, channel dimension `C`) is a multiple of 8. When the channel dimension is not a multiple of 8, refer to Padding.
- The filter is of type `CUDNN_TENSOR_NCHW` or `CUDNN_TENSOR_NHWC`.
- If using a filter of type `CUDNN_TENSOR_NHWC`, then the input, filter, and output data pointers (`X`, `Y`, `W`, `dX`, `dY`, and `dW` as applicable) are aligned to 128-bit boundaries.

4.2. RNN Functions

4.2.1. Prerequisites

Tensor Core operations are triggered for these RNN functions only when `cudnnSetRNNMatrixMathType()` is called on the appropriate RNN descriptor setting `mathType` to `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`.

4.2.2. Supported Algorithms

When the above prerequisites are met, the RNN functions below can be run as Tensor Core operations:

- `cudnnRNNForwardInference()`
- `cudnnRNNForwardTraining()`
- `cudnnRNNBackwardData()`
- `cudnnRNNBackwardWeights()`
- `cudnnRNNForwardInferenceEx()`
- `cudnnRNNForwardTrainingEx()`
- `cudnnRNNBackwardDataEx()`
- `cudnnRNNBackwardWeightsEx()`
- `cudnnRNNForward()`
- `cudnnRNNBackwardData_v8()`
- `cudnnRNNBackwardWeights_v8()`

Refer to the following table for a list of supported algorithms:

RNN Function	Support Algos
All RNN functions that support Tensor Core operations.	`CUDNN_RNN_ALGO_STANDARD` `CUDNN_RNN_ALGO_PERSIST_STATIC`

4.2.3. Data and Filter Formats

When the following requirements are met, then the cuDNN library triggers the Tensor Core operations:

- For `algo = CUDNN_RNN_ALGO_STANDARD`:
  - The hidden state size, input size, and the batch size is a multiple of 8.
  - All user-provided tensors, workspace, and reserve space are aligned to 128-bit boundaries.
  - For FP16 input/output, the `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
  - For FP32 input/output, `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
- For `algo = CUDNN_RNN_ALGO_PERSIST_STATIC`:
  - The hidden state size and the input size is a multiple of 32.
  - The batch size is a multiple of 8.
  - If the batch size exceeds 96 (for forward training or inference) or 32 (for backward data), then the batch size constraints may be stricter, and large power-of-two batch sizes may be needed.
  - All user-provided tensors, workspace, and reserve space are aligned to 128-bit boundaries.
  - For FP16 input/output, `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
  - For FP32 input/output, `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.

For more information, refer to Features of RNN Functions.

4.2.4. Features of RNN Functions

Refer to the following table for a list of features supported by each RNN function.

Note:

For each of these terms, the short-form versions shown in the parenthesis are used in the tables below for brevity: `CUDNN_RNN_ALGO_STANDARD` (`_ALGO_STANDARD`), `CUDNN_RNN_ALGO_PERSIST_STATIC` (`_ALGO_PERSIST_STATIC`), `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` (`_ALGO_PERSIST_DYNAMIC`), and `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` (`_ALLOW_CONVERSION`).

Functions	I/O layout supported	Supports variable sequence length in batch	Commonly supported
cudnnRNNForwardInference() cudnnRNNForwardTraining() cudnnRNNBackwardData() cudnnRNNBackwardWeights()	Only Sequence major, packed (non-padded)	Only with _ALGO_STANDARD Require input sequences descending sorted according to length.	Mode (cell type) supported:CUDNN_RNN_RELU, CUDNN_RNN_TANH, CUDNN_LSTM, CUDNN_GRU Algo supported[1] (refer to the table for for information on these algorithms):_ALGO_STANDARD, _ALGO_PERSIST_STATIC, _ALGO_PERSIST_DYNAMIC Math mode supported: CUDNN_DEFAULT_MATH,CUDNN_TENSOR_O (will automatically fall back if run on pre-Volta or if algo doesn't support Tensor Cores) _ALLOW_CONVERSION (may perform down conversion to utilize Tensor Cores) Direction mode supported: CUDNN_UNIDIRECTIONAL, CUDNN_BIDIRECTIONAL RNN input mode: CUDNN_LINEAR_INPUT, CUDNN_SKIP_INPUT
cudnnRNNForwardInferenceEx() cudnnRNNForwardTrainingEx() cudnnRNNBackwardDataEx() cudnnRNNBackwardWeightsEx()	Sequence major unpacked Batch major unpacked[2] Sequence major packed[3]	Only with _ALGO_STANDARD For unpacked layout, no input sorting required. [4] For packed layout, require input sequences descending sorted according to length.	

The following table provides the features supported by the algorithms referred in the above table: CUDNN_RNN_ALGO_STANDARD, CUDNN_RNN_ALGO_PERSIST_STATIC, and CUDNN_RNN_ALGO_PERSIST_DYNAMIC.

Features	_ALGO_STANDARD	_ALGO_PERSIST_S	CUDNN_RNN_ALGO_	_ALGO_PERSIST_D	MATCH
Half input Single accumulation Half output	Supported Half intermediate storage Single accumulation				

[1] Do not mix different algos for different steps of training. It's also not recommended to mix non-extended and extended API for different steps of training.
[2] To use an unpacked layout, users need to set CUDNN_RNN_PADDED_IO_ENABLED through cudnnSetRNNPaddingMode().
[4] To use an unpacked layout, set CUDNN_RNN_PADDED_IO_ENABLED through cudnnSetRNNPaddingMode().
[3] To use an unpacked layout, users need to set CUDNN_RNN_PADDED_IO_ENABLED through cudnnSetRNNPaddingMode().

