CUDNN DEVELOPER'S GUIDE

DU-06702-001_v7.6.5 | November 2019

Developer Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

NVIDIA 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
- Pooling forward and backward
- Softmax forward and backward
- Neuron activations forward and backward:
  - Rectified linear (ReLU)
  - Sigmoid
  - Hyperbolic tangent (TANH)
- Tensor transformation functions
- LRN, LCN and batch normalization forward and backward

cuDNN's 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 CUDA streams.

Chapter 3. 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 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 app 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.

Chapter 4. CONVOLUTION FORMULAS

This section describes the various convolution formulas implemented in cuDNN convolution functions.

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

Table 1 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)

Term	Description
r	Current Filter Height
R	Total Filter Height
s	Current Filter Width
S	Total Filter Width
C_g	$\frac{C}{G}$
K_g	$\frac{K}{G}$

Normal Convolution (using cross-correlation mode)

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

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 using Convolution Mode

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

Convolution using Grouped Convolution

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

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

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

Chapter 5. NOTATION

As of CUDNN v4 we have adopted a mathematicaly-inspired notation for layer inputs and outputs using **`x,y,dx,dy,b,w`** for common layer parameters. This was done to improve readability and ease of understanding of the meaning of the parameters. All layers now follow a uniform convention as below:

During inference

```
y = layerFunction(x, otherParams)
```

During backpropagation

```
(dx, dOtherParams) = layerFunctionGradient(x,y,dy,otherParams)
```

During convolution

For **convolution**, the notation is:

```
y = x*w+b
```

where:

- **`w`** is the matrix of filter weights
- **`x`** is the previous layer's data (during inference)
- **`y`** is the next layer's data
- **`b`** is the bias and **`*`** is the convolution operator

In backpropagation routines the parameters keep their meanings.

The parameters **`dx,dy,dw,db`** always refer to the gradient of the final network error function with respect to a given parameter. So **`dy`** in all backpropagation routines always refers to error gradient backpropagated through the network computation graph so far.

Similarly, other parameters in more specialized layers, such as, for instance, **`dMeans`** or **`dBnBias`** refer to gradients of the loss function with regard to those parameters.

`w` is used in the API for both the width of the **`x`** tensor and convolution filter matrix. To resolve this ambiguity we use **`w`** and **`filter`** notation interchangeably for convolution filter weight matrix. The meaning is clear from the context since the layer width is always referenced near its height.

Chapter 6. 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 others 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.

6.1. 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

6.2. 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`**

6.3. 5-D Tensor Description

A 5-D Tensor descriptor is used to define the format of 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`**

6.4. 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.

6.5. 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 last tensor's dimension is present in the -packed suffix, its stride is 1.

For example, a **`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 with slowest changing dimensions of a tensor but it is also possible to refer to a **`NCHW`** tensor that is only N-packed.

6.6. 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.

6.7. 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 7. DATA LAYOUT FORMATS

