# CUDNN DEVELOPER'S GUIDE

DU-06702-001_v7.6.4 | September 2019

## Developer Guide

# TABLE OF CONTENTS

**Chapter 1. Overview** .... 1
**Chapter 2. General Description** .... 2
2.1. Programming Model .... 2
2.2. Convolution Formulas .... 3
2.3. Notation .... 4
2.4. Tensor Descriptor .... 5
2.4.1. WXYZ Tensor Descriptor .... 6
2.4.2. 4-D Tensor Descriptor .... 6
2.4.3. 5-D Tensor Description .... 6
2.4.4. Fully-packed tensors .... 6
2.4.5. Partially-packed tensors .... 6
2.4.6. Spatially packed tensors .... 7
2.4.7. Overlapping tensors .... 7
2.5. Data Layout Formats .... 7
2.5.1. Example .... 7
2.5.2. NCHW Memory Layout .... 8
2.5.3. NHWC Memory Layout .... 9
2.5.4. NC/32HW32 Memory Layout .... 10
2.6. Thread Safety .... 12
2.7. Reproducibility (determinism) .... 12
2.8. Scaling Parameters .... 12
2.9. Tensor Core Operations .... 14
2.9.1. Basics .... 14
2.9.2. Convolution Functions .... 15
2.9.2.1. Prerequisite .... 15
2.9.2.2. Supported Algorithms .... 15
2.9.2.3. Data and Filter Formats .... 15
2.9.3. RNN Functions .... 16
2.9.3.1. Prerequisite .... 16
2.9.3.2. Supported Algorithms .... 16
2.9.3.3. Data and Filter Formats .... 16
2.9.4. Tensor Transformations .... 17
2.9.4.1. FP16 Data .... 17
2.9.4.2. FP32-to-FP16 Conversion .... 17
2.9.4.3. Padding .... 18
2.9.4.4. Folding .... 19
2.9.4.5. Conversion Between NCHW and NHWC .... 19
2.9.5. Guidelines for a Deep Learning Compiler .... 19
2.10. GPU and driver requirements .... 19
2.11. Backward compatibility and deprecation policy .... 20

2.12. Grouped Convolutions........20
2.13. API Logging........22
2.14. Features of RNN Functions........23
2.15. Mixed Precision Numerical Accuracy........26
**Chapter 3. cuDNN Datatypes Reference........27**
3.1. cudnnActivationDescriptor_t........27
3.2. cudnnActivationMode_t........27
3.3. cudnnAttnDescriptor_t........28
3.4. cudnnBatchNormMode_t........28
3.5. cudnnBatchNormOps_t........29
3.6. cudnnConvolutionBwdDataAlgo_t........29
3.7. cudnnConvolutionBwdDataAlgoPerf_t........30
3.8. cudnnConvolutionBwdDataPreference_t........31
3.9. cudnnConvolutionBwdFilterAlgo_t........31
3.10. cudnnConvolutionBwdFilterAlgoPerf_t........32
3.11. cudnnConvolutionBwdFilterPreference_t........33
3.12. cudnnConvolutionDescriptor_t........33
3.13. cudnnConvolutionFwdAlgo_t........34
3.14. cudnnConvolutionFwdAlgoPerf_t........34
3.15. cudnnConvolutionFwdPreference_t........35
3.16. cudnnConvolutionMode_t........36
3.17. cudnnCTCLossAlgo_t........36
3.18. cudnnCTCLossDescriptor_t........36
3.19. cudnnDataType_t........37
3.20. cudnnDeterminism_t........37
3.21. cudnnDirectionMode_t........38
3.22. cudnnDivNormMode_t........38
3.23. cudnnDropoutDescriptor_t........38
3.24. cudnnErrQueryMode_t........39
3.25. cudnnFilterDescriptor_t........39
3.26. cudnnFoldingDirection_t........39
3.27. cudnnFusedOps_t........39
3.28. cudnnFusedOpsConstParamLabel_t........41
3.29. cudnnFusedOpsConstParamPack_t........55
3.30. cudnnFusedOpsPlan_t........55
3.31. cudnnFusedOpsPointerPlaceHolder_t........55
3.32. cudnnFusedOpsVariantParamLabel_t........56
3.33. cudnnFusedOpsVariantParamPack_t........64
3.34. cudnnHandle_t........64
3.35. cudnnIndicesType_t........65
3.36. cudnnLossNormalizationMode_t........65
3.37. cudnnLRNMode_t........65
3.38. cudnnMathType_t........66

3.39. cudnnMultiHeadAttnWeightKind_t........66
3.40. cudnnNanPropagation_t........67
3.41. cudnnOpTensorDescriptor_t........67
3.42. cudnnOpTensorOp_t........67
3.43. cudnnPersistentRNNPlan_t........68
3.44. cudnnPoolingDescriptor_t........68
3.45. cudnnPoolingMode_t........68
3.46. cudnnReduceTensorDescriptor_t........68
3.47. cudnnReduceTensorIndices_t........69
3.48. cudnnReduceTensorOp_t........69
3.49. cudnnReorderType_t........70
3.50. cudnnRNNAlgo_t........70
3.51. cudnnRNNBiasMode_t........71
3.52. cudnnRNNClipMode_t........71
3.53. cudnnRNNDataDescriptor_t........71
3.54. cudnnRNNDataLayout_t........71
3.55. cudnnRNNDescriptor_t........72
3.56. cudnnRNNInputMode_t........72
3.57. cudnnRNNMode_t........72
3.58. cudnnRNNPaddingMode_t........75
3.59. cudnnSamplerType_t........75
3.60. cudnnSeqDataAxis_t........75
3.61. cudnnSeqDataDescriptor_t........76
3.62. cudnnSoftmaxAlgorithm_t........76
3.63. cudnnSoftmaxMode_t........77
3.64. cudnnSpatialTransformerDescriptor_t........77
3.65. cudnnStatus_t........77
3.66. cudnnTensorDescriptor_t........79
3.67. cudnnTensorFormat_t........79
3.68. cudnnTensorTransformDescriptor_t........80
3.69. cudnnWgradMode_t........80
**Chapter 4. cuDNN API Reference........81**
4.1. cudnnActivationBackward........81
4.2. cudnnActivationForward........83
4.3. cudnnAddTensor........84
4.4. cudnnBatchNormalizationBackward........86
4.5. cudnnBatchNormalizationBackwardEx........89
4.6. cudnnBatchNormalizationForwardInference........92
4.7. cudnnBatchNormalizationForwardTraining........94
4.8. cudnnBatchNormalizationForwardTrainingEx........97
4.9. cudnnConvolutionBackwardBias........101
4.10. cudnnConvolutionBackwardData........102
4.11. cudnnConvolutionBackwardFilter........112