Features	_ALGO_STANDARD	_ALGO_PERSIST_S	CUDNN_RNN_ALGO_	_ALGO_PERSIST_DYNAMIC
Single input Single accumulation Single output	Supported If running on Volta, with CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION[5], will down-convert and use half intermediate storage. Otherwise: Single intermediate storage Single accumulation			
Double input Double accumulation Double output	Supported Double intermediate storage Double accumulation	Not Supported	Not Supported	Supported Double intermediate storage Double accumulation
LSTM recurrent projection	Supported	Not Supported	Not Supported	Not Supported
LSTM cell clipping	Supported			
Variable sequence length in batch	Supported	Not Supported	Not Supported	Not Supported
Tensor Cores	Supported For half input/output, acceleration requires setting CUDNN_TENSOR_OP_MATH[6] or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION [7] Acceleration requires inputSize and hiddenSize to be a multiple of 8 For single input/output on NVIDIA Volta, NVIDIA Xavier, and NVIDIA Turing, acceleration requires setting CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION[8] Acceleration requires inputSize and hiddenSize to be a multiple of 8 For single input/output on NVIDIA Ampere architecture, acceleration requires setting			Not Supported, will execute normally ignoring CUDNN_TENSOR_OP_MATH[10] or _ALLOW_CONVERSION[11]

[5] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

[10] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

[6] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

[7] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

[11] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

[8] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

Features	_ALGO_STANDARD	_ALGO_PERSIST_S	CUDNN_RNN_ALGO_	_ALGO_PERSIST_DYNAMIC
	CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH, or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION[9] Acceleration requires inputSize and hiddenSize to be a multiple of 4.			
Other limitations		Max problem size is limited by GPU specifications.	Forward RNN: ▶ RELU and TANH RNN: hidden_size <= 384 ▶ LSTM and GRU: hidden_size <= 192 BackwardData RNN: ▶ RELU and TANH RNN: hidden_size <= 256 ▶ LSTM and GRU: hidden_size <= 128	Requires real time compilation through NVRTC

4.3. Tensor Transformations

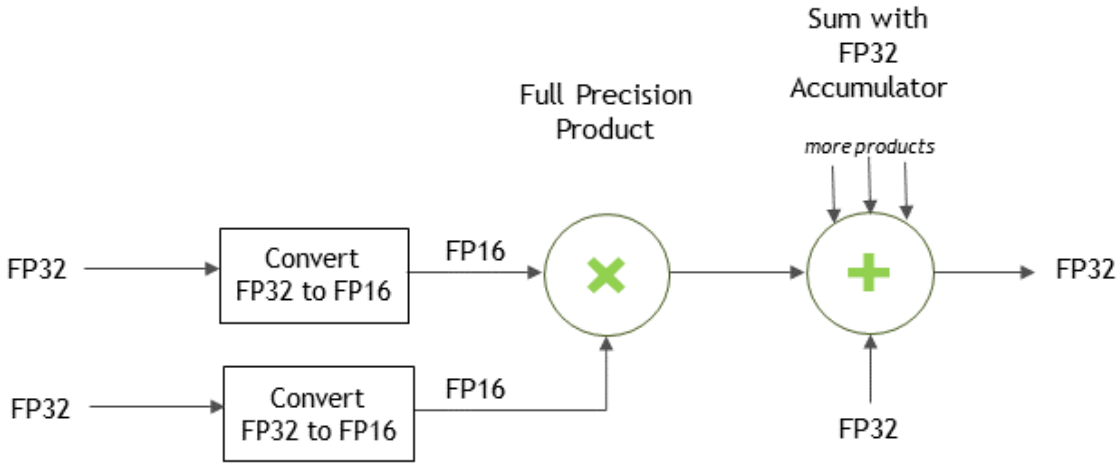
A few functions in the cuDNN library will perform transformations such as folding, padding, and NCHW-to-NHWC conversion while performing the actual function operation.

4.3.1. Conversion Between FP32 and FP16

The cuDNN API Reference allows you to specify that FP32 input data may be copied and converted to FP16 data internally to use Tensor Core operations for potentially improved performance. This can be achieved by selecting CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION enum for cudnnMathType_t. In this mode, the FP32 tensors are internally down-converted to FP16, the Tensor Op math is performed, and finally up-converted to FP32 as outputs. For more information, refer to Figure 33.

[9] CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().

Figure 33. Tensor Operation with FP32 Inputs



For Convolutions

For convolutions, the FP32-to-FP16 conversion can be achieved by passing the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum value to the `cudnnSetConvolutionMathType()` call.

```
// Set the math type to allow cuDNN to use Tensor Cores:
checkCudnnErr(cudnnSetConvolutionMathType(cudnnConvDesc,
 CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION));
```

For RNNs

For RNNs, the FP32-to-FP16 conversion can be achieved by passing the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum value to the `cudnnSetRNNMatrixMathType()` call to allow FP32 data to be converted for use in RNNs.

```
// Set the math type to allow cuDNN to use Tensor Cores:
checkCudnnErr(cudnnSetRNNMatrixMathType(cudnnRnnDesc,
 CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION));
```

4.3.2. Padding

For packed NCHW data, when the channel dimension is not a multiple of 8, then the cuDNN library will pad the tensors as needed to enable Tensor Core operations. This padding is automatic for packed NCHW data in both the `CUDNN_TENSOR_OP_MATH` and the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` cases.

4.3.3. Folding

In the folding operation, the cuDNN library implicitly performs the formatting of input tensors and saves the input tensors in an internal workspace. This can lead to an acceleration of the call to Tensor Cores.

With folding or channel-folding, cuDNN can implicitly format the input tensors within an internal workspace to accelerate the overall calculation. Performing this transformation for the user often allows cuDNN to use kernels with restrictions on convolution stride to support a strided convolution problem.