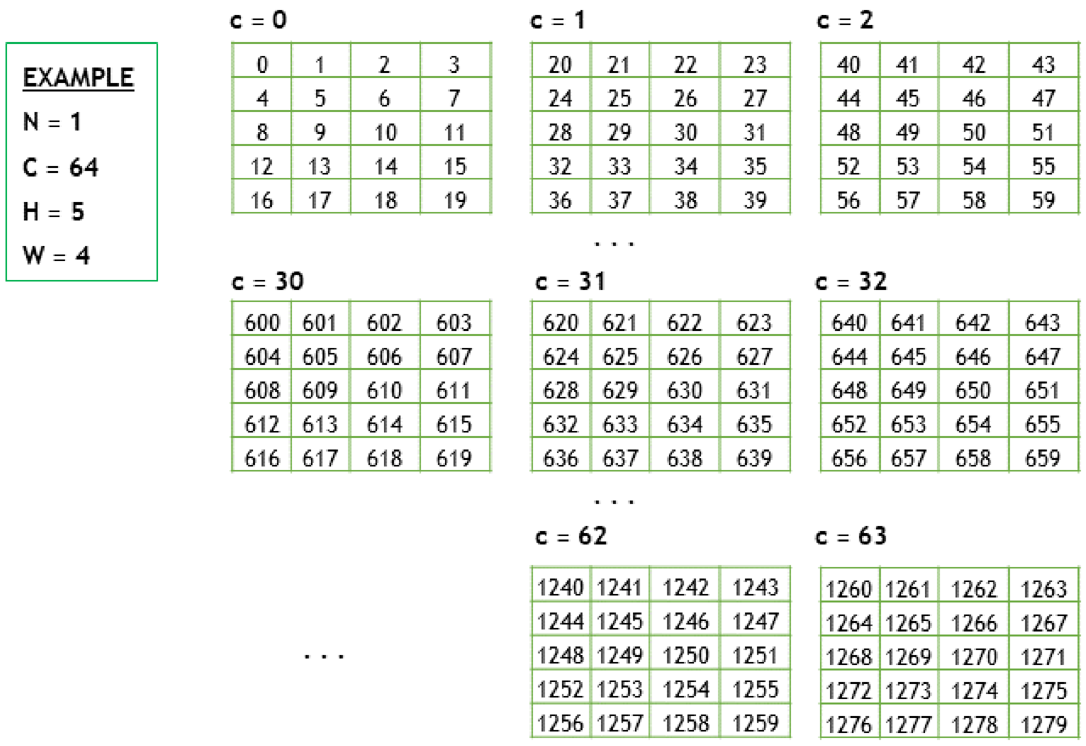
This section describes how cuDNN Tensors are arranged in memory. See cudnnTensorFormat_t for enumerated Tensor format types.

7.1. 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.



EXAMPLE

N = 1

C = 64

H = 5

W = 4

c = 0

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

c = 1

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

c = 2

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

. . .

c = 30

600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619

c = 31

620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639

c = 32

640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659

. . .

c = 62

1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

c = 63

1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

. . .

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

7.2. 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).