4.12. cudnnConvolutionBiasActivationForward........120
4.13. cudnnConvolutionForward........124
4.14. cudnnCreate........133
4.15. cudnnCreateActivationDescriptor........134
4.16. cudnnCreateAlgorithmDescriptor........135
4.17. cudnnCreateAlgorithmPerformance........135
4.18. cudnnCreateAttnDescriptor........136
4.19. cudnnCreateConvolutionDescriptor........136
4.20. cudnnCreateCTCLossDescriptor........136
4.21. cudnnCreateDropoutDescriptor........137
4.22. cudnnCreateFilterDescriptor........137
4.23. cudnnCreateFusedOpsConstParamPack........138
4.24. cudnnCreateFusedOpsPlan........138
4.25. cudnnCreateFusedOpsVariantParamPack........139
4.26. cudnnCreateLRNDescriptor........139
4.27. cudnnCreateOpTensorDescriptor........140
4.28. cudnnCreatePersistentRNNPlan........140
4.29. cudnnCreatePoolingDescriptor........141
4.30. cudnnCreateReduceTensorDescriptor........141
4.31. cudnnCreateRNNDataDescriptor........142
4.32. cudnnCreateRNNDescriptor........142
4.33. cudnnCreateSeqDataDescriptor........142
4.34. cudnnCreateSpatialTransformerDescriptor........143
4.35. cudnnCreateTensorDescriptor........143
4.36. cudnnCreateTensorTransformDescriptor........144
4.37. cudnnCTCLoss........144
4.38. cudnnDeriveBNTensorDescriptor........146
4.39. cudnnDestroy........147
4.40. cudnnDestroyActivationDescriptor........148
4.41. cudnnDestroyAlgorithmDescriptor........148
4.42. cudnnDestroyAlgorithmPerformance........148
4.43. cudnnDestroyAttnDescriptor........149
4.44. cudnnDestroyConvolutionDescriptor........149
4.45. cudnnDestroyCTCLossDescriptor........149
4.46. cudnnDestroyDropoutDescriptor........150
4.47. cudnnDestroyFilterDescriptor........150
4.48. cudnnDestroyFusedOpsConstParamPack........150
4.49. cudnnDestroyFusedOpsPlan........151
4.50. cudnnDestroyFusedOpsVariantParamPack........151
4.51. cudnnDestroyLRNDescriptor........151
4.52. cudnnDestroyOpTensorDescriptor........152
4.53. cudnnDestroyPersistentRNNPlan........152
4.54. cudnnDestroyPoolingDescriptor........152

4.55. cudnnDestroyReduceTensorDescriptor....153
4.56. cudnnDestroyRNNDataDescriptor....153
4.57. cudnnDestroyRNNDescriptor....153
4.58. cudnnDestroySeqDataDescriptor....154
4.59. cudnnDestroySpatialTransformerDescriptor....154
4.60. cudnnDestroyTensorDescriptor....154
4.61. cudnnDestroyTensorTransformDescriptor....155
4.62. cudnnDivisiveNormalizationBackward....155
4.63. cudnnDivisiveNormalizationForward....157
4.64. cudnnDropoutBackward....159
4.65. cudnnDropoutForward....161
4.66. cudnnDropoutGetReserveSpaceSize....162
4.67. cudnnDropoutGetStatesSize....163
4.68. cudnnFindConvolutionBackwardDataAlgorithm....163
4.69. cudnnFindConvolutionBackwardDataAlgorithmEx....165
4.70. cudnnFindConvolutionBackwardFilterAlgorithm....167
4.71. cudnnFindConvolutionBackwardFilterAlgorithmEx....169
4.72. cudnnFindConvolutionForwardAlgorithm....171
4.73. cudnnFindConvolutionForwardAlgorithmEx....173
4.74. cudnnFindRNNBackwardDataAlgorithmEx....175
4.75. cudnnFindRNNBackwardWeightsAlgorithmEx....181
4.76. cudnnFindRNNForwardInferenceAlgorithmEx....185
4.77. cudnnFindRNNForwardTrainingAlgorithmEx....190
4.78. cudnnFusedOpsExecute....195
4.79. cudnnGetActivationDescriptor....195
4.80. cudnnGetAlgorithmDescriptor....196
4.81. cudnnGetAlgorithmPerformance....196
4.82. cudnnGetAlgorithmSpaceSize....197
4.83. cudnnGetAttnDescriptor....197
4.84. cudnnGetBatchNormalizationBackwardExWorkspaceSize....199
4.85. cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize....201
4.86. cudnnGetBatchNormalizationTrainingExReserveSpaceSize....202
4.87. cudnnGetCallback....203
4.88. cudnnGetConvolution2dDescriptor....204
4.89. cudnnGetConvolution2dForwardOutputDim....205
4.90. cudnnGetConvolutionBackwardDataAlgorithm....206
4.91. cudnnGetConvolutionBackwardDataAlgorithm_v7....208
4.92. cudnnGetConvolutionBackwardDataAlgorithmMaxCount....209
4.93. cudnnGetConvolutionBackwardDataWorkspaceSize....209
4.94. cudnnGetConvolutionBackwardFilterAlgorithm....211
4.95. cudnnGetConvolutionBackwardFilterAlgorithm_v7....212
4.96. cudnnGetConvolutionBackwardFilterAlgorithmMaxCount....213
4.97. cudnnGetConvolutionBackwardFilterWorkspaceSize....214

4.98. cudnnGetConvolutionForwardAlgorithm....215
4.99. cudnnGetConvolutionForwardAlgorithm_v7....216
4.100. cudnnGetConvolutionForwardAlgorithmMaxCount....218
4.101. cudnnGetConvolutionForwardWorkspaceSize....218
4.102. cudnnGetConvolutionGroupCount....219
4.103. cudnnGetConvolutionMathType....220
4.104. cudnnGetConvolutionNdDescriptor....220
4.105. cudnnGetConvolutionNdForwardOutputDim....221
4.106. cudnnGetConvolutionReorderType....223
4.107. cudnnGetCTCLossDescriptor....223
4.108. cudnnGetCTCLossWorkspaceSize....224
4.109. cudnnGetCudartVersion....225
4.110. cudnnGetDropoutDescriptor....225
4.111. cudnnGetErrorString....226
4.112. cudnnGetFilter4dDescriptor....226
4.113. cudnnGetFilterNdDescriptor....227
4.114. cudnnGetFusedOpsConstParamPackAttribute....228
4.115. cudnnGetFusedOpsVariantParamPackAttribute....229
4.116. cudnnGetLRNDescriptor....230
4.117. cudnnGetMultiHeadAttnBuffers....230
4.118. cudnnGetMultiHeadAttnWeights....232
4.119. cudnnGetOpTensorDescriptor....233
4.120. cudnnGetPooling2dDescriptor....234
4.121. cudnnGetPooling2dForwardOutputDim....235
4.122. cudnnGetPoolingNdDescriptor....236
4.123. cudnnGetPoolingNdForwardOutputDim....237
4.124. cudnnGetProperty....238
4.125. cudnnGetReduceTensorDescriptor....239
4.126. cudnnGetReductionIndicesSize....239
4.127. cudnnGetReductionWorkspaceSize....240
4.128. cudnnGetRNNBiasMode....241
4.129. cudnnGetRNNDataDescriptor....241
4.130. cudnnGetRNNDescriptor....243
4.131. cudnnGetRNNLinLayerBiasParams....244
4.132. cudnnGetRNNLinLayerMatrixParams....247
4.133. cudnnGetRNNPaddingMode....250
4.134. cudnnGetRNNParamsSize....250
4.135. cudnnGetRNNProjectionLayers....251
4.136. cudnnGetRNNTrainingReserveSize....252
4.137. cudnnGetRNNWorkspaceSize....253
4.138. cudnnGetSeqDataDescriptor....255
4.139. cudnnGetStream....256
4.140. cudnnGetTensor4dDescriptor....257