4.3.4. Conversion Between NCHW And NHWC

Tensor Cores require that the tensors be in the NHWC data layout. Conversion between NCHW and NHWC is performed when the user requests Tensor Op math. However, a request to use Tensor Cores is just that, a request and Tensor Cores may not be used in some cases. The cuDNN library converts between NCHW and NHWC if and only if Tensor Cores are requested and are actually used.

If your input (and output) are NCHW, then expect a layout change.

Non-Tensor Op convolutions will not perform conversions between NCHW and NHWC.

In very rare and difficult-to-qualify cases that are a complex function of padding and filter sizes, it is possible that Tensor Ops is not enabled. In such cases, users can pre-pad to enable the Tensor Ops path.

4.4. Mixed Precision Numerical Accuracy

When the computation precision and the output precision are not the same, it is possible that the numerical accuracy will vary from one algorithm to the other.

For example, when the computation is performed in FP32 and the output is in FP16, the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` (`ALGO_0`) has lower accuracy compared to the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` (`ALGO_1`). This is because `ALGO_0` does not use extra workspace, and is forced to accumulate the intermediate results in FP16, that is, half precision float, and this reduces the accuracy. The `ALGO_1`, on the other hand, uses additional workspace to accumulate the intermediate values in FP32, that is, full precision float.

Chapter 5. Odds and Ends

This section includes a random set of topics and concepts.

5.1. Thread Safety

The cuDNN library is thread-safe. Its functions can be called from multiple host threads, so long as the threads do not share the same cuDNN handle simultaneously.

When creating a per-thread cuDNN handle, it is recommended that a single synchronous call of `cudnnCreate()` be made first before each thread creates its own handle asynchronously.

Per `cudnnCreate()`, for multi-threaded applications that use the same device from different threads, the recommended programming model is to create one (or a few, as is convenient) cuDNN handles per thread and use that cuDNN handle for the entire life of the thread.

5.2. Reproducibility (Determinism)

By design, most of cuDNN's routines from a given version generate the same bit-wise results across runs when executed on GPUs with the same architecture. There are some exceptions. For example, the following routines do not guarantee reproducibility across runs, even on the same architecture, because they use atomic operations in a way that introduces truly random floating point rounding errors:

- `cudnnConvolutionBackwardFilter` when `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` or `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3` is used
- `cudnnConvolutionBackwardData` when `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0` is used
- `cudnnPoolingBackward` when `CUDNN_POOLING_MAX` is used
- `cudnnSpatialTfSamplerBackward`
- `cudnnCTCLoss` and `cudnnCTCLoss_v8` when `CUDNN_CTC_LOSS_ALGO_NON_DETERMINSTIC` is used

Across different architectures, no cuDNN routines guarantee bit-wise reproducibility. For example, there is no guarantee of bit-wise reproducibility when comparing the same

routine run on NVIDIA Volta™ and NVIDIA Turing™, NVIDIA Turing, and NVIDIA Ampere architecture.

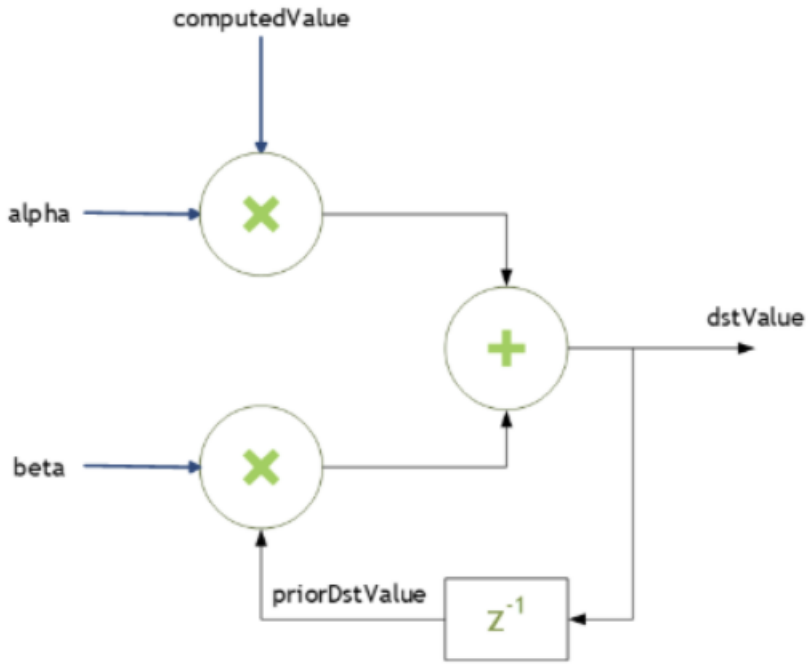
5.3. Scaling Parameters

Many cuDNN routines like `cudnnConvolutionForward()` accept pointers in host memory to scaling factors `alpha` and `beta`. These scaling factors are used to blend the computed values with the prior values in the destination tensor as follows (refer to Figure 34):

```
dstValue = alpha*computedValue + beta*priorDstValue
```

Note: The `dstValue` is written to after being read.

Figure 34. Scaling Parameters for Convolution



When `beta` is zero, the output is not read and may contain uninitialized data (including NaN).

These parameters are passed using a host memory pointer. The storage data types for `alpha` and `beta` are:

- `float` for HALF and FLOAT tensors, and

- `double` for DOUBLE tensors.