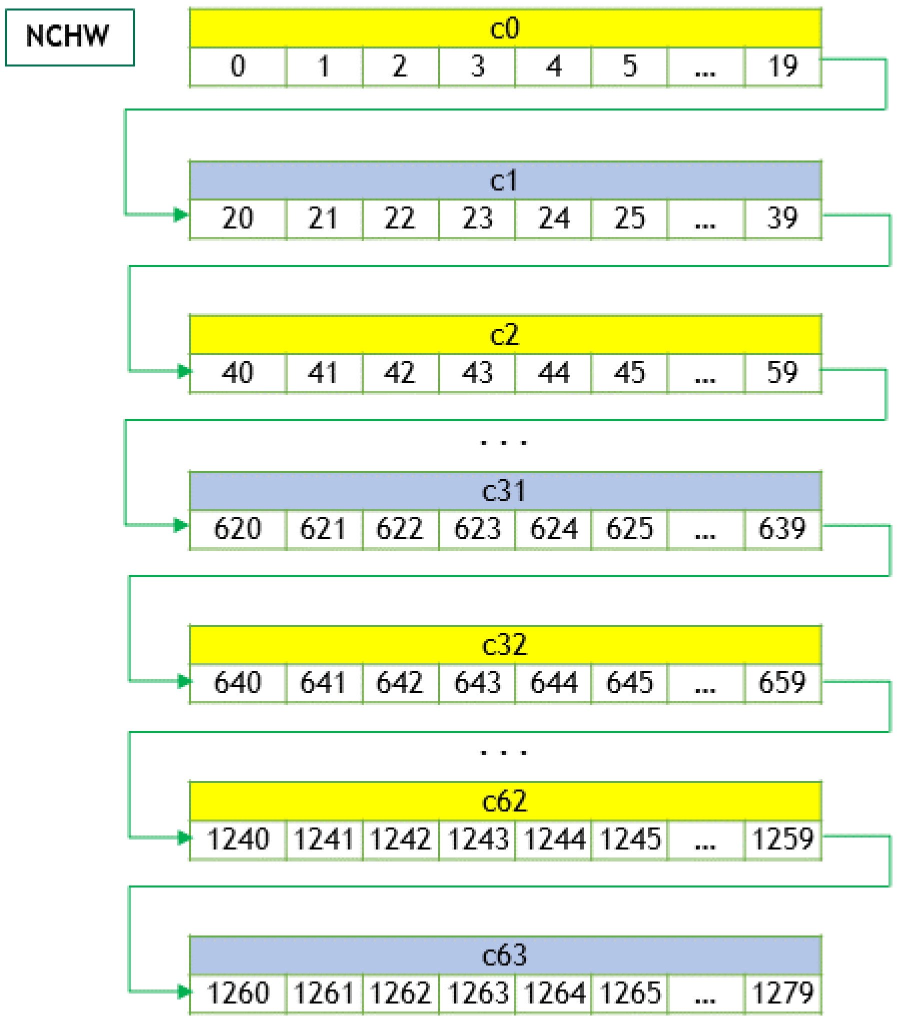


Figure 2 NCHW Memory Layout

7.3. 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 in channel 0 and complete all the elements. See Figure 3.
4. Proceed to the next batch (if **N** is > 1).

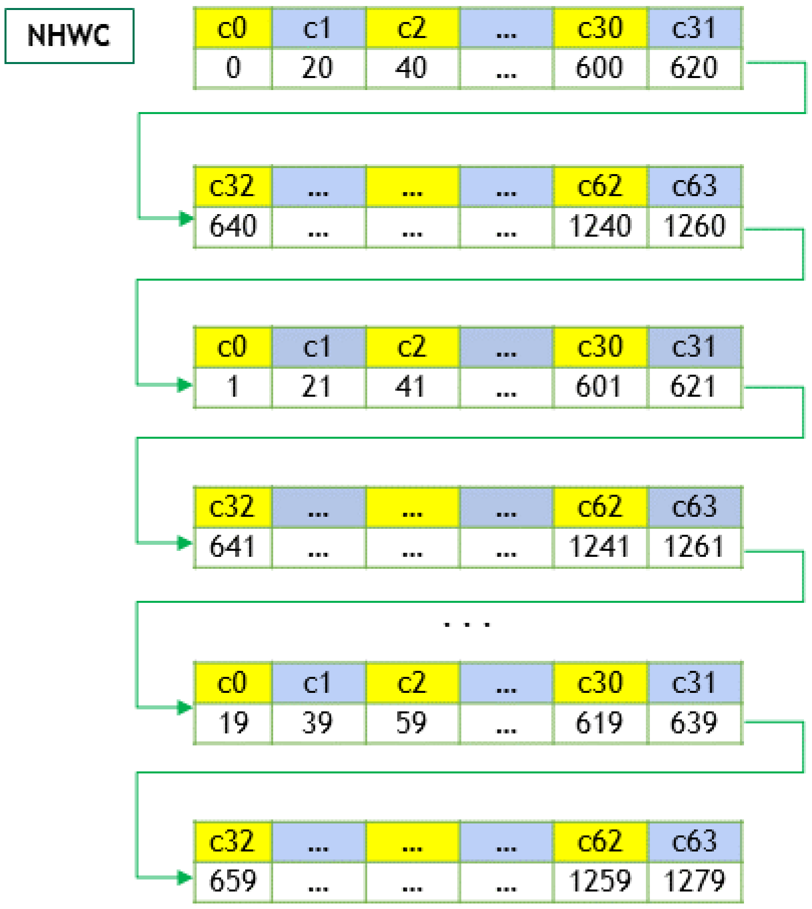


Figure 3 NHWC Memory Layout

7.4. 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.

