NVIDIA cuDNN

Developer Guide | NVIDIA Docs

PG-06702-001_v8.4.0 | April 2022

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

NVIDIA® CUDA® Deep Neural Network LIbrary (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines 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
- Tensor transformation functions
- LRN, LCN and batch normalization forward and backward

cuDNN convolution routines aim for a performance that is competitive with the fastest GEMM (matrix multiply)-based implementations of such routines while using significantly less memory.

cuDNN features include customizable data layouts, supporting flexible dimension ordering, striding, and subregions for the 4D tensors used as inputs and outputs to all of its routines. This flexibility allows easy integration into any neural network implementation and avoids the input/output transposition steps sometimes necessary with GEMM-based convolutions.

cuDNN offers a context-based API that allows for easy multithreading and (optional) interoperability with NVIDIA® CUDA® streams.

1.1. Programming Model

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 to associate different devices with different host threads, and in each of those host threads, use a unique cuDNN handle that directs the library calls to the device associated with it. Thus the cuDNN library calls made with different handles will 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()`.

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 beginning with cuDNN 7 are binary backward-compatible with applications built against the same or earlier patch release (meaning, an application built against cuDNN 7.x is binary compatible with cuDNN library 7.y, where y>=x).
- Applications compiled with a cuDNN version 7.y are not guaranteed to work with 7.x release when y > x.

1.2. GPU And Driver Requirements

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

1.3. Backward Compatibility And 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 1. 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.

1.4. 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.

Chapter 2. Tensor Descriptor

The cuDNN library describes data holding images, videos and any other data with contents with a generic n-D tensor defined with the following parameters:

- a dimension `nbDims` from 3 to 8
- a data type (32-bit floating-point, 64 bit-floating point, 16-bit floating-point...)
- `dimA` integer array defining the size of each dimension
- `strideA` 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)

The first dimension of the tensor defines the batch size `n`, and the second dimension defines the number of features maps `c`. 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.

2.1. 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. 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

2.3. 5-D Tensor Description

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

2.4. 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.

2.5. 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.

2.6. 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.

2.7. 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.

Chapter 3. Data Layout Formats

This section describes how cuDNN tensors are arranged in memory. See cudnnTensorFormat_t for enumerated tensor format types.

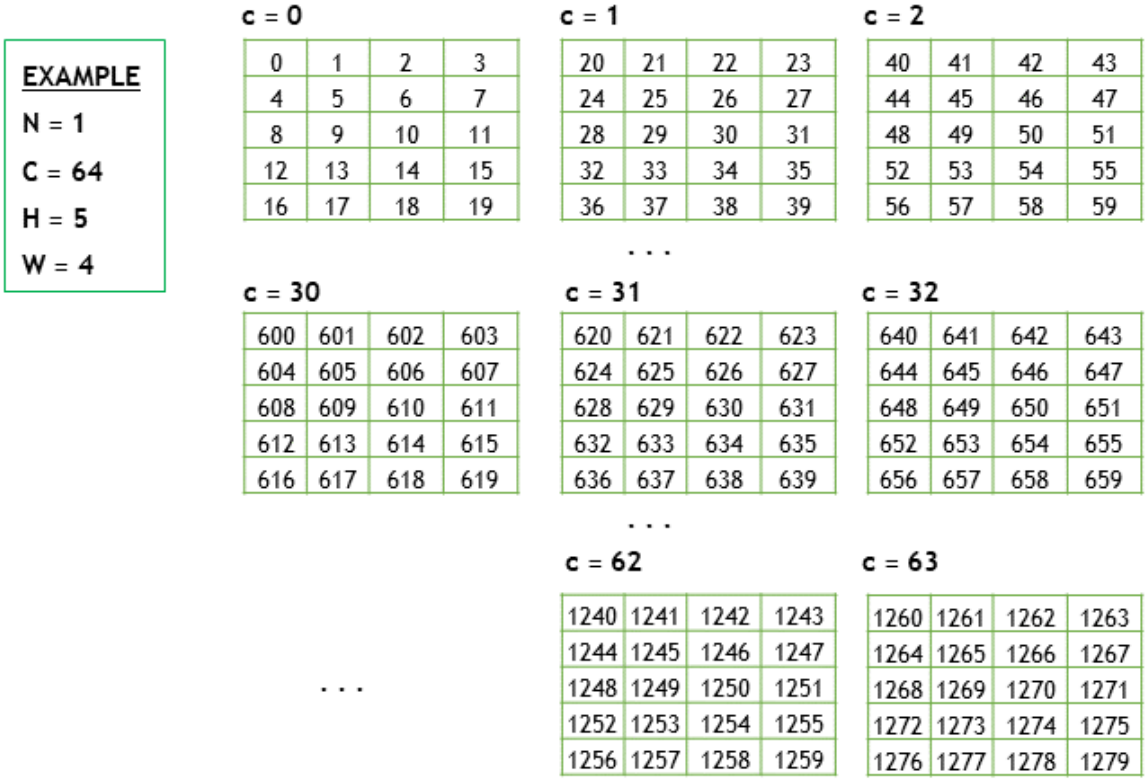
3.1. Data Layout Example

Consider a batch of images in 4D with the following dimensions:

- **N** is the batch size; 1.
- **C** is the number of feature maps (i.e., 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. See Figure 1.