4.141. cudnnGetTensorNdDescriptor....258
4.142. cudnnGetTensorSizeInBytes....259
4.143. cudnnGetTensorTransformDescriptor....260
4.144. cudnnGetVersion....261
4.145. cudnnIm2Col....261
4.146. cudnnInitTransformDest....262
4.147. cudnnLRNCrossChannelBackward....263
4.148. cudnnLRNCrossChannelForward....265
4.149. cudnnMakeFusedOpsPlan....266
4.150. cudnnMultiHeadAttnBackwardData....267
4.151. cudnnMultiHeadAttnBackwardWeights....270
4.152. cudnnMultiHeadAttnForward....273
4.153. cudnnOpTensor....277
4.154. cudnnPoolingBackward....279
4.155. cudnnPoolingForward....281
4.156. cudnnQueryRuntimeError....283
4.157. cudnnReduceTensor....284
4.158. cudnnReorderFilterAndBias....286
4.159. cudnnRestoreAlgorithm....287
4.160. cudnnRestoreDropoutDescriptor....288
4.161. cudnnRNNBackwardData....289
4.162. cudnnRNNBackwardDataEx....295
4.163. cudnnRNNBackwardWeights....300
4.164. cudnnRNNBackwardWeightsEx....302
4.165. cudnnRNNForwardInference....305
4.166. cudnnRNNForwardInferenceEx....309
4.167. cudnnRNNForwardTraining....313
4.168. cudnnRNNForwardTrainingEx....317
4.169. cudnnRNNGetClip....322
4.170. cudnnRNNSetClip....323
4.171. cudnnSaveAlgorithm....324
4.172. cudnnScaleTensor....324
4.173. cudnnSetActivationDescriptor....325
4.174. cudnnSetAlgorithmDescriptor....326
4.175. cudnnSetAlgorithmPerformance....326
4.176. cudnnSetAttnDescriptor....327
4.177. cudnnSetCallback....332
4.178. cudnnSetConvolution2dDescriptor....334
4.179. cudnnSetConvolutionGroupCount....335
4.180. cudnnSetConvolutionMathType....335
4.181. cudnnSetConvolutionNdDescriptor....336
4.182. cudnnSetConvolutionReorderType....337
4.183. cudnnSetCTCLossDescriptor....338

4.184. cudnnSetCTCLossDescriptorEx....338
4.185. cudnnSetDropoutDescriptor....339
4.186. cudnnSetFilter4dDescriptor....340
4.187. cudnnSetFilterNdDescriptor.... 342
4.188. cudnnSetFusedOpsConstParamPackAttribute.... 343
4.189. cudnnSetFusedOpsVariantParamPackAttribute....344
4.190. cudnnSetLRNDescriptor....345
4.191. cudnnSetOpTensorDescriptor.... 346
4.192. cudnnSetPersistentRNNPlan....346
4.193. cudnnSetPooling2dDescriptor....347
4.194. cudnnSetPoolingNdDescriptor.... 348
4.195. cudnnSetReduceTensorDescriptor....349
4.196. cudnnSetRNNBiasMode....350
4.197. cudnnSetRNNDataDescriptor....351
4.198. cudnnSetRNNDescriptor.... 352
4.199. cudnnSetRNNDescriptor_v5.... 354
4.200. cudnnSetRNNDescriptor_v6.... 355
4.201. cudnnSetRNNMatrixMathType....357
4.202. cudnnSetRNNPaddingMode....357
4.203. cudnnSetRNNProjectionLayers....358
4.204. cudnnSetSeqDataDescriptor.... 359
4.205. cudnnSetSpatialTransformerNdDescriptor.... 363
4.206. cudnnSetStream.... 364
4.207. cudnnSetTensor.... 364
4.208. cudnnSetTensor4dDescriptor....365
4.209. cudnnSetTensor4dDescriptorEx....366
4.210. cudnnSetTensorNdDescriptor.... 368
4.211. cudnnSetTensorNdDescriptorEx.... 369
4.212. cudnnSetTensorTransformDescriptor.... 370
4.213. cudnnSoftmaxBackward.... 371
4.214. cudnnSoftmaxForward....373
4.215. cudnnSpatialTfGridGeneratorBackward....374
4.216. cudnnSpatialTfGridGeneratorForward.... 375
4.217. cudnnSpatialTfSamplerBackward.... 376
4.218. cudnnSpatialTfSamplerForward.... 378
4.219. cudnnTransformTensor....380
4.220. cudnnTransformTensorEx....381
**Chapter 5. Acknowledgments.... 383**
5.1. University of Tennessee....383
5.2. University of California, Berkeley....383
5.3. Facebook AI Research, New York....384

# 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 2. GENERAL DESCRIPTION

Basic concepts are described in this section.

## 2.1. 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 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 (i.e., 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 (i.e., 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.

## 2.2. 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 OF CONVOLUTION TERMS**

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

| Term | Description |
| --- | --- |
| $C_g$ | $\frac{C}{G}$ |
| $K_g$ | $\frac{K}{G}$ |