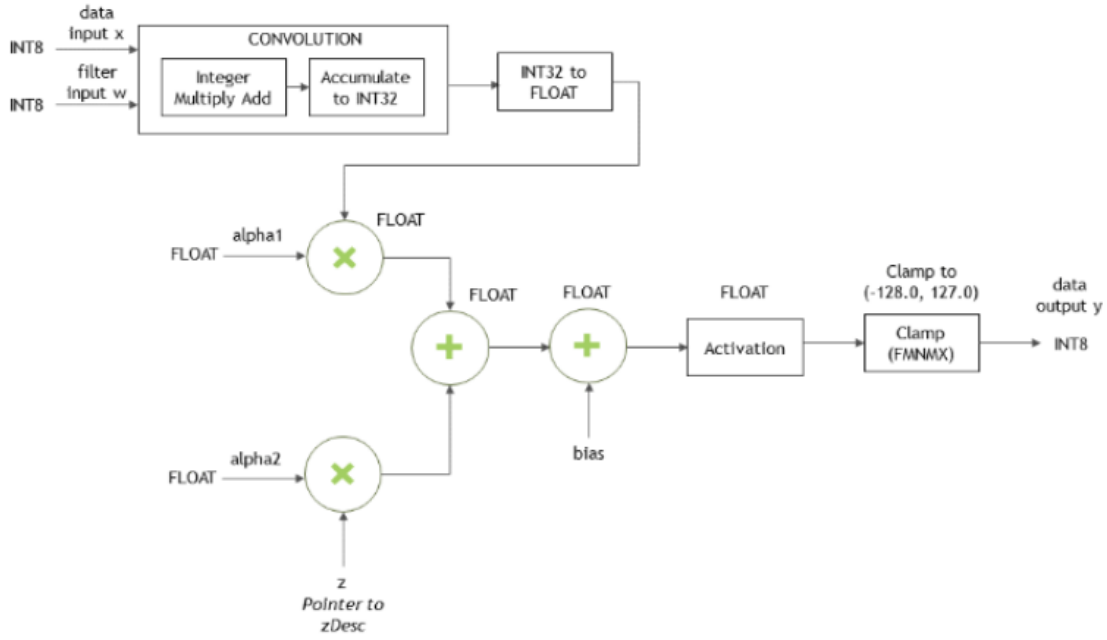
Note: For improved performance use `beta = 0.0`. Use a non-zero value for beta only when you need to blend the current output tensor values with the prior values of the output tensor.

Type Conversion

When the data input `x`, the filter input `w` and the output `y` are all in INT8 data type, the function `cudnnConvolutionBiasActivationForward()` will perform the type conversion as shown in Figure 35:

Note: Accumulators are 32-bit integers that wrap on overflow.

Figure 35. INT8 for `cudnnConvolutionBiasActivationForward`



5.4. cuDNN API Compatibility

Beginning in cuDNN 7, the binary compatibility of a patch and minor releases is maintained as follows:

- Any patch release x.y.z is forward or backward-compatible with applications built against another cuDNN patch release x.y.w (meaning, of the same major and minor version number, but having w!=z).

- cuDNN minor releases are binary backward-compatible with applications built against the same or earlier patch release (meaning, cuDNN x.y is binary compatible with an app built against cuDNN x.z, where z<=y).
- Applications compiled with a cuDNN version x.z are not guaranteed to work with x.y release when z>y.

5.5. Deprecation Policy

cuDNN version 8 introduces a new API deprecation policy to enable a faster pace of innovation.

The old deprecation policy required three major library releases to complete an API update. During this process, the original function name was first assigned to the legacy API, and then to the revised API, depending on the library version. The user wishing to migrate to the new API version had to update his or her code twice. In the first update, the original call `foo()` had to be changed to `foo_vN()`, where `N` is the new major cuDNN version. After the next major cuDNN release, the `foo_vN()` function had to be renamed back as `foo()`. Clearly, the above process could be difficult for code maintenance, especially when many functions are upgraded.

A streamlined, two-step, deprecation policy will be used for all API changes starting with cuDNN version 8. Let us explain the process using two subsequent, major cuDNN releases, version 8 and 9:

Table 20. Two-step, deprecation policy

cuDNN version	Explanation
Major release 8	The updated API is introduced as `foo_v8()`. The deprecated API `foo()` is kept unchanged to maintain backward compatibility until the next major release.
Major release 9	The deprecated API `foo()` is permanently removed and its name is not reused. The `foo_v8()` function supersedes the retired call `foo()`.

If the existing API needs to be updated, a new function flavor is introduced with the `_v` tag followed by the current, major cuDNN version. In the next major release, the deprecated function is removed, and its name is never reused. A brand-new API is first introduced without the `_v` tag.

The revised depreciation scheme allows us to retire the legacy API in just one major release. Similarly to the previous API deprecation policy, the user is able to compile the legacy code without any changes using the next major release of the cuDNN library. The backward compatibility ends when another major cuDNN release is introduced.

The updated function name embeds the information in which the cuDNN version of the API call was modified. As a result, the API changes will be easier to track and document.

The new deprecation policy is applied also to pending API changes from previous cuDNN releases. For example, according to the old deprecation policy, `cudnnSetRNNDescriptor_v6()` should be removed in cuDNN version 8 and the upgraded call `cudnnSetRNNDescriptor()` with the same arguments and behavior should be kept. Instead, the new deprecation policy is applied to this case and the tagged function is kept.

Prototypes of deprecated functions will be prepended in cuDNN version 8 headers using the `CUDNN_DEPRECATED` macro. When the `-DCUDNN_WARN_DEPRECATED` switch is passed to the compiler, any deprecated function call in the user's code will emit a compiler warning, for example:

```
warning: 'cudnnStatus_t cudnnSetRNNMatrixMathType(cudnnRNNDescriptor_t, cudnnMathType_t)' is
 deprecated [-Wdeprecated-declarations]
```

Or

```
warning C4996: 'cudnnSetRNNMatrixMathType': was declared deprecated
```

The above warnings are disabled by default to avoid potential build breaks in software setups where compiler warnings are treated as errors.

Note that the simple swapping of older cuDNN version 7 shared library files will not work with the cuDNN version 8 release. The user source code needs to be recompiled from scratch with the cuDNN version 8 headers and linked with the version 8 libraries.

5.6. GPU And Driver Requirements

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, refer to the NVIDIA cuDNN Support Matrix.