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

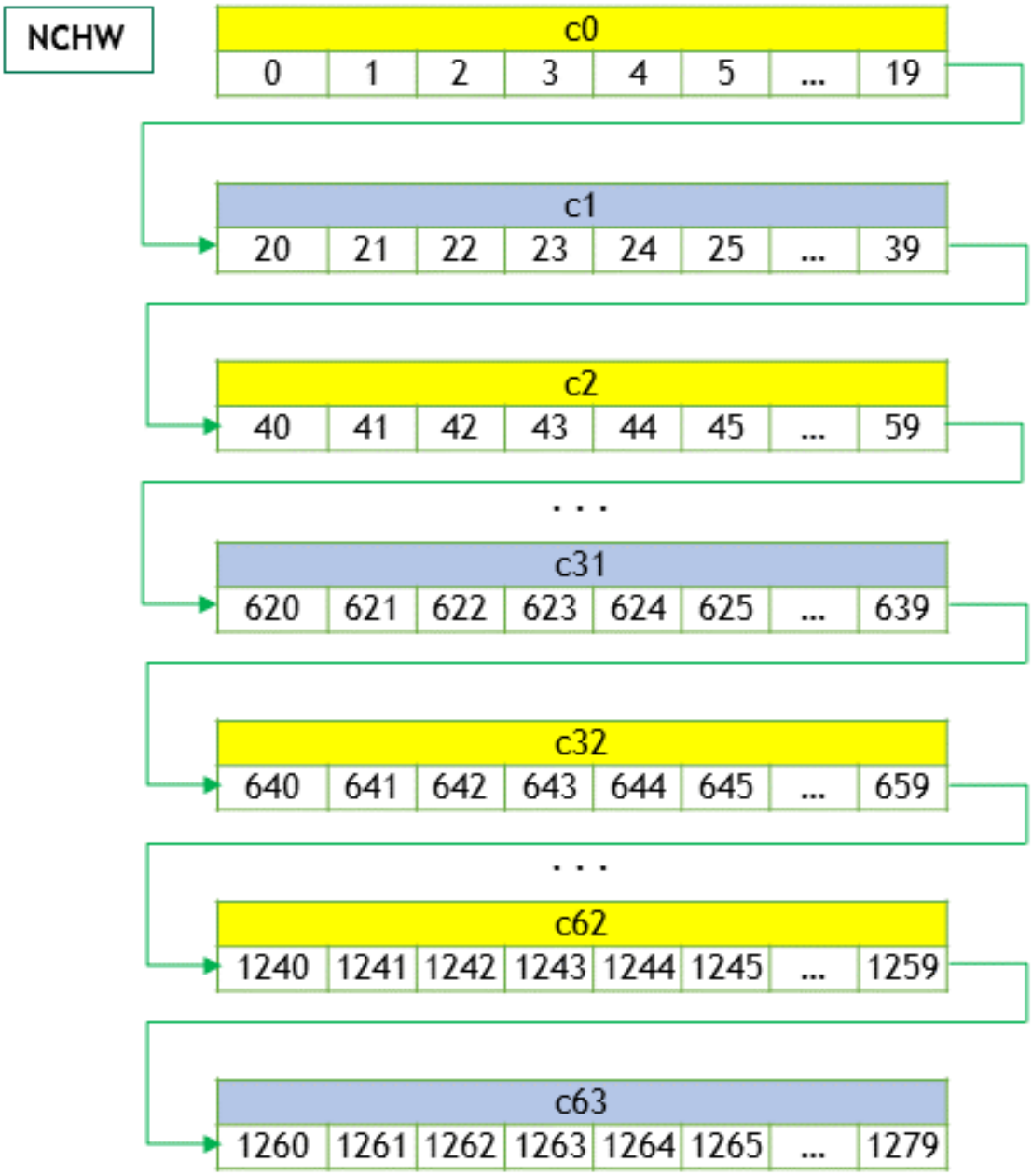


3.2. NCHW Memory Layout

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. See Figure 2.
3. Proceed to the next batch (if **N** is > 1).