**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**

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

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

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

# 2.3. 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 the 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)
```

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 wrt 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.

## 2.4. 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 (e.g 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.

## 2.4.1. 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

## 2.4.2. 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

## 2.4.3. 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 descreasing order of the strides. The commonly used 5-D tensor formats are called :

- NCDHW
- NDHWC
- CDHWN

## 2.4.4. 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.

## 2.4.5. Partially-packed tensors

The partially 'XYZ-packed' terminology only applies in a 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.

### 2.4.6. 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.

### 2.4.7. Overlapping tensors

A tensor is defined to be overlapping if a iterating over a full range of dimensions produces the same address more than once.

In practice an overlapped tensor will have stride[i-1] < stride[i]*dim[i] for some of the i from [1,nbDims] interval.

## 2.5. Data Layout Formats

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

### 2.5.1. Example

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

- **N**, the batch size, is 1
- **C**, the number of feature maps (i.e., number of channels), is 64
- **H**, the image height, is 5, and
- **W**, the image width, is 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.

## 2.5.2. 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).

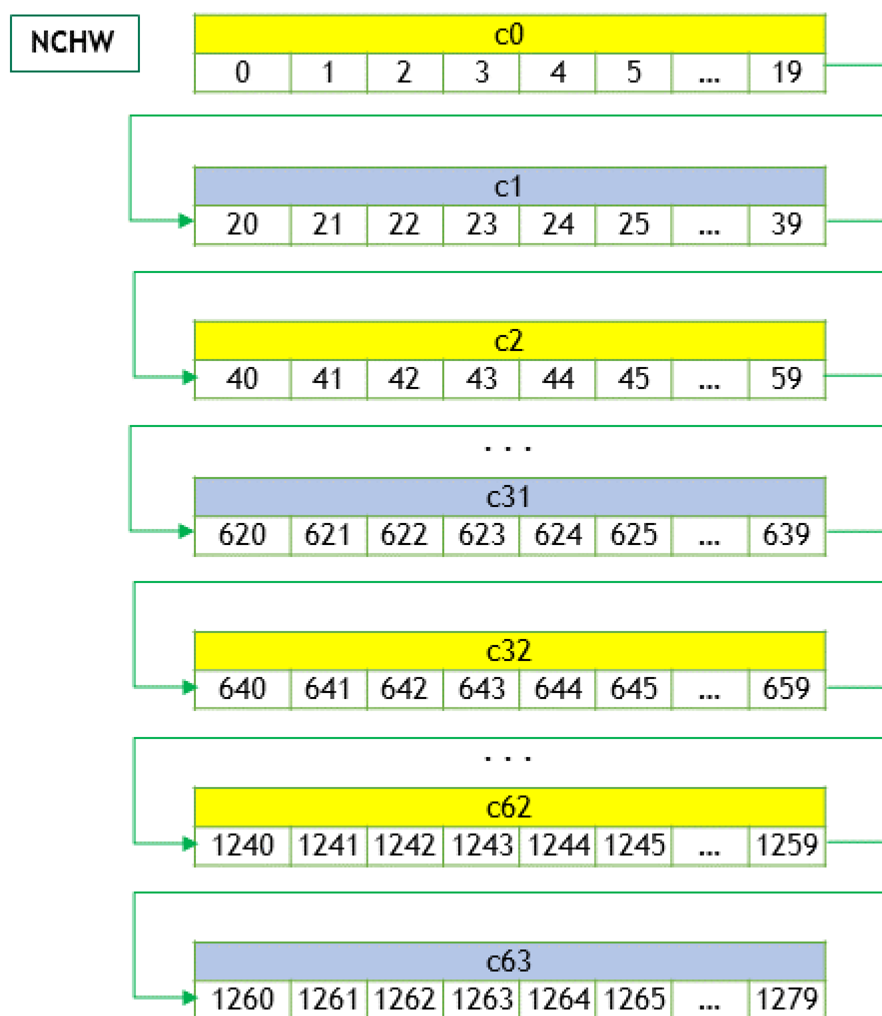



Figure 2 NCHW Memory Layout

## 2.5.3. 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).

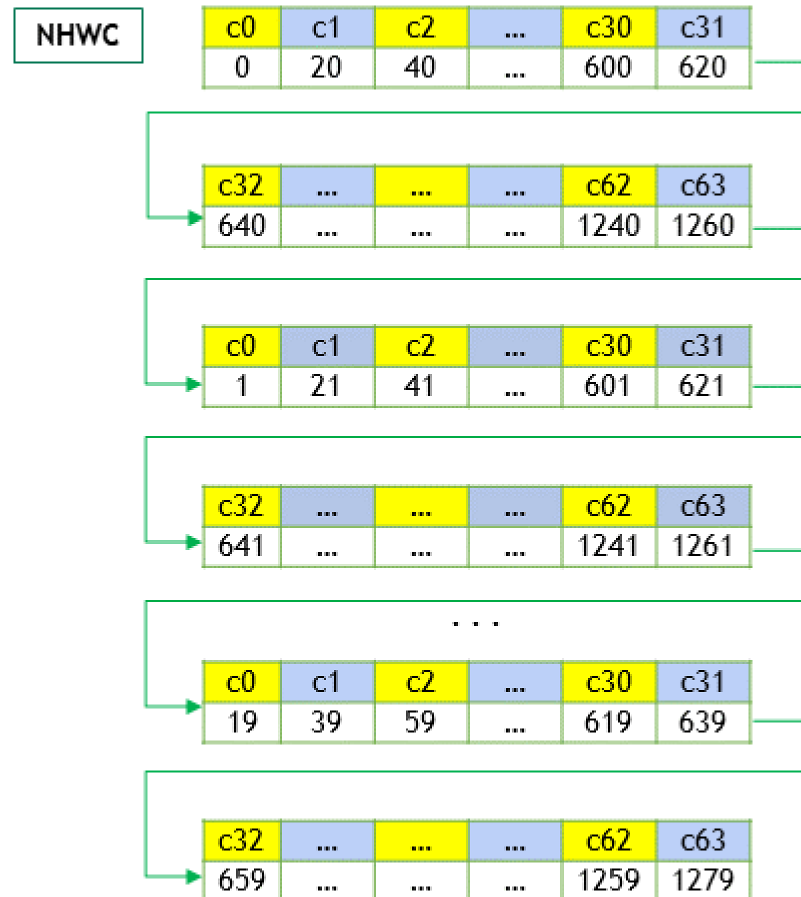



Figure 3 NHWC Memory Layout