5.7. Convolutions

The convolution functions are:

- `cudnnConvolutionBackwardData()`
- `cudnnConvolutionBiasActivationForward()`
- `cudnnConvolutionForward()`
- `cudnnConvolutionBackwardBias()`
- `cudnnConvolutionBackwardFilter()`

5.7.1. Convolution Formulas

This section describes the various convolution formulas implemented in cuDNN convolution functions for the `cudnnConvolutionForward()` path.

The convolution terms described in the table below apply to all the convolution formulas that follow.

Table 21. Convolution terms

Term	Description
x	Input (image) Tensor
w	Weight Tensor
y	Output Tensor
n	Current Batch Size
c	Current Input Channel
C	Total Input Channels
H	Input Image Height
W	Input Image Width
k	Current Output Channel
K	Total Output Channels
p	Current Output Height Position
q	Current Output Width Position
G	Group Count
pad	Padding Value
u	Vertical Subsample Stride (along Height)
v	Horizontal Subsample Stride (along Width)
dil_h	Vertical Dilation (along Height)
dil_w	Horizontal Dilation (along Width)
r	Current Filter Height
R	Total Filter Height
s	Current Filter Width
S	Total Filter Width
C_g	$\frac{C}{G}$
K_g	$\frac{K}{G}$

Convolution (convolution mode set to `CUDNN_CROSS_CORRELATION`)

$$y_{n,\,k,\,p,\,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,\,c,\,p+r,\,q+s} \times w_{k,c,r,s}$$

Convolution with Padding

$$x_{<0,\,<0} = 0$$

$$x_{>H,\,>W} = 0$$

$$y_{n,\,k,\,p,\,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,\,c,\,p+r-pad,\,q+s-pad} \times w_{k,c,r,s}$$

Convolution with Subsample-Striding

$$y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,c,(p*u)+r,(q*v)+s} \times w_{k,c,r,s}$$

Convolution with Dilation

$$y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,c,p+(r*dilh),q+(s*dilw)} \times w_{k,c,r,s}$$

Convolution (convolution mode set to `CUDNN_CONVOLUTION`)

$$y_{n,k,p,q} = \sum_{c}^{C} \sum_{r}^{R} \sum_{s}^{S} x_{n,c,p+r,q+s} \times w_{k,c,R-r-1,S-s-1}$$

Convolution using Grouped Convolution

$$C_g = \frac{C}{G}$$

$$K_g = \frac{K}{G}$$

$$y_{n,k,p,q} = \sum_{c}^{Cg} \sum_{r}^{R} \sum_{s}^{S} x_{n,Cg*floor(k/Kg)+c,p+r,q+s} \times w_{k,c,r,s}$$

5.7.2. Grouped Convolutions

cuDNN supports grouped convolutions by setting `groupCount` > 1 for the convolution descriptor `convDesc`, using `cudnnSetConvolutionGroupCount()`.

Note: By default, the convolution descriptor `convDesc` is set to `groupCount` of 1.

Basic Idea

Conceptually, in grouped convolutions, the input channels and the filter channels are split into a `groupCount` number of independent groups, with each group having a reduced number of channels. The convolution operation is then performed separately on these input and filter groups.

For example, consider the following: if the number of input channels is 4, and the number of filter channels of 12. For a normal, ungrouped convolution, the number of computation operations performed are 12*4.

If the `groupCount` is set to 2, then there are now two input channel groups of two input channels each, and two filter channel groups of six filter channels each.

As a result, each grouped convolution will now perform 2*6 computation operations, and two such grouped convolutions are performed. Hence the computation savings are 2x: `(12*4)/(2*(2*6))` .

cuDNN Grouped Convolution

- When using `groupCount` for grouped convolutions, you must still define all tensor descriptors so that they describe the size of the entire convolution, instead of specifying the sizes per group.
- Grouped convolutions are supported for all formats that are currently supported by the functions `cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()` and `cudnnConvolutionBackwardFilter()`.
- The tensor stridings that are set for `groupCount` of 1 are also valid for any group count.
- By default, the convolution descriptor `convDesc` is set to `groupCount` of 1.

Note: Refer to Convolution Formulas for the math behind the cuDNN grouped convolution.

Example

Below is an example showing the dimensions and strides for grouped convolutions for NCHW format, for 2D convolution.

Note: The symbols `*` and `/` are used to indicate multiplication and division.

`xDesc` or `dxDesc`

- Dimensions: `[batch_size, input_channel, x_height, x_width]`
- Strides: `[input_channels*x_height*x_width, x_height*x_width, x_width, 1]`

`wDesc` or `dwDesc`

- Dimensions: `[output_channels, input_channels/groupCount, w_height, w_width]`
- Format: `NCHW`

`convDesc`

- Group Count: `groupCount`

`yDesc` or `dyDesc`

- Dimensions: `[batch_size, output_channels, y_height, y_width]`
- Strides: `[output_channels*y_height*y_width, y_height*y_width, y_width, 1]`

5.7.3. Best Practices for 3D Convolutions

> **!** ATTENTION: These guidelines are applicable to 3D convolution and deconvolution functions starting in cuDNN v7.6.3.

The following guidelines are for setting the cuDNN library parameters to enhance the performance of 3D convolutions. Specifically, these guidelines are focused on settings such as filter sizes, padding and dilation settings. Additionally, an application-specific use-case, namely, medical imaging, is presented to demonstrate the performance enhancement of 3D convolutions with these recommended settings.

Specifically, these guidelines are applicable to the following functions and their associated data types:

- `cudnnConvolutionForward()`
- `cudnnConvolutionBackwardData()`
- `cudnnConvolutionBackwardFilter()`

For more information, refer to the NVIDIA cuDNN API Reference.

5.7.3.1. Recommended Settings

The following table shows the recommended settings while performing 3D convolutions for cuDNN.