Figure 2. NCHW Memory Layout

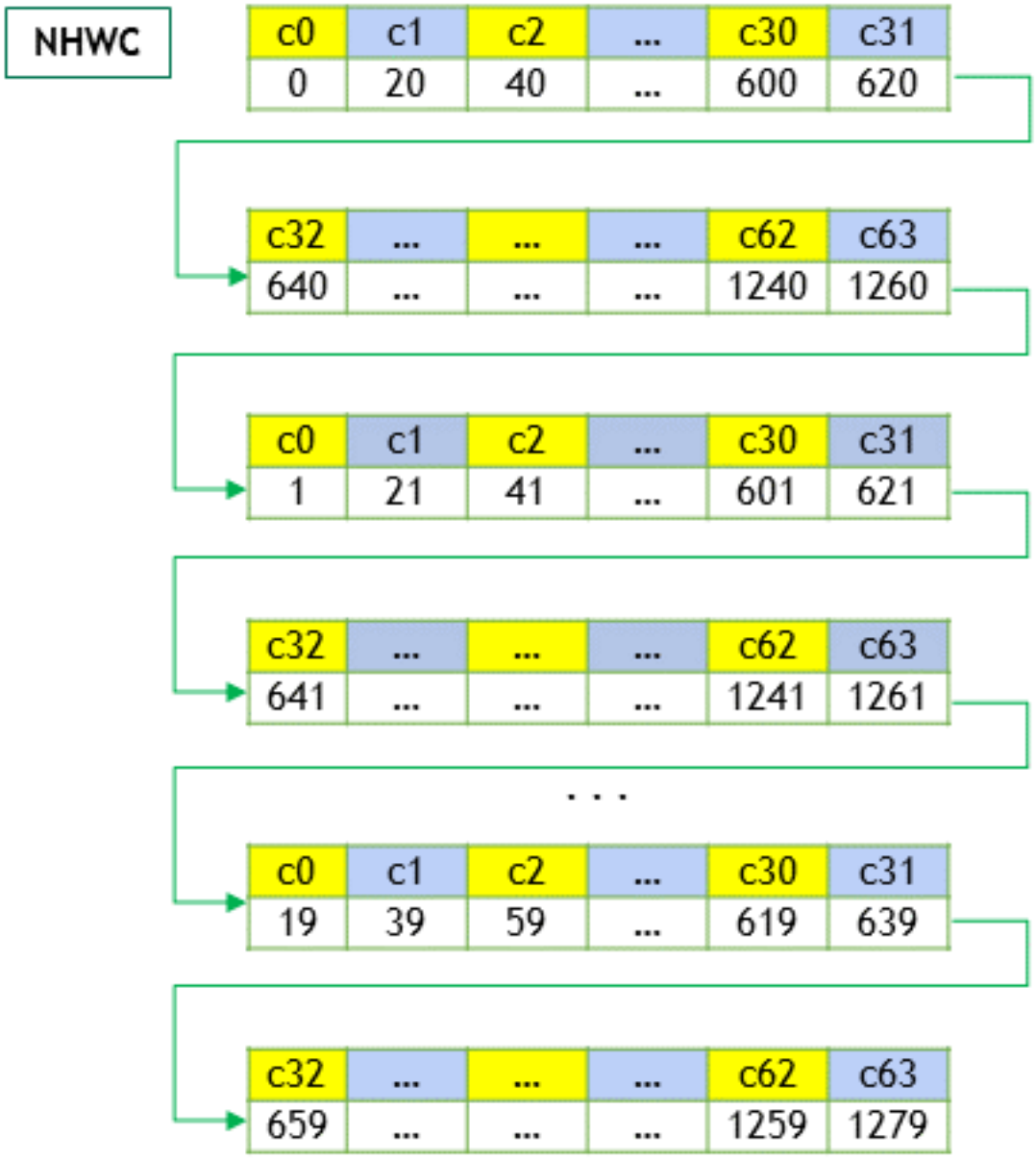


3.3. NHWC Memory Layout

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. See Figure 3.
4. Proceed to the next batch (if **N** is > 1).

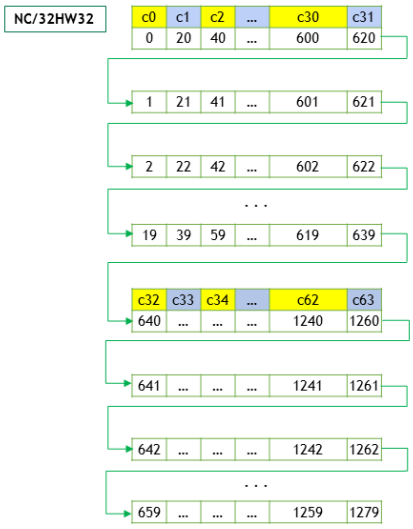
Figure 3. NHWC Memory Layout



3.4. 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. See 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 (i.e, one group) are arranged contiguously (in the row-major order), before proceeding to the next group (i.e., next channel). This is the same as NCHW format.
- When `x = C`, then NC/xHWx is identical to NHWC, i.e., 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 CUDNN_TENSOR_NCHW_VECT_C 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.

Chapter 4. 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™, or NVIDIA Turing and NVIDIA Ampere Architecture.

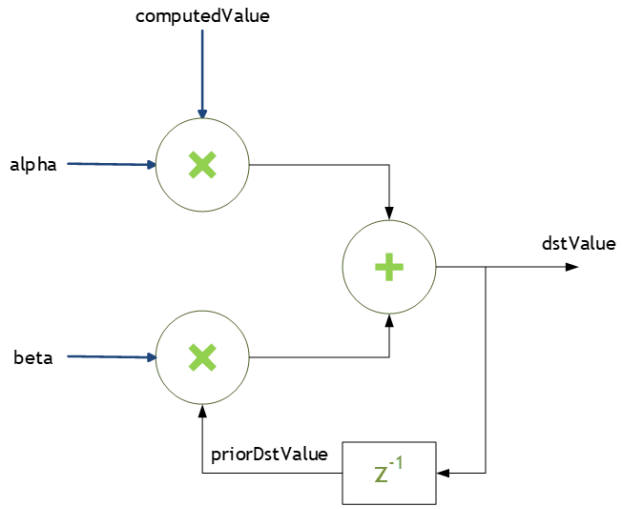
Chapter 5. Scaling Parameters

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 (see Figure 5):

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

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

Figure 5. 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.