## 2.5.4. 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.

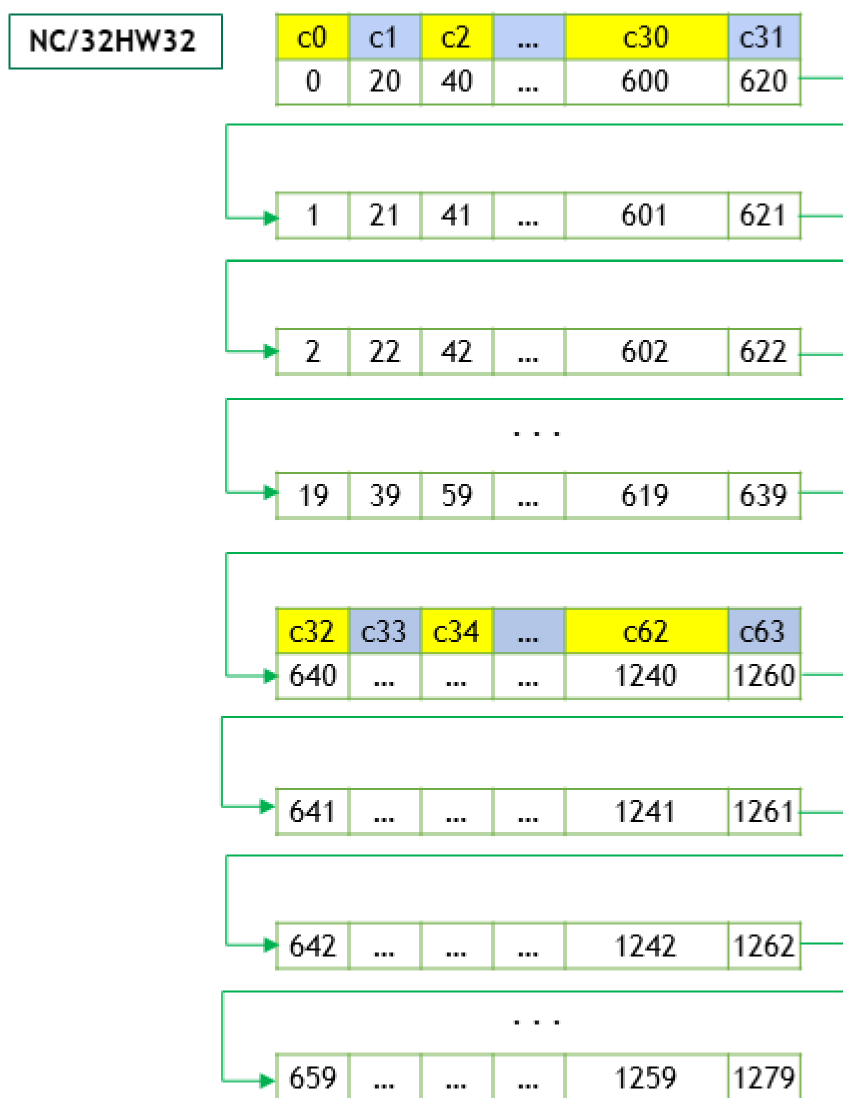



Figure 4 NC/32HW32 Memory Layout

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.

## 2.6. 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.

## 2.7. 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`**

## 2.8. 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.

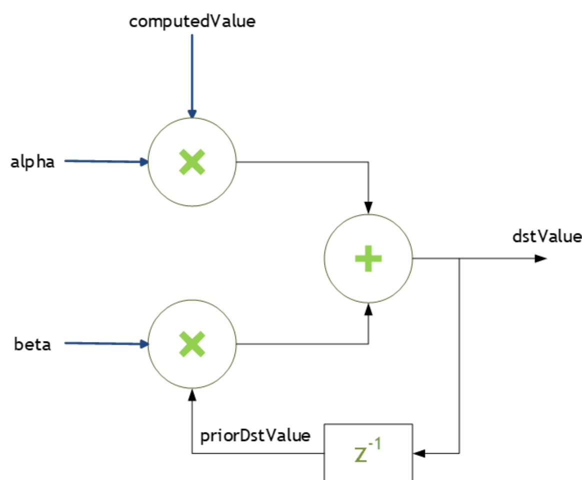



Figure 5 Scaling Parameters for Convolution

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.

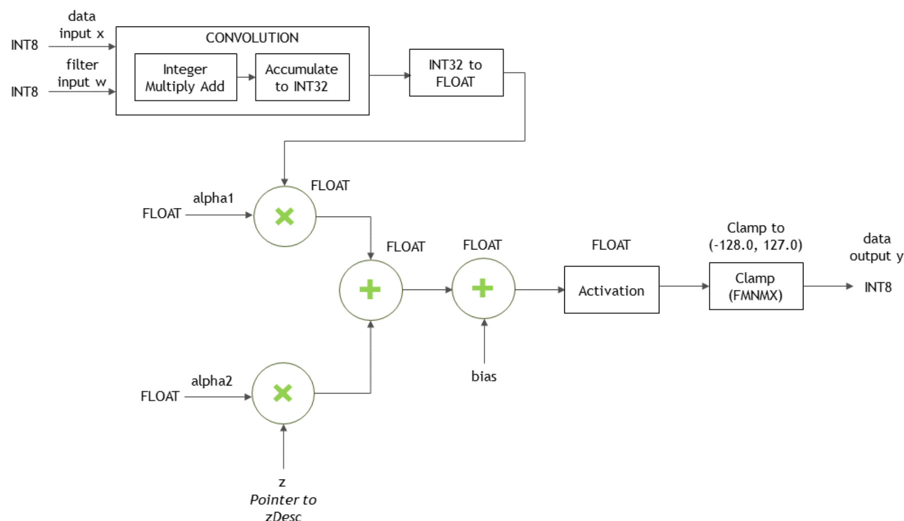



Figure 6 INT8 for cudnnConvolutionBiasActivationForward

# 2.9. 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.

## 2.9.1. 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.

## 2.9.2. Convolution Functions

### 2.9.2.1. Prerequisite

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.

### 2.9.2.2. 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,<br>-CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED |
| cudnnConvolutionBackwardData | -CUDNN_CONVOLUTION_BWD_DATA_ALGO_1,<br>-CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED |
| cudnnConvolutionBackwardFilter | -CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1,<br>-CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED |

### 2.9.2.3. 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.

## 2.9.3. RNN Functions

### 2.9.3.1. Prerequisite

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.

### 2.9.3.2. 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<br>-CUDNN_RNN_ALGO_PERSIST_STATIC (new for cuDNN 7.1) |

### 2.9.3.3. 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. (new for 7.1).
  - 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.

## 2.9.4. 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.

### 2.9.4.1. 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.

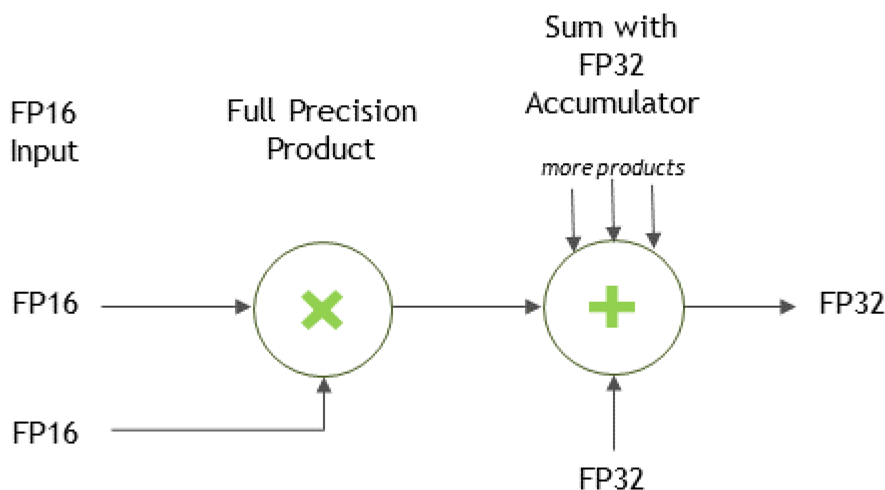



Figure 7 Tensor Operation with FP16 Inputs

### 2.9.4.2. 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.

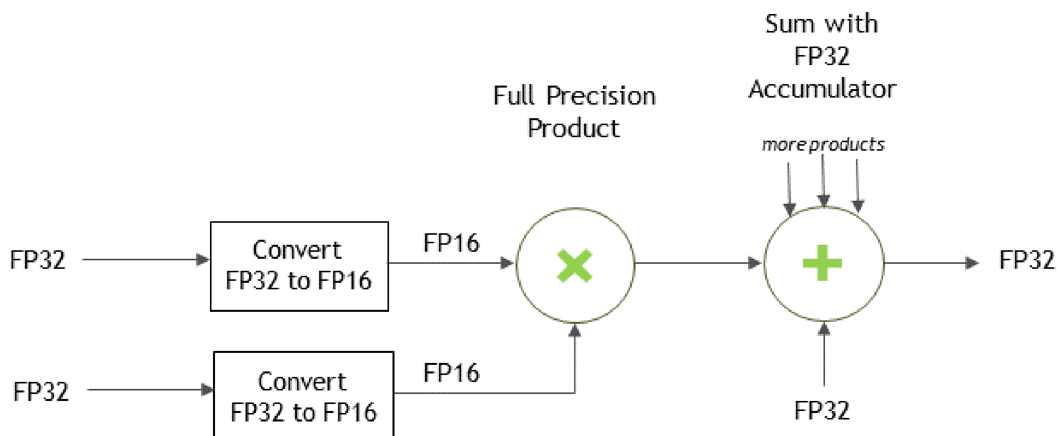



Figure 8 Tensor Operation with FP32 Inputs

**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. See the below code snippet:

```
// 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. See the below code snippet example:

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

## 2.9.4.3. 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};
```

### 2.9.4.4. 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).

### 2.9.4.5. 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. See also for packed NCHW data.

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.

## 2.9.5. 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.

# 2.10. GPU and driver requirements

cuDNN v7.0 supports NVIDIA GPUs of compute capability 3.0 and higher. For x86_64 platform, cuDNN v7.0 comes with two deliverables: one requires a NVIDIA Driver

compatible with CUDA Toolkit 8.0, the other requires a NVIDIA Driver compatible with CUDA Toolkit 9.0.

If you are using cuDNN with a Volta GPU, version 7 or later is required.

## 2.11. 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 compatibiliy with the following cuDNN release.

## 2.12. 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.

Note that 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]**

## 2.13. 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 varialbe **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 specificers: **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 1 for the impact on performance of API logging using environment variables.

Table 1 API Logging Using Environment Variables

| Environment variables | CUDNN_LOGINFO_DBG=0 | CUDNN_LOGINFO_DBG=1 |
|---|---|---|
| CUDNN_LOGDEST_DBG not set | - No logging output<br>- No performance loss | - No logging output<br>- No performance loss |
| CUDNN_LOGDEST_DBG=**NULL** | - No logging output<br>- No performance loss | - No logging output<br>- No performance loss |
| CUDNN_LOGDEST_DBG=**stdout** or **stderr** | - No logging output<br>- No performance loss | - Logging to **stdout** or **stderr**<br>- Some performance loss |
| CUDNN_LOGDEST_DBG=<br>**filename.txt** | - No logging output<br>- No performance loss | - Logging to **filename.txt**<br>- 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()**.

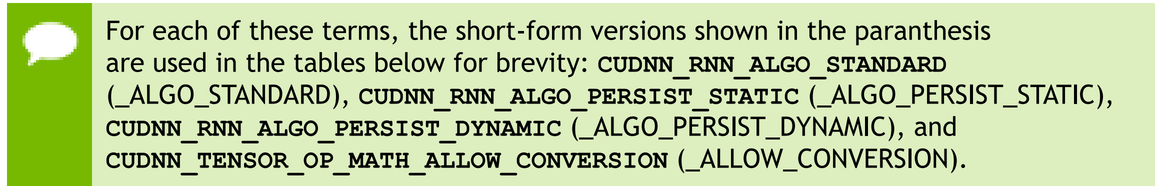
## 2.14. Features of RNN Functions

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 paranthesis 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**<br>Require input sequences descending sorted according to length | Mode (cell type) supported: **CUDNN_RNN_RELU, CUDNN_RNN_TANH, CUDNN_LSTM, CUDNN_GRU**<br>Algo supported* (see the table below for an elaboration on these algorithms): **_ALGO_STANDARD, _ALGO_PERSIST_STATIC, _ALGO_PERSIST_DYNAMIC**<br>Math mode supported: **CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH** (will automatically fall back if run on pre-Volta or if algo doesn't support Tensor Cores)<br>**_ALLOW_CONVERSION** (may do down conversion to utilize Tensor Cores)<br>Direction mode supported: **CUDNN_UNIDIRECTIONAL, CUDNN_BIDIRECTIONAL**<br>RNN input mode: **CUDNN_LINEAR_INPUT, CUDNN_SKIP_INPUT** |
| **cudnnRNNForwardTraining** | | | |
| **cudnnRNNBackwardData** | | | |
| **cudnnRNNBackwardWeights** | | | |
| **cudnnRNNForwardInferenceEx** | Sequence major unpacked,<br>Batch major unpacked**,<br>Sequence major packed** | Only with **_ALGO_STANDARD**<br>For unpacked layout**, no input sorting required.<br>For packed layout, require input sequences descending sorted according to length | |
| **cudnnRNNForwardTrainingEx** | | | |
| **cudnnRNNBackwardDataEx** | | | |
| **cudnnRNNBackwardWeightsEx** | | | |

* 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.

** To use unpacked layout, user need to set CUDNN_RNN_PADDED_IO_ENABLED through **`cudnnSetRNNPaddingMode`**.

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`**.

<table>
<tr><th>Features</th><th>_ALGO_STANDARD</th><th>_ALGO_PERSIST_STATIC</th><th>_ALGO_PERSIST_DYNAMIC</th></tr>
<tr><td>Half input<br>Single accumulation<br>Half output</td><td colspan="3">Supported<br>Half intermediate storage<br>Single accumulation</td></tr>
<tr><td>Single input<br>Single accumulation<br>Single output</td><td colspan="3">Supported<br>If running on Volta, with CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION', will down-convert and use half intermediate storage.<br>Otherwise: Single intermediate storage<br>Single accumulation</td></tr>
<tr><td>Double input<br>Double accumulation<br>Double output</td><td>Supported<br>Double intermediate storage<br>Double accumulation</td><td>Not Supported</td><td>Supported<br>Double intermediate storage<br>Double accumulation</td></tr>
<tr><td>LSTM recurrent projection</td><td>Supported</td><td>Not Supported</td><td>Not Supported</td></tr>
<tr><td>LSTM cell clipping</td><td colspan="3">Supported</td></tr>
<tr><td>Variable sequence length in batch</td><td>Supported</td><td>Not Supported</td><td>Not Supported</td></tr>
<tr><td>Tensor Cores on Volta/ Xavier</td><td colspan="2">Supported<br>For half input/output, acceleration requires setting<br>CUDNN_TENSOR_OP_MATH' or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION'<br>Acceleration requires inputSize and hiddenSize to be multiple of 8<br>For single input/output, acceleration requires setting<br>CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION'<br>Acceleration requires inputSize and hiddenSize to be multiple of 8</td><td>Not Supported, will execute normally ignoring CUDNN_TENSOR_OP_MATH' or<br>_ALLOW_CONVERSION'</td></tr>
</table>

| Other limitations | | Max problem size is limited by GPU specifications. | Requires real time compilation through NVRTC |
|---|---|---|---|

!**`CUDNN_TENSOR_OP_MATH`** or **`CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`** can be set through **`cudnnSetRNNMatrixMathType`**.

## 2.15. 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 additonal workspace to accumulate the intermediate values in FP32, i.e., full precision float.

# Chapter 3. CUDNN DATATYPES REFERENCE

This chapter describes all the types and enums of the cuDNN library API.

## 3.1. cudnnActivationDescriptor_t

**cudnnActivationDescriptor_t** is a pointer to an opaque structure holding the description of an activation operation. cudnnCreateActivationDescriptor is used to create one instance, and cudnnSetActivationDescriptor must be used to initialize this instance.

## 3.2. cudnnActivationMode_t

**cudnnActivationMode_t** is an enumerated type used to select the neuron activation function used in **cudnnActivationForward()**, **cudnnActivationBackward()** and **cudnnConvolutionBiasActivationForward()**.

**Values**

**CUDNN_ACTIVATION_SIGMOID**

Selects the sigmoid function.

**CUDNN_ACTIVATION_RELU**

Selects the rectified linear function.

**CUDNN_ACTIVATION_TANH**

Selects the hyperbolic tangent function.

**CUDNN_ACTIVATION_CLIPPED_RELU**

Selects the clipped rectified linear function.

**CUDNN_ACTIVATION_ELU**

Selects the exponential linear function.

**CUDNN_ACTIVATION_IDENTITY**

Selects the identity function, intended for bypassing the activation step in **cudnnConvolutionBiasActivationForward()**. (The **cudnnConvolutionBiasActivationForward()** function must use **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**.) Does not work with **cudnnActivationForward()** or **cudnnActivationBackward()**.

## 3.3. cudnnAttnDescriptor_t

**cudnnAttnDescriptor_t** is a pointer to an opaque structure holding parameters of the multi-head attention layer such as:

- weight and bias tensor shapes (vector lengths before and after linear projections)
- parameters that can be set in advance and do not change when invoking functions to evaluate forward responses and gradients (number of attention heads, softmax smoothing/sharpening coefficient)
- other settings that are necessary to compute temporary buffer sizes.

Use the cudnnCreateAttnDescriptor function to create an instance of the attention descriptor object and cudnnDestroyAttnDescriptor to delete the previously created descriptor. Use the cudnnSetAttnDescriptor function to configure the descriptor.