Table 22. Recommended settings while performing 3D convolutions for cuDNN

	cuDNN 8.8.0
Platform	NVIDIA Hopper architecture NVIDIA Ampere architecture NVIDIA Turing architecture NVIDIA Volta architecture
Convolution (3D or 2D)	3D and 2D

Convolution or deconvolution (`fprop`, `dgrad`, or `wgrad`)	`fprop`
	`dgrad`
	`wgrad`

Grouped convolution size	`C_per_group == K_per_group == {1,4,8,16,32,64,128,256}` Not supported for INT8

		cuDNN 8.8.0
Data layout format (NHWC/NCHW)[12]		NDHWC
Input/output precision (FP16, FP32, INT8, or FP64)		FP16, FP32[13], INT8[14]
Accumulator (compute) precision (FP16, FP32, INT32 or FP64)		FP32, INT32
Filter (kernel) sizes		No limitation
Padding		No limitation
Image sizes		2 GB limitation for a tensor
Number of channels	C	`0 mod 8` `0 mod 16` (for INT8)
	K	`0 mod 8` `0 mod 16` (for INT8)
Convolution mode		Cross-correlation and convolution
Strides		No limitation
Dilation		No limitation
Data pointer alignment		All data pointers are 16-bytes aligned.

5.7.3.2. Limitations

Your application will be functional but could be less performant if the model has channel counts lower than 32 (gets worse the lower it is).

If the above is in the network, use `cuDNNFind` to get the best option.

[12] NHWC/NCHW corresponds to NDHWC/NCDHW in 3D convolution.

[13] With CUDNN_TENSOROP_MATH_ALLOW_CONVERSION pre-Ampere. Default TF32 math in NVIDIA Ampere architecture.

[14] INT8 does not support dgrad and wgrad. INT8 3D convolutions are only supported in the backend API. Refer to the tables in cudnnConvolutionForward() for more information.

Chapter 6. Troubleshooting

The following sections help answer the most commonly asked questions regarding typical use cases.

6.1. Error Reporting And API Logging

The cuDNN error reporting and API logging is a utility for recording the cuDNN API execution and error information. For each cuDNN API function call, all input parameters are reported in the API logging. If errors occur during the execution of the cuDNN API, a traceback of the error conditions can also be reported to help troubleshooting. This functionality is disabled by default, and can be enabled using the methods described in the later part of this section through three logging severity levels: `CUDNN_LOGINFO_DBG`, `CUDNN_LOGWARN_DBG` and `CUDNN_LOGERR_DBG`.

The log output contains variable names, data types, parameter values, device pointers, process ID, thread ID, cuDNN handle, CUDA stream ID, and metadata such as time of the function call in microseconds.

For example, when the severity level `CUDNN_LOGINFO_DBG` is enabled, the user will receive the API loggings, such as:

```
cuDNN (v8300) function cudnnSetActivationDescriptor() called:
    mode: type=cudnnActivationMode_t; val=CUDNN_ACTIVATION_RELU (1);
    reluNanOpt: type=cudnnNanPropagation_t; val=CUDNN_NOT_PROPAGATE_NAN (0);
    coef: type=double; val=1000.000000;
Time: 2017-11-21T14:14:21.366171 (0d+0h+1m+5s since start)
Process: 21264, Thread: 21264, cudnn_handle: NULL, cudnn_stream: NULL.
```

Starting in cuDNN 8.3.0, when the severity level `CUDNN_LOGWARN_DBG` or `CUDNN_LOGERR_DBG` are enabled, the log output additionally reports an error traceback such as the example below (currently only cuDNN version 8 graph APIs and legacy convolution APIs are using this error reporting feature). This traceback reports the relevant error/warning conditions, aiming to provide the user hints for troubleshooting purposes. Within the traceback, each message may have their own severity and will only be reported when the respective severity level is enabled. The traceback messages are printed in the reverse order of the execution so the messages at the top will be the root cause and tend to be more helpful for debugging.

```
cuDNN (v8300) function cudnnBackendFinalize() called:
    Info: Traceback contains 5 message(s)
        Error: CUDNN_STATUS_BAD_PARAM; reason: out <= 0
        Error: CUDNN_STATUS_BAD_PARAM; reason: is_valid_spacial_dim(xSpatialDimA[dim],
 wSpatialDimA[dim], ySpatialDimA[dim], cDesc.getPadLowerA()[dim], cDesc.getPadUpperA()[dim],
 cDesc.getStrideA()[dim], cDesc.getDilationA()[dim])
```

```
        Error: CUDNN_STATUS_BAD_PARAM; reason: is_valid_convolution(xDesc, wDesc, cDesc,
 yDesc)
        Error: CUDNN_STATUS_BAD_PARAM; reason: convolution.init(xDesc, wDesc, cDesc, yDesc)
        Error: CUDNN_STATUS_BAD_PARAM; reason: finalize_internal()
Time: 2021-10-05T17:11:07.935640 (0d+0h+0m+15s since start)
Process=87720; Thread=87720; GPU=NULL; Handle=NULL; StreamId=NULL.
```

There are two methods, as described below, to enable the error/warning reporting and API logging. For convenience, the log output can be handled by the built-in default callback function, which will direct the output to a log file or the standard I/O as designated by the user. The user may also write their own callback function to handle this information programmably, and use the `cudnnSetCallback()` to pass in the function pointer of their own callback function.