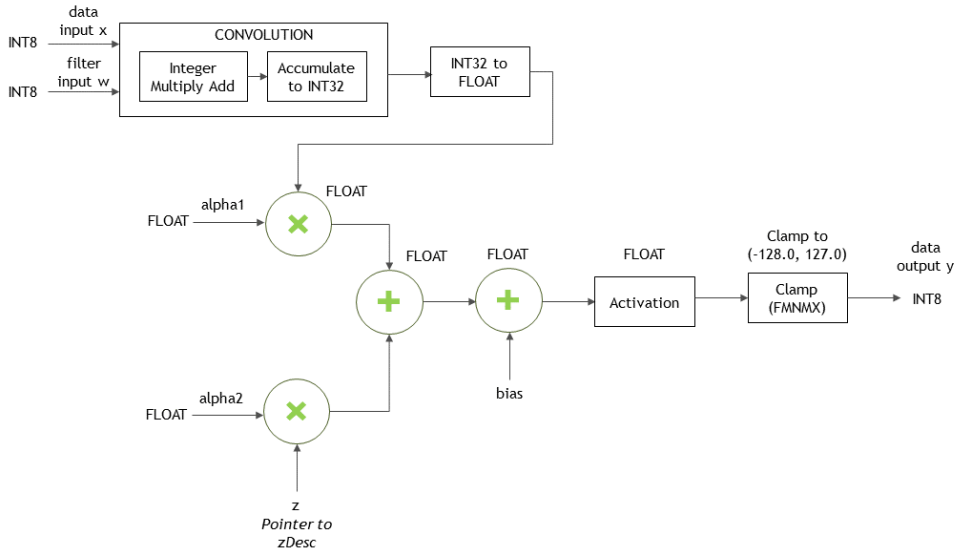
Note: For improved performance use `beta = 0.0`. Use a non-zero value for beta only when you need to blend the current output tensor values with the prior values of the output tensor.

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 6:

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

Figure 6. INT8 for cudnnConvolutionBiasActivationForward



Chapter 6. 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.

6.1. Basics

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

See also Training with Mixed Precision.

6.2. Convolution Functions

6.2.1. 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`.

6.2.2. Supported Algorithms

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

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

See the table below for 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`

6.2.3. Data And Filter Formats

The cuDNN library may use padding, folding, and NCHW-to-NHWC transformations to call the Tensor Core operations. See 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` (i.e., FP16). For FP32 `dataType` see FP32-to-FP16 Conversion.
- The number of input and output feature maps (i.e., channel dimension `C`) is a multiple of 8. When the channel dimension is not a multiple of 8, see 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.

6.3. RNN Functions

6.3.1. 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`.

6.3.2. Supported Algorithms

When the above prerequisite is 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()

See the table below for the supported algorithms:

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

6.3.3. 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.

See also Features Of RNN Functions.

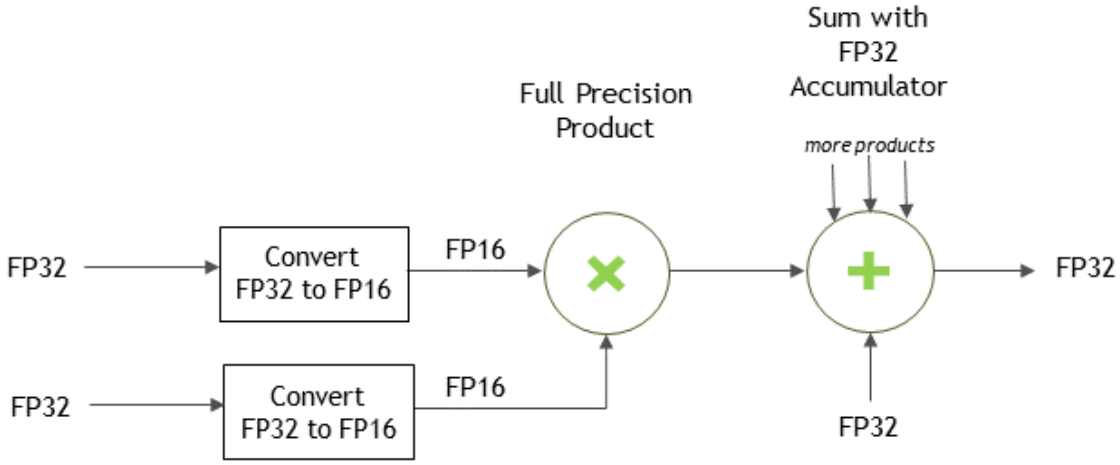
6.4. Tensor Transformations

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. See below.

6.4.1. FP32-to-FP16 Conversion

The cuDNN API allows the user 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. See Figure 7.