## 3.4. cudnnBatchNormMode_t

**cudnnBatchNormMode_t** is an enumerated type used to specify the mode of operation in cudnnBatchNormalizationForwardInference, cudnnBatchNormalizationForwardTraining, cudnnBatchNormalizationBackward and cudnnDeriveBNTensorDescriptor routines.

**Values**

**CUDNN_BATCHNORM_PER_ACTIVATION**

Normalization is performed per-activation. This mode is intended to be used after the non-convolutional network layers. In this mode, the tensor dimensions of **bnBias** and **bnScale** and the parameters used in the **cudnnBatchNormalization*** functions, are 1xCxHxW.

**CUDNN_BATCHNORM_SPATIAL**

Normalization is performed over N+spatial dimensions. This mode is intended for use after convolutional layers (where spatial invariance is desired). In this mode the **bnBias** and **bnScale** tensor dimensions are 1xCx1x1.

**CUDNN_BATCHNORM_SPATIAL_PERSISTENT**

This mode is similar to **CUDNN_BATCHNORM_SPATIAL** but it can be faster for some tasks.

An optimized path may be selected for **CUDNN_DATA_FLOAT** and **CUDNN_DATA_HALF** types, compute capability 6.0 or higher for the following two batch

normalization API calls: cudnnBatchNormalizationForwardTraining, and cudnnBatchNormalizationBackward. In the case of cudnnBatchNormalizationBackward, the **savedMean** and **savedInvVariance** arguments should not be **NULL**.

**The rest of this section applies to NCHW mode only:**

This mode may use a scaled atomic integer reduction that is deterministic but imposes more restrictions on the input data range. When a numerical overflow occurs, the algorithm may produce NaN-s or Inf-s (infinity) in output buffers.

When Inf-s/NaN-s are present in the input data, the output in this mode is the same as from a pure floating-point implementation.

For finite but very large input values, the algorithm may encounter overflows more frequently due to a lower dynamic range and emit Inf-s/NaN-s while **CUDNN_BATCHNORM_SPATIAL** will produce finite results. The user can invoke cudnnQueryRuntimeError to check if a numerical overflow occurred in this mode.

## 3.5. cudnnBatchNormOps_t

**cudnnBatchNormOps_t** is an enumerated type used to specify the mode of operation in **cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()**, **cudnnBatchNormalizationForwardTrainingEx()**, **cudnnGetBatchNormalizationBackwardExWorkspaceSize()**, **cudnnBatchNormalizationBackwardEx()**, and **cudnnGetBatchNormalizationTrainingExReserveSpaceSize()** functions.

**Values**

**CUDNN_BATCHNORM_OPS_BN**

Only batch normalization is performed, per-activation.

**CUDNN_BATCHNORM_OPS_BN_ACTIVATION**

First, the batch normalization is performed, and then the activation is performed.

**CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION**

Performs the batch normalization, then element-wise addition, followed by the activation operation.

## 3.6. cudnnConvolutionBwdDataAlgo_t

**cudnnConvolutionBwdDataAlgo_t** is an enumerated type that exposes the different algorithms available to execute the backward data convolution operation.

**Values**

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_0**

This algorithm expresses the convolution as a sum of matrix product without actually explicitly form the matrix that holds the input tensor data. The sum is done using atomic adds operation, thus the results are non-deterministic.

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_1**

This algorithm expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT**

This algorithm uses a Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING**

This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than **CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT** for large size images. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD**

This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED**

This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results. The results are deterministic.

## 3.7. cudnnConvolutionBwdDataAlgoPerf_t

**cudnnConvolutionBwdDataAlgoPerf_t** is a structure containing performance results returned by **cudnnFindConvolutionBackwardDataAlgorithm()** or heuristic results returned by **cudnnGetConvolutionBackwardDataAlgorithm_v7()**.

### Data Members

**cudnnConvolutionBwdDataAlgo_t algo**

The algorithm runs to obtain the associated performance metrics.

**cudnnStatus_t status**

If any error occurs during the workspace allocation or timing of **cudnnConvolutionBackwardData()**, this status will represent that error. Otherwise, this status will be the return status of **cudnnConvolutionBackwardData()**.

- **CUDNN_STATUS_ALLOC_FAILED** if any error occurred during workspace allocation or if the provided workspace is insufficient.

- **`CUDNN_STATUS_INTERNAL_ERROR`** if any error occurred during timing calculations or workspace deallocation.
- Otherwise, this will be the return status of **`cudnnConvolutionBackwardData()`**.

**`float time`**

The execution time of **`cudnnConvolutionBackwardData()`** (in milliseconds).

**`size_t memory`**

The workspace size (in bytes).

**`cudnnDeterminism_t determinism`**

The determinism of the algorithm.

**`cudnnMathType_t mathType`**

The math type provided to the algorithm.

**`int reserved[3]`**

Reserved space for future properties.

## 3.8. cudnnConvolutionBwdDataPreference_t

**`cudnnConvolutionBwdDataPreference_t`** is an enumerated type used by **`cudnnGetConvolutionBackwardDataAlgorithm()`** to help the choice of the algorithm used for the backward data convolution.

**Values**

**`CUDNN_CONVOLUTION_BWD_DATA_NO_WORKSPACE`**

In this configuration, the routine **`cudnnGetConvolutionBackwardDataAlgorithm()`** is guaranteed to return an algorithm that does not require any extra workspace to be provided by the user.

**`CUDNN_CONVOLUTION_BWD_DATA_PREFER_FASTEST`**

In this configuration, the routine **`cudnnGetConvolutionBackwardDataAlgorithm()`** will return the fastest algorithm regardless of how much workspace is needed to execute it.

**`CUDNN_CONVOLUTION_BWD_DATA_SPECIFY_WORKSPACE_LIMIT`**

In this configuration, the routine **`cudnnGetConvolutionBackwardDataAlgorithm()`** will return the fastest algorithm that fits within the memory limit that the user provided.

## 3.9. cudnnConvolutionBwdFilterAlgo_t

**`cudnnConvolutionBwdFilterAlgo_t`** is an enumerated type that exposes the different algorithms available to execute the backward filter convolution operation.

**Values**

**CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0**

This algorithm expresses the convolution as a sum of matrix product without actually explicitly form the matrix that holds the input tensor data. The sum is done using atomic adds operation, thus the results are non-deterministic.

**CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1**

This algorithm expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT**

This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant workspace is needed to store intermediate results. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3**

This algorithm is similar to **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0** but uses some small workspace to precomputes some indices. The results are also non-deterministic.

**CUDNN_CONVOLUTION_BWD_FILTER_WINOGRAD_NONFUSED**

This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results. The results are deterministic.

**CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING**

This algorithm uses the Fast-Fourier Transform approach to compute the convolution but splits the input tensor into tiles. A significant workspace may be needed to store intermediate results. The results are deterministic.