Method 1: Using Environment Variables

To enable API logging using environment variables, follow these steps:

- Decide which logging severity levels to include from these three options: `CUDNN_LOGINFO_DBG`, `CUDNN_LOGWARN_DBG`, or `CUDNN_LOGERR_DBG`. The logging severity levels are independent of each other. Any combination of them is valid.
- Set the environment variables `CUDNN_LOGINFO_DBG`, `CUDNN_LOGWARN_DBG`, or `CUDNN_LOGERR_DBG` to `1`, and
- Set the environment variable `CUDNN_LOGDEST_DBG` to one of the following:
  - `stdout`, `stderr`, or a user-desired file path, for example, `/home/userName1/log.txt`.
- Include the conversion specifiers in the file name. For example:
  - To include date and time in the file name, use the date and time conversion specifiers: `log_%Y_%m_%d_%H_%M_%S.txt`. The conversion specifiers will be automatically replaced with the date and time when the program is initiated, resulting in `log_2017_11_21_09_41_00.txt`.
  - To include the process id in the file name, use the `%i` conversion specifier: `log_%Y_%m_%d_%H_%M_%S_%i.txt` for the result: `log_2017_11_21_09_41_00_21264.txt` when the process id is `21264`. When you have several processes running, using the process id conversion specifier will prevent these processes from writing to the same file at the same time.

Note: The supported conversion specifiers are similar to the `strftime` function.

If the file already exists, the log will overwrite the existing file.

Note: These environmental variables are only checked once at the initialization. Any subsequent changes in these environmental variables will not be effective in the current run. Also note that these environment settings can be overridden by Method 2 below.

Refer to Table 23 for the impact on the performance of API logging using environment variables. The `CUDNN_LOG{INFO,WARN,ERR}_DBG` notation in the table header means the conclusion is applicable to either one of the environment variables.

Table 23. API Logging Using Environment Variables

Environment variables	`CUDNN_LOG{INFO,WARN,ERR}_`	`CUDNN_LOG{INFO,WARN,ERR}_DBG=1`
`CUDNN_LOGDEST_DBG` not set	No logging output No performance loss	No logging output No performance loss
`CUDNN_LOGDEST_DBG=NULL`	No logging output No performance loss	No logging output No performance loss
`CUDNN_LOGDEST_DBG=stdout` or `stderr`	No logging output No performance loss	Logging to `stdout` or `stderr` Some performance loss
`CUDNN_LOGDEST_DBG=filename.txt`	No logging output No performance loss	Logging to `filename.txt` Some performance loss

Method 2: Using the API

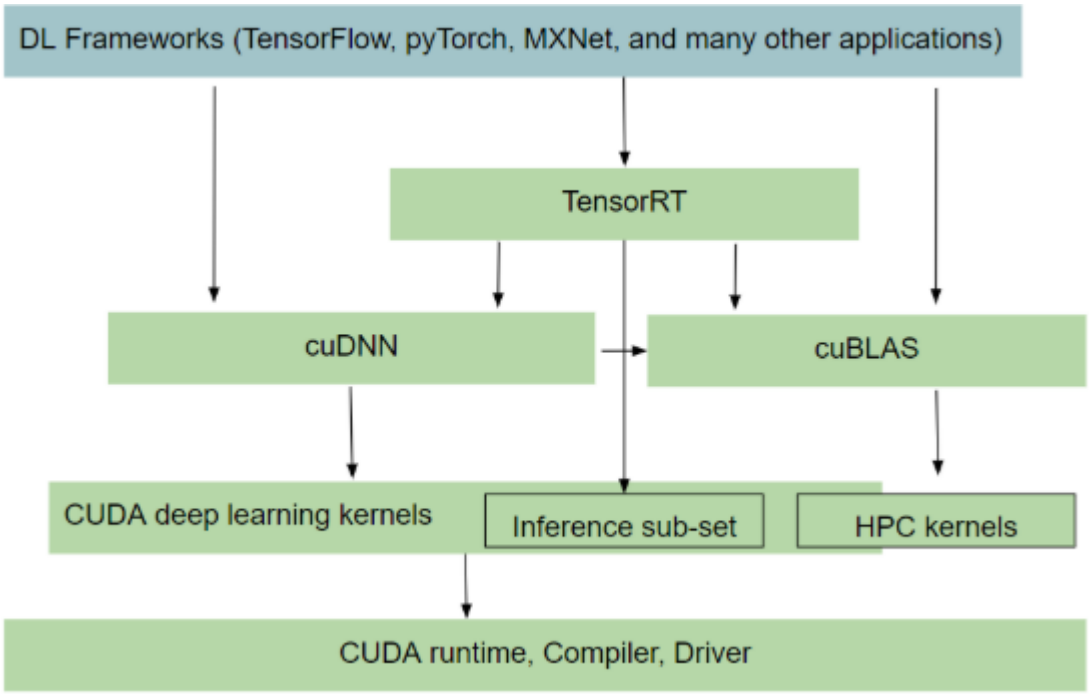
To use API function calls to enable API logging, refer to the API description of `cudnnSetCallback()` and `cudnnGetCallback()`.

6.2. FAQs

Q: Where in the software stack does cuDNN sit? What is the interaction between CUDA, cuDNN, and TensorRT?

A: The following graphic shows how cuDNN relates to other software in the stack.

Figure 36. Software Stack With cuDNN



Q: I'm not sure if I should use cuDNN for inference or training. How does it compare with TensorRT?

A: cuDNN provides the building blocks for common routines such as convolution, pooling, activation and RNN/LSTMs. You can use cuDNN for both training and inference. However, where it differs from TensorRT is that the latter (TensorRT) is a programmable inference accelerator; just like a framework. TensorRT sees the whole graph and optimizes the network by fusing/combining layers and optimizing kernel selection for improved latency, throughout, power efficiency and for reducing memory requirements.

A rule of thumb you can apply is to check out TensorRT, see if it meets your inference needs, if it doesn't, then look at cuDNN for a closer, more in-depth perspective.

Q: How does heuristics in cuDNN work? How does it know what is the optimal solution for a given problem?

A: NVIDIA actively monitors the Deep Learning space for important problem specifications such as commonly used models. The heuristics are produced by sampling a portion of these problem specifications with available computational choices. Over time, more models are discovered and incorporated into the heuristics.