Figure 7. 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));
```

6.4.2. 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.

6.4.3. 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.

6.4.4. 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, as stated in Basics, 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.

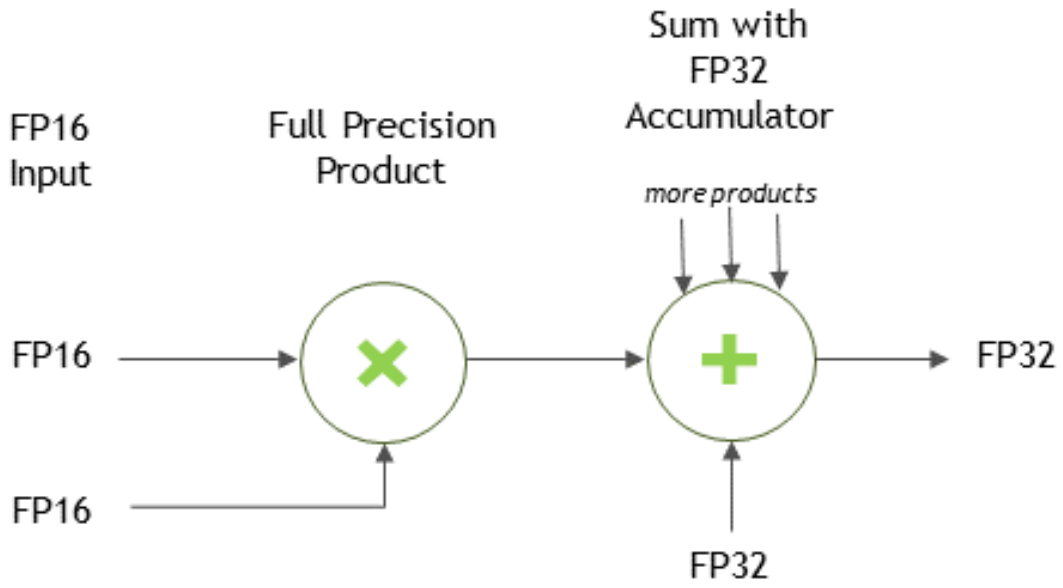
In very rare and difficult-to-qualify cases that are a complex function of padding and filter sizes, it is possible that Tensor Ops is not enabled. In such cases, users can pre-pad to enable the Tensor Ops path.

6.5.1. FP16 Data

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. See Figure 8.

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 8. 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.



6.6. Guidelines For Good Performance On Tensor Cores

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.

Chapter 7. Convolutions

The convolution functions are:

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

7.1. Convolution Formulas

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

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

Table 2. 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

Term	Description
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

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

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

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

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: See 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]`

7.3. Best Practices For 3D Convolutions

ATTENTION: These guidelines are applicable to 3D convolution and deconvolution functions starting in NVIDIA® CUDA® Deep Neural Network library (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 Developer Guide* and the *NVIDIA cuDNN API Reference*.

7.3.1. Recommended Settings

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

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

		cuDNN 8.4.0
Platform		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
Data layout format (NHWC/NCHW)[1]		NDHWC
Input/output precision (FP16, FP32, INT8, or FP64)		FP16, FP32[2], INT8[3]
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)

[1] NHWC/NCHW corresponds to NDHWC/NCDHW in 3D convolution.
[2] With CUDNN_TENSOROP_MATH_ALLOW_CONVERSION pre-Ampere. Default TF32 math in NVIDIA Ampere Architecture.
[3] INT8 does not support `dgrad` and `wgrad`.

	cuDNN 8.4.0
Convolution mode	Cross-correlation and convolution
Strides	No limitation
Dilation	No limitation
Data pointer alignment	All data pointers are 16-bytes aligned.

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.

Chapter 8. Features Of RNN Functions

Refer to the table below 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	Input/output 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[4] (see the table below for an elaboration 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 do down

`cudnnRNNForwardInferenceEx()`
`cudnnRNNForwardTrainingEx()`
`cudnnRNNBackwardDataEx()`
`cudnnRNNBackwardWeightsEx()`
	Sequence major unpacked
	Batch major unpacked[5]
	Sequence major packed[6]
		Only with `_ALGO_STANDARD`
		For unpacked layout, no input sorting required. [7]
		For packed layout, require input sequences descending sorted according to length.	

[4] 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.

[5] To use an unpacked layout, users need to set `CUDNN_RNN_PADDED_IO_ENABLED` through `cudnnSetRNNPaddingMode()`.

[7] To use an unpacked layout, users need to set `CUDNN_RNN_PADDED_IO_ENABLED` through `cudnnSetRNNPaddingMode()`.

[6] To use an unpacked layout, users need to set `CUDNN_RNN_PADDED_IO_ENABLED` through `cudnnSetRNNPaddingMode()`.

Functions	Input/output layout supported	Supports variable sequence length in batch	Commonly supported
			conversion to utilize Tensor Cores) Direction mode supported: `CUDNN_UNIDIRECTIONAL`, `CUDNN_BIDIRECTIONAL` RNN input mode: `CUDNN_LINEAR_INPUT`, `CUDNN_SKIP_INPUT`