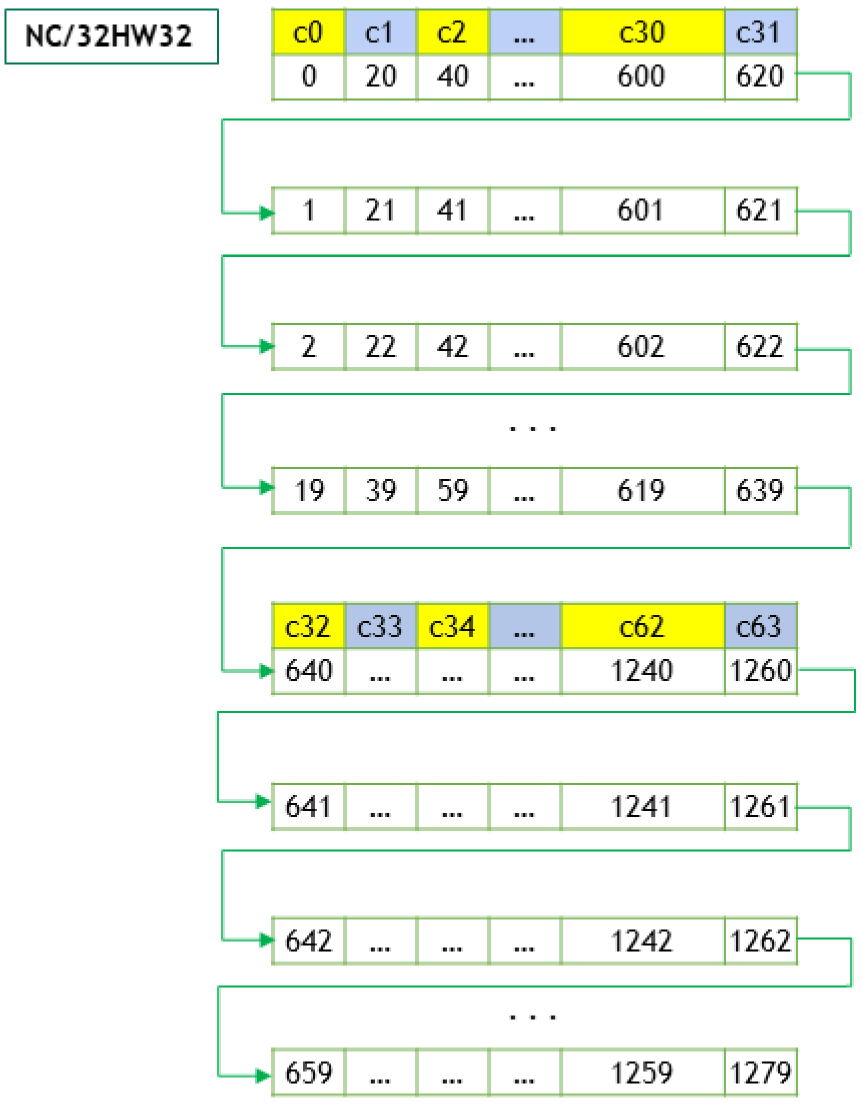


Figure 4 NC/32HW32 Memory Layout

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 entire C dimension as one big vector, laying out all the Cs, 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

Cs for every W), just as the NCHW INT8x4 format is N x (C/4) x H x W x 4 (4 Cs for every W). Hence the "VECT_C" name - each W is a vector (4 or 32) of Cs.

Chapter 8. THREAD SAFETY

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

Chapter 9. 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 and the same number of SMs. However, bit-wise reproducibility is not guaranteed across versions, as the implementation of a given routine may change. With the current release, the following routines do not guarantee reproducibility because they use atomic operations:

- **`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`**

Chapter 10. 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
```

The **dstValue** is written to after being read.

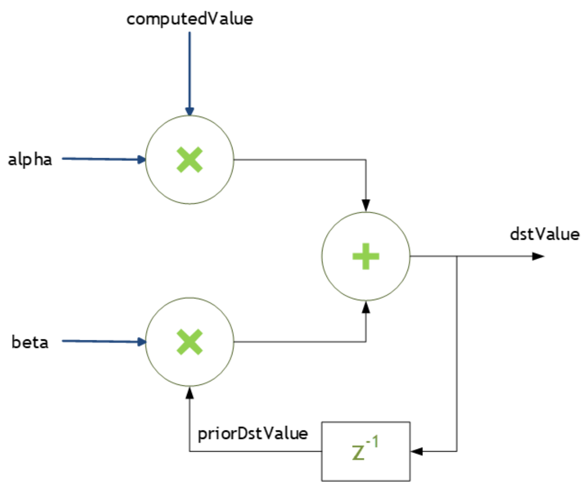


Figure 5 Scaling Parameters for Convolution

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.

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:

Accumulators are 32-bit integers which wrap on overflow.

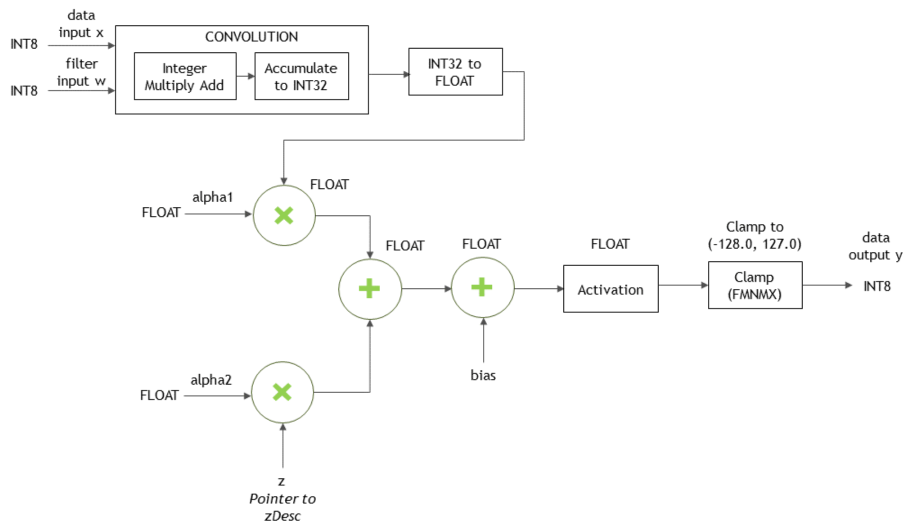


Figure 6 INT8 for cudnnConvolutionBiasActivationForward

Chapter 11. 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 on the Volta and Turing GPU families.

11.1. Basics

Tensor core operations perform parallel floating point accumulation of multiple floating point product terms. 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 to, but not always identical, 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 both convolutions and RNNs.

See also Training with Mixed Precision.

11.2. Convolution Functions

11.2.1. 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`**.

11.2.2. 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`

11.2.3. 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.

11.3. RNN Functions

11.3.1. Prerequisites

Tensor core operations will be 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`**.

11.3.2. 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()

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`

11.3.3. Data And Filter Formats

When the following requirements are met, then the cuDNN library will trigger 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 sizes 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.

11.4. 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.

11.4.1. FP16 Data

Tensor Cores operate on FP16 input data with FP32 accumulation. 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 7.

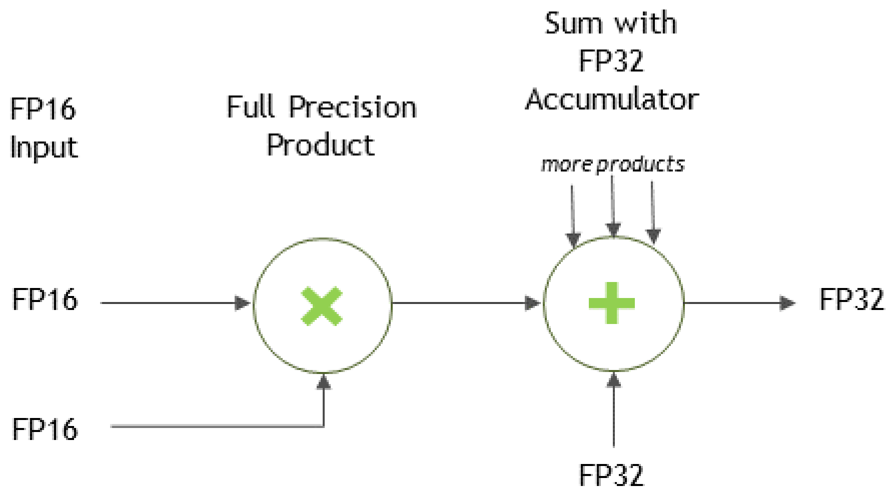


Figure 7 Tensor Operation with FP16 Inputs

11.4.2. FP32-to-FP16 Conversion

The cuDNN API for 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 8.

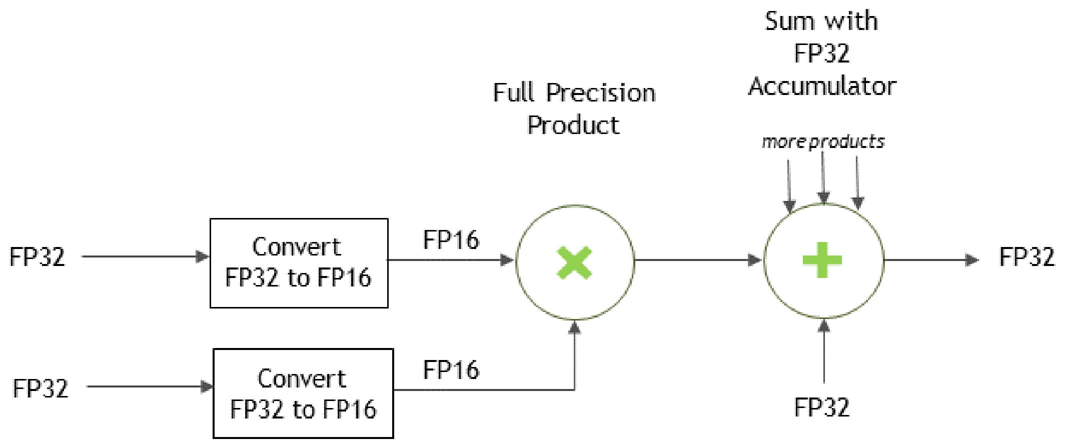


Figure 8 Tensor Operation with FP32 Inputs

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));
```