Q: Is cuDNN going to support running arbitrary graphs?

A: No, we don't plan to become a framework and execute the whole graph one op at a time. At this time, we are focused on a subgraph given by the user, where we try to

produce an optimized fusion kernel. We will document the rules regarding what can be fused and what cannot. The goal is to support general and flexible fusion, however, it will take time and there will be limits in what it can do in the cuDNN version 8.0.0 launch.

Q: What's the difference between TensorRT, TensorFlow/XLA's fusion, and cuDNN's fusion?

A: TensorRT and TensorFlow are frameworks; they see the whole graph and can do global optimization, however, they generally only fuse pointwise ops together or pattern match to a limited set of pre-compiled fixed fusion patterns like conv-bias-relu. On the other hand, cuDNN targets a subgraph, but can fuse convolutions with pointwise ops, thus providing potentially better performance. CuDNN fusion kernels can be utilized by TensorRT and TensorFlow/XLA as part of their global graph optimization.

Q: Can I write an application calling cuDNN directly?

A: Yes, you can call the C/C++ API directly. Usually, data scientists would wait for framework integration and use the Python API which is more convenient. However, if your use case requires better performance, you can target the cuDNN API directly.

Q: How does mixed precision training work?

A: Several components need to work together to make mixed precision training possible. CuDNN needs to support the layers with the required datatype config and have optimized kernels that run very fast. In addition, there is a module called automatic mixed precision (AMP) in frameworks which intelligently decides which op can run in a lower precision without affecting convergence and minimize the number of type conversions/transposes in the entire graph. These work together to give you speed up. For more information, refer to Mixed Precision Numerical Accuracy.

Q: How can I pick the fastest convolution kernels with cuDNN version 8.0.0?

A: In the API introduced in cuDNN v8, convolution kernels are grouped by similar computation and numerical properties into engines. Every engine has a queryable set of performance tuning knobs. A computation case such as a convolution operation graph can be computed using different valid combinations of engines and their knobs, known as an engine configuration. Users can query an array of engine configurations for any given computation case ordered by performance, from fastest to slowest according to cuDNN's own heuristics. Alternately, users can generate all possible engine configurations by querying the engine count and available knobs for each engine. This generated list could be used for auto-tuning or the user could create their own heuristics.

Q: Why is cuDNN version 8.0 convolution API call much slower on the first call than subsequent calls?

A: Due to the library split, cuDNN version 8.0 API will only load the necessary kernels on the first API call that requires it. In previous versions, this load would have been observed in the first cuDNN API call that triggers CUDA context initialization, typically `cudnnCreate()`. In version 8.0, this is delayed until the first sub-library call that triggers CUDA context initialization. Users who desire to have CUDA context preloaded can call the new `cudnnCnnInferVersionCheck()` API (or its related cousins), which has the side effect of initializing a CUDA context. This will reduce the run time for all subsequent API calls.

Q: How do I build the cuDNN version 8.0.0 split library?

A: cuDNN v8.0 library is split into multiple sub-libraries. Each library contains a subset of the API. Users can link directly against the individual libraries or link with a `dlopen` layer which follows a plugin architecture.

To link against an individual library, users can directly specify it and its dependencies on the linker command line. For example, for infer libraries: `-lcudnn_adv_infer`, `-lcudnn_cnn_infer`, or `-lcudnn_ops_infer`.

For all libraries, `-lcudnn_adv_train`, `-lcudnn_cnn_train`, `-lcudnn_ops_train`, `-lcudnn_adv_infer`, `-lcudnn_cnn_infer`, and `-lcudnn_ops_infer`.

The dependency order is documented in the NVIDIA cuDNN 8.0.0 Preview Release Notes and the NVIDIA cuDNN API Reference.

Alternatively, the user can continue to link against a shim layer (`-libcudnn`) which can `dlopen` the correct library that provides the implementation of the function. When the function is called for the first time, the dynamic loading of the library takes place.

Linker argument:

```
-lcudnn
```

Q: What are the new APIs in cuDNN version 8.0.0?

A: The new cuDNN APIs are listed in the cuDNN 8.0.0 Release Notes as well as in the API changes for cuDNN 8.0.0.

6.3. Support

Support, resources, and information about cuDNN can be found online at https://developer.nvidia.com/cudnn. This includes downloads, webinars, NVIDIA Developer Forums, and more.

We appreciate all types of feedback. Consider posting on the forums with questions, comments, and suspected bugs that are appropriate to discuss publicly. cuDNN-related posts are reviewed by the cuDNN engineering team, and internally we will file bugs where appropriate. It's helpful if you can paste or attach an API log to help us reproduce.

External users can also file bugs directly by following these steps:

1. Register for the NVIDIA Developer website.
2. Log in to the developer site.
3. Click on your name in the upper right corner.
4. Click My account > My Bugs and select Submit a New Bug.
5. Fill out the bug reporting page. Be descriptive and if possible, provide the steps that you are following to help reproduce the problem. If possible, paste or attach an API log.
6. Click Submit a bug.

Chapter 7. Acknowledgments

Some of the cuDNN library routines were derived from code developed by others and are subject to the following:

7.1. University of Tennessee

7.2. University of California, Berkeley

7.3. Facebook AI Research, New York

Notice

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

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

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

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

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

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

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

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

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

Arm

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

HDMI

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

Blackberry/QNX

Copyright © 2020 BlackBerry Limited. All rights reserved.

Trademarks, including but not limited to BLACKBERRY, EMBLEM Design, QNX, AVIAGE, MOMENTICS, NEUTRINO and QNX CAR are the trademarks or registered trademarks of BlackBerry Limited, used under license, and the exclusive rights to such trademarks are expressly reserved.

Google

Android, Android TV, Google Play and the Google Play logo are trademarks of Google, Inc.

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

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