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_DYNAMIC`
Half input Single accumulation Half output	Supported Half intermediate storage Single accumulation			
Single input Single accumulation Single output	Supported If running on Volta, with `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`*, 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			Not Supported, will execute normally ignoring `CUDNN_TENSOR_OP_MATH`[11]

[11] `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_TENSOR_OP_MATH[8] or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION [9] 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[10] Acceleration requires inputSize and hiddenSize to be a multiple of 8 For single input/output on NVIDIA Ampere Architecture, acceleration requires setting CUDNN_DEFAULT_MATH,CUDNN_TENSOR_OP_MATH,Or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION* Acceleration requires inputSize and hiddenSize to be a multiple of 4			or _ALLOW_CONVERSION[12]
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

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

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

[12] 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().

Chapter 9. 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.

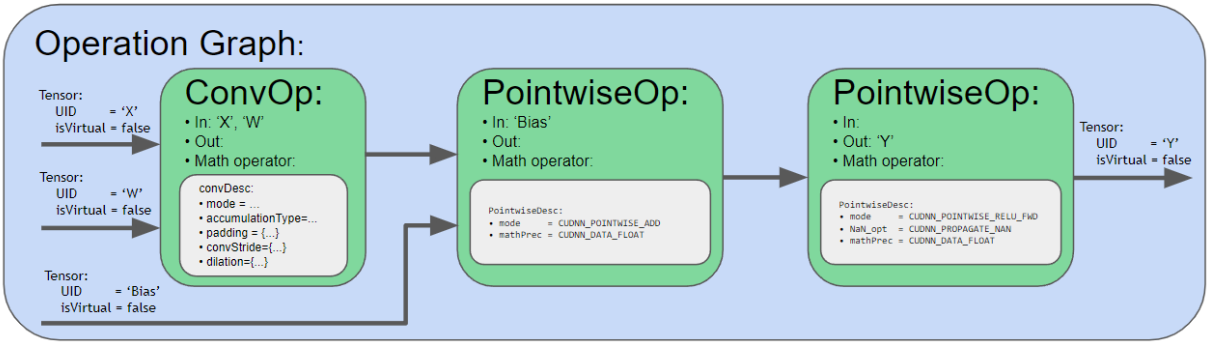
For example, when the computation is performed in FP32 and the output is in FP16, the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` (`ALGO_0`) has lower accuracy compared to the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` (`ALGO_1`). This is because `ALGO_0` does not use extra workspace, and is forced to accumulate the intermediate results in FP16, i.e., 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, i.e., full precision float.

Chapter 10. The cuDNN 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.

The user starts by building a graph of operations, like the one pictured below:

Figure 9. Example operation graph for convolution > bias-relu



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 next subsection walks through an example operation graph, covering the process in more detail.

The graph API has two entry points:

- C backend API (lowest level entry point into the graph API)
- C++ frontend API (convenience layer on top of the C backend API)

We expect that most users prefer the C++ 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 (i.e. the backend or frontend API), the high level concepts are the same.

10.1. Graph API Example with Operation Fusion

In the following example, the user would like to implement a fusion operation of convolution, bias, and activation.

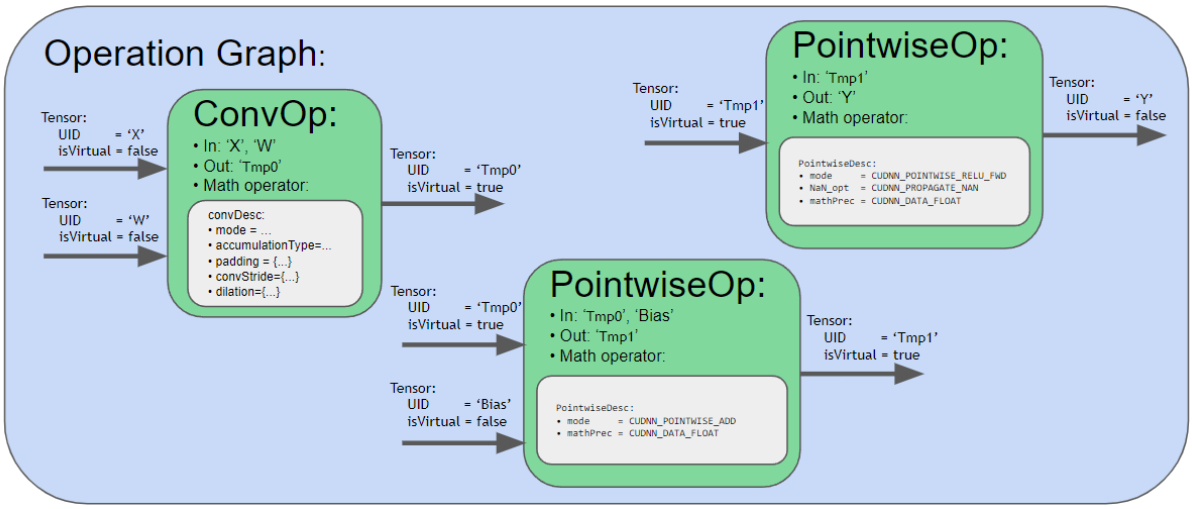
10.1.1. Creating Operation and Tensor Descriptors to Specify the Graph Dataflow

First, the user should create three cuDNN backend operation descriptors - one convolution operation descriptor and two pointwise operation descriptors (one for the bias and one for the activation).

The user 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, 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 (i.e. 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.

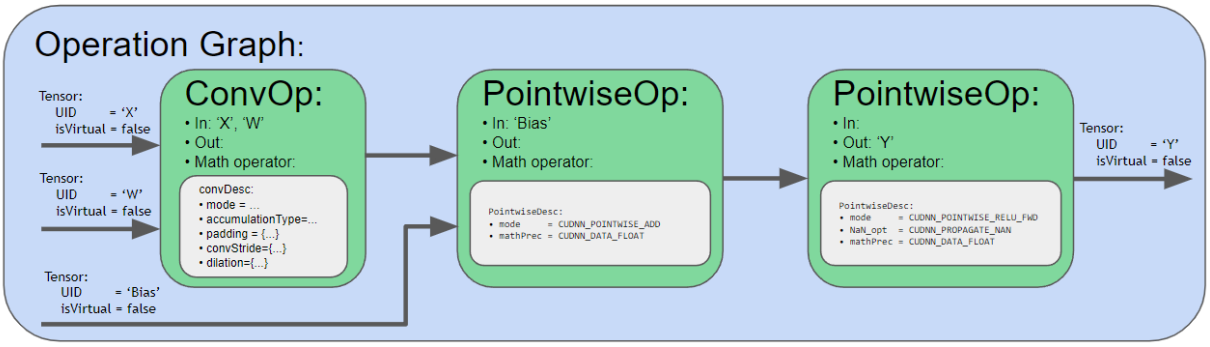
Figure 10. A set of operation descriptors the user passes to the operation graph



10.1.2. 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 11. The operation graph after finalization



10.1.3. 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. There are three methods:

1. **Heuristics.** Users that prefer cuDNN to recommend the best engine and knob choices can query cuDNN's heuristics to get a list of engine configs, sorted by predicted performance.

Typically, the user constructs the execution plan using the top ranked engine config in the list.

2. **Auto-tuning.** The user can also iterate over a list of engine configs and time each one to choose the best engine config for a particular problem on a particular device. The C ++ frontend API provides a convenience function, `cudnnFindPlan`, which does this. To reduce overhead, a user might only auto-tune over the top N engine configs returned by the heuristics.