11.4.3. 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.

The padding occurs with a negligible loss of performance. Hence, the NCHW tensor dimensions such as below are allowed:

```
// Set NCHW Tensor dimensions, not necessarily as multiples of eight (only the
 input tensor is shown here):
int dimA[] = {1, 7, 32, 32};
int strideA[] = {7168, 1024, 32, 1};
```

11.4.4. 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.

Folding enables the input tensors to be transformed to a format that the Tensor Cores support (i.e., no strides).

11.4.5. Conversion Between NCHW And NHWC

Tensor Cores require that the tensors be in 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 are not enabled. In such cases, users should pre-pad.

11.5. Guidelines For A Deep Learning Compiler

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.

Chapter 12. GPU AND DRIVER REQUIREMENTS

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

Chapter 13. BACKWARD COMPATIBILITY AND DEPRECATION POLICY

When changing the API of an existing cuDNN function "foo" (usually to support some new functionality), first, a new routine "foo_v**<n>**" is created where **n** represents the cuDNN version where the new API is first introduced, leaving "foo" untouched. This ensures backward compatibility with the version **n-1** of cuDNN. At this point, "foo" is considered deprecated, and should be treated as such by users of cuDNN. We gradually eliminate deprecated and suffixed API entries over the course of a few releases of the library per the following policy:

- In release **n+1**, the legacy API entry "foo" is remapped to a new API "foo_v**<f>**" where **f** is some cuDNN version anterior to **n**.
- Also, in release **n+1**, the unsuffixed API entry "foo" is modified to have the same signature as "foo_**<n>**". "foo_**<n>**" is retained as-is.
- The deprecated former API entry with an anterior suffix _v**<f>** and new API entry with suffix _v**<n>** are maintained in this release.
- In release **n+2**, both suffixed entries of a given entry are removed.

As a rule of thumb, when a routine appears in two forms, one with a suffix and one with no suffix, the non-suffixed entry is to be treated as deprecated. In this case, it is strongly advised that users migrate to the new suffixed API entry to guarantee backwards compatibility in the following cuDNN release. When a routine appears with multiple suffixes, the unsuffixed API entry is mapped to the higher numbered suffix. In that case, it is strongly advised to use the non-suffixed API entry to guarantee backward compatibility with the following cuDNN release.

Chapter 14. GROUPED CONVOLUTIONS

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

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 **`groupCount`** number of independent groups, with each group having a reduced number of channels. 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.

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.

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]`**

Chapter 15. API LOGGING

cuDNN API logging is a tool that records all input parameters passed into every cuDNN API function call. This functionality is disabled by default, and can be enabled through methods described in this section.

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.

When logging is enabled, the log output will be handled by the built-in default callback function. The user may also write their own callback function, and use the cudnnSetCallback() to pass in the function pointer of their own callback function. The following is a sample output of the API log.

```
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.
```

There are two methods to enable API logging.

Method 1: Using Environment Variables

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

- Set the environment variable **`CUDNN_LOGINFO_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 writing to the same file at the same time.

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

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

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 the Method 2 below.

See also Table 2 for the impact on performance of API logging using environment variables.

Table 2 API Logging Using Environment Variables

Environment variables	CUDNN_LOGINFO_DBG=0	CUDNN_LOGINFO_DBG=1
`CUDNN_LOGDEST_DBG` not set	No logging output No performance loss	No logging output No performance loss
`CUDNN_LOGDEST_DB =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

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

Chapter 16. FEATURES OF RNN FUNCTIONS

The RNN functions are:

- cudnnRNNForwardInference()
- cudnnRNNForwardTraining()
- cudnnRNNBackwardData()
- cudnnRNNBackwardWeights()
- cudnnRNNForwardInferenceEx()
- cudnnRNNForwardTrainingEx()
- cudnnRNNBackwardDataEx()
- cudnnRNNBackwardWeightsEx()

See the table below for a list of features supported by each RNN function:

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`	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] (see the table below for an elaboration on these algorithms):
`cudnnRNNForwardTraining`			
`cudnnRNNBackwardData`			
`cudnnRNNBackwardWeights`			
`cudnnRNNForwardInferenceEx`	▸ Sequence major unpacked	Only with `_ALGO_STANDARD`	
`cudnnRNNForwardTrainingEx`			

[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.

Functions	Input output layout supported	Supports variable sequence length in batch	Commonly supported
cudnnRNNBackwardDataEx	▸ Batch major unpacked[2] ▸ Sequence major packed[2]	For unpacked layout[2], no input sorting required. For packed layout, require input sequences descending sorted according to length	**_ALGO_STANDARD**, **_ALGO_PERSIST_STATIC**, **_ALGO_PERSIST_DYNAMIC** Math mode supported: **CUDNN_DEFAULT_MATH**,**CUDNN_TENSOR_OP_** (will automatically fall back if run on pre-Volta or if algo doesn't support Tensor Cores) **_ALLOW_CONVERSION** (may do down conversion to utilize Tensor Cores) Direction mode supported: **CUDNN_UNIDIRECTIONAL**, **CUDNN_BIDIRECTIONAL** RNN input mode: **CUDNN_LINEAR_INPUT**, **CUDNN_SKIP_INPUT**
cudnnRNNBackwardWeightsEx			

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_STAT**	**_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**[1], 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	Supported Double intermediate storage Double accumulation

[2] To use unpacked layout, user need to set **CUDNN_RNN_PADDED_IO_ENABLED** through **cudnnSetRNNPaddingMode()**.

Features	_ALGO_STANDARD	_ALGO_PERSIST_STAT	_ALGO_PERSIST_DYNAMIC
LSTM recurrent projection	Supported	Not Supported	Not Supported
LSTM cell clipping	Supported		
Variable sequence length in batch	Supported	Not Supported	Not Supported
Tensor Cores on Volta/ Xavier	Supported For half input/output, acceleration requires setting `CUDNN_TENSOR_OP_MATH`[3] or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` [3] Acceleration requires `inputSize` and `hiddenSize` to be a multiple of 8 For single input/output, acceleration requires setting `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`[3] Acceleration requires `inputSize` and `hiddenSize` to be a multiple of 8		Not Supported, will execute normally ignoring `CUDNN_TENSOR_OP_MATH`[3] or `_ALLOW_CONVERSION`[3]
Other limitations		Max problem size is limited by GPU specifications.	Requires real time compilation through NVRTC

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

Chapter 17. 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 18. ACKNOWLEDGMENTS

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

18.1. University of Tennessee

18.2. University of California, Berkeley

18.3. Facebook AI Research, New York