## 3.10. cudnnConvolutionBwdFilterAlgoPerf_t

**cudnnConvolutionBwdFilterAlgoPerf_t** is a structure containing performance results returned by **cudnnFindConvolutionBackwardFilterAlgorithm()** or heuristic results returned by **cudnnGetConvolutionBackwardFilterAlgorithm_v7()**.

### Data Members

**cudnnConvolutionBwdFilterAlgo_t algo**

The algorithm runs to obtain the associated performance metrics.

**cudnnStatus_t status**

If any error occurs during the workspace allocation or timing of **cudnnConvolutionBackwardFilter()**, this status will represent that error. Otherwise, this status will be the return status of **cudnnConvolutionBackwardFilter()**.

- **CUDNN_STATUS_ALLOC_FAILED** if any error occurred during workspace allocation or if the provided workspace is insufficient.

- **CUDNN_STATUS_INTERNAL_ERROR** if any error occurred during timing calculations or workspace deallocation.
- Otherwise, this will be the return status of **cudnnConvolutionBackwardFilter()**.

**float time**

The execution time of **cudnnConvolutionBackwardFilter()** (in milliseconds).

**size_t memory**

The workspace size (in bytes).

**cudnnDeterminism_t determinism**

The determinism of the algorithm.

**cudnnMathType_t mathType**

The math type provided to the algorithm.

**int reserved[3]**

Reserved space for future properties.

## 3.11. cudnnConvolutionBwdFilterPreference_t

**cudnnConvolutionBwdFilterPreference_t** is an enumerated type used by **cudnnGetConvolutionBackwardFilterAlgorithm()** to help the choice of the algorithm used for the backward filter convolution.

**Values**

**CUDNN_CONVOLUTION_BWD_FILTER_NO_WORKSPACE**

In this configuration, the routine **cudnnGetConvolutionBackwardFilterAlgorithm()** is guaranteed to return an algorithm that does not require any extra workspace to be provided by the user.

**CUDNN_CONVOLUTION_BWD_FILTER_PREFER_FASTEST**

In this configuration, the routine **cudnnGetConvolutionBackwardFilterAlgorithm()** will return the fastest algorithm regardless of how much workspace is needed to execute it.

**CUDNN_CONVOLUTION_BWD_FILTER_SPECIFY_WORKSPACE_LIMIT**

In this configuration, the routine **cudnnGetConvolutionBackwardFilterAlgorithm()** will return the fastest algorithm that fits within the memory limit that the user provided.

## 3.12. cudnnConvolutionDescriptor_t

**cudnnConvolutionDescriptor_t** is a pointer to an opaque structure holding the description of a convolution operation. **cudnnCreateConvolutionDescriptor()**

is used to create one instance, and **cudnnSetConvolutionNdDescriptor()** or **cudnnSetConvolution2dDescriptor()** must be used to initialize this instance.

## 3.13. cudnnConvolutionFwdAlgo_t

**cudnnConvolutionFwdAlgo_t** is an enumerated type that exposes the different algorithms available to execute the forward convolution operation.

**Values**

**CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM**

This algorithm expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data.

**CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**

This algorithm expresses convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.

**CUDNN_CONVOLUTION_FWD_ALGO_GEMM**

This algorithm expresses the convolution as an explicit matrix product. A significant memory workspace is needed to store the matrix that holds the input tensor data.

**CUDNN_CONVOLUTION_FWD_ALGO_DIRECT**

This algorithm expresses the convolution as a direct convolution (for example, without implicitly or explicitly doing a matrix multiplication).

**CUDNN_CONVOLUTION_FWD_ALGO_FFT**

This algorithm uses the Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results.

**CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING**

This algorithm uses the Fast-Fourier Transform approach but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than **CUDNN_CONVOLUTION_FWD_ALGO_FFT** for large size images.

**CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD**

This algorithm uses the Winograd Transform approach to compute the convolution. A reasonably sized workspace is needed to store intermediate results.

**CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED**

This algorithm uses the Winograd Transform approach to compute the convolution. A significant workspace may be needed to store intermediate results.

## 3.14. cudnnConvolutionFwdAlgoPerf_t

**cudnnConvolutionFwdAlgoPerf_t** is a structure containing performance results returned by **cudnnFindConvolutionForwardAlgorithm()** or heuristic results returned by **cudnnGetConvolutionForwardAlgorithm_v7()**.

**Data Members**

**cudnnConvolutionFwdAlgo_t algo**

The algorithm runs to obtain the associated performance metrics.

**cudnnStatus_t status**

If any error occurs during the workspace allocation or timing of **cudnnConvolutionForward()**, this status will represent that error. Otherwise, this status will be the return status of **cudnnConvolutionForward()**.

- **CUDNN_STATUS_ALLOC_FAILED** if any error occurred during workspace allocation or if the provided workspace is insufficient.
- **CUDNN_STATUS_INTERNAL_ERROR** if any error occurred during timing calculations or workspace deallocation.
- Otherwise, this will be the return status of **cudnnConvolutionForward()**.

**float time**

The execution time of **cudnnConvolutionForward()** (in milliseconds).

**size_t memory**

The workspace size (in bytes).

**cudnnDeterminism_t determinism**

The determinism of the algorithm.

**cudnnMathType_t mathType**

The math type provided to the algorithm.

**int reserved[3]**

Reserved space for future properties.

## 3.15. cudnnConvolutionFwdPreference_t

**cudnnConvolutionFwdPreference_t** is an enumerated type used by **cudnnGetConvolutionForwardAlgorithm()** to help the choice of the algorithm used for the forward convolution.

**Values**

**CUDNN_CONVOLUTION_FWD_NO_WORKSPACE**

In this configuration, the routine **cudnnGetConvolutionForwardAlgorithm()** is guaranteed to return an algorithm that does not require any extra workspace to be provided by the user.

**CUDNN_CONVOLUTION_FWD_PREFER_FASTEST**

In this configuration, the routine **cudnnGetConvolutionForwardAlgorithm()** will return the fastest algorithm regardless of how much workspace is needed to execute it.

**CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT**

In this configuration, the routine **cudnnGetConvolutionForwardAlgorithm()** will return the fastest algorithm that fits within the memory limit that the user provided.

## 3.16. cudnnConvolutionMode_t

**cudnnConvolutionMode_t** is an enumerated type used by **cudnnSetConvolutionDescriptor()** to configure a convolution descriptor. The filter used for the convolution can be applied in two different ways, corresponding mathematically to a convolution or to a cross-correlation. (A cross-correlation is equivalent to a convolution with its filter rotated by 180 degrees.)

**Values**

**CUDNN_CONVOLUTION**

In this mode, a convolution operation will be done when applying the filter to the images.

**CUDNN_CROSS_CORRELATION**

In this mode, a cross-correlation operation will be done when applying the filter to the images.

## 3.17. cudnnCTCLossAlgo_t

**cudnnCTCLossAlgo_t** is an enumerated type that exposes the different algorithms available to execute the CTC loss operation.

**Values**

**CUDNN_CTC_LOSS_ALGO_DETERMINISTIC**

Results are guaranteed to be reproducible

**CUDNN_CTC_LOSS_ALGO_NON_DETERMINISTIC**

Results are not guaranteed to be reproducible

## 3.18. cudnnCTCLossDescriptor_t

**cudnnCTCLossDescriptor_t** is a pointer to an opaque structure holding the description of a CTC loss operation. **cudnnCreateCTCLossDescriptor()** is used to create one instance, **cudnnSetCTCLossDescriptor()** is used to initialize this instance, and **cudnnDestroyCTCLossDescriptor()** is used to destroy this instance.

# 3.19. cudnnDataType_t

**cudnnDataType_t** is an enumerated type indicating the data type to which a tensor descriptor or filter descriptor refers.

**Values**

**CUDNN_DATA_FLOAT**

The data is a 32-bit single-precision floating-point (**float**).

**CUDNN_DATA_DOUBLE**