3. **Manual.** Expert users can query for all engines that can support the operation graph. For each engine, the user can then further query the numerical notes and adjustable knobs. Numerical notes inform the user about the numerical behavior of the engine such as whether it does datatype down conversion at the input or during output reduction. The adjustable knobs allow fine grained control of the engine's behavior and performance. With the engine choice and the knob choice determined, the user can construct the engine, engine config, and execution plan.

There are two different mechanisms in cuDNN that support the graph API.

1. There are engines containing offline compiled kernels that can support certain fusion patterns. These engines try to match the user provided operation graph with their supported fusion pattern. If there is a match, then that particular engine is deemed suitable for this use case.
2. There are also runtime fusion engines. Instead of passively matching the user graph, such engines actively walk the graph and assemble code blocks to form a CUDA kernel and compile on the fly. Such runtime fusion engines are much more flexible in range of support. However, because the construction of the execution plans requires runtime compilation, the one-time CPU overhead is higher than the other engines.

10.1.4. 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.

10.2. Supported Graph Patterns

The table below briefly summarizes some example graph patterns that are currently supported by cuDNN with fused implementations. Other patterns are supported as well, but not documented here yet. We will add additional support in the upcoming releases. We welcome feature suggestions. For feedback, email cudnn@nvidia.com.

Table 4. Supported graph patterns

Fusion Graph Pattern	Supported Device Compute Capabilities	Supported Data Config and Layout	Supported Engine Types
`Conv_Bias_Add_activation`	All that cuDNN supports	Same as `cudnnConvolutionBiasActivationForward()`	Pattern matching engines, runtime fusion engines
`Scale_Bias_Activation_conv_bn_stats`	Compute capability 70 or above	`PSEUDO_HALF_CONFIG`, NHWC layout	Pattern matching engines, runtime fusion engines
`Convolution_Pointwise`	Compute capability 75 or above	Flexible	Runtime fusion engines
`Gemm_Pointwise`	Compute capability 75 or above	Flexible	Runtime fusion engines

Chapter 11. Troubleshooting

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

11.1. 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`, `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` or `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 5 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 5. 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

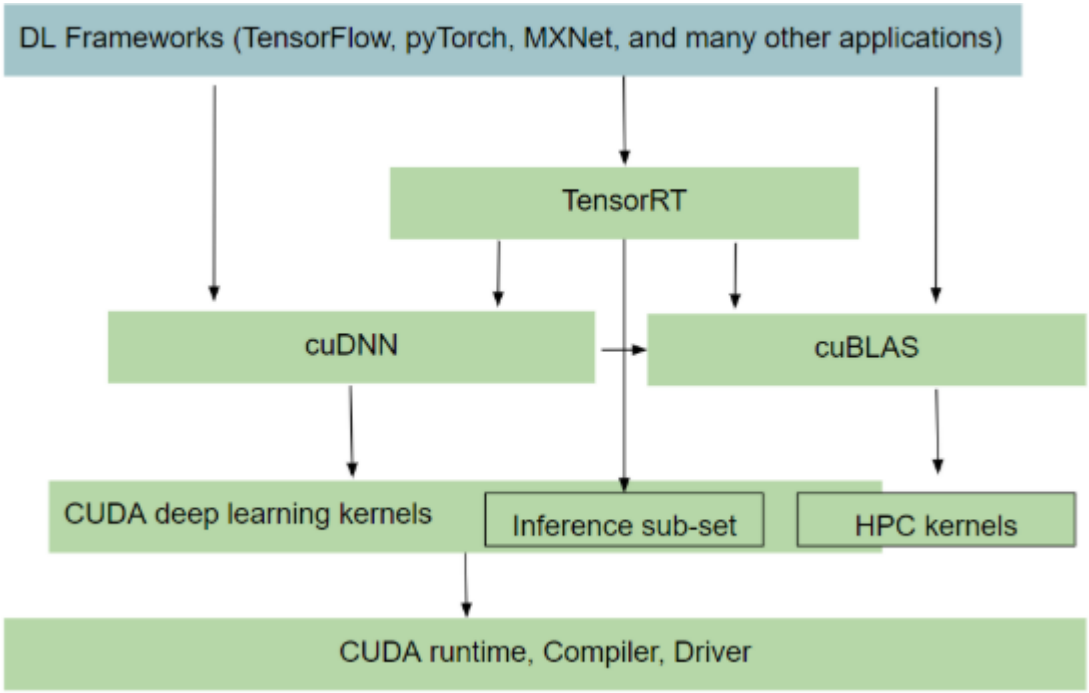
Method 2: To use API function calls to enable API logging, refer to the API description of cudnnSetCallback() and cudnnGetCallback().

11.2. 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 12. 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 what 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. 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, see 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 cuDNN 8.0.0 Preview Release Notes and the *NVIDIA cuDNN API Reference*.

Alternatively, the user can continue to link against a shim layer (`-lcudnn`) 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.

11.3. How Do I Report A Bug?

We appreciate all types of feedback. If you encounter any issues, please report them 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, attach an API log.
6. Click **Submit a bug**.

11.4. 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.

For questions or to provide feedback, please contact cuDNN@nvidia.com.

Chapter 12. Acknowledgments

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

12.1. University of Tennessee

```
Copyright (c) 2010 The University of Tennessee.

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer listed in this license in the documentation and/or
      other materials provided with the distribution.
    * Neither the name of the copyright holders nor the names of its
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

12.2. University of California, Berkeley

```
COPYRIGHT

All contributions by the University of California:
Copyright (c) 2014, The Regents of the University of California (Regents)
All rights reserved.

All other contributions:
Copyright (c) 2014, the respective contributors
All rights reserved.
```

```
Caffe uses a shared copyright model: each contributor holds copyright over
their contributions to Caffe. The project versioning records all such
contribution and copyright details. If a contributor wants to further mark
their specific copyright on a particular contribution, they should indicate
their copyright solely in the commit message of the change when it is
committed.

LICENSE

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTION AGREEMENT

By contributing to the BVLC/caffe repository through pull-request, comment,
or otherwise, the contributor releases their content to the
license and copyright terms herein.
```

12.3. Facebook AI Research, New York

```
Copyright (c) 2014, Facebook, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification,
are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

 * Neither the name Facebook nor the names of its contributors may be used to
   endorse or promote products derived from this software without specific
   prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
```

```
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
Additional Grant of Patent Rights

"Software" means fbcunn software distributed by Facebook, Inc.

Facebook hereby grants you a perpetual, worldwide, royalty-free, non-exclusive,
irrevocable (subject to the termination provision below) license under any
rights in any patent claims owned by Facebook, to make, have made, use, sell,
offer to sell, import, and otherwise transfer the Software. For avoidance of
doubt, no license is granted under Facebookâ€™s rights in any patent claims that
are infringed by (i) modifications to the Software made by you or a third party,
or (ii) the Software in combination with any software or other technology
provided by you or a third party.

The license granted hereunder will terminate, automatically and without notice,
for anyone that makes any claim (including by filing any lawsuit, assertion or
other action) alleging (a) direct, indirect, or contributory infringement or
inducement to infringe any patent: (i) by Facebook or any of its subsidiaries or
affiliates, whether or not such claim is related to the Software, (ii) by any
party if such claim arises in whole or in part from any software, product or
service of Facebook or any of its subsidiaries or affiliates, whether or not
such claim is related to the Software, or (iii) by any party relating to the
Software; or (b) that any right in any patent claim of Facebook is invalid or
unenforceable.
```

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

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

Trademarks

NVIDIA, the NVIDIA logo, and CUDA, DRIVE, JetPack, Kepler, Maxwell, Pascal, Turing, Volta and Xavier are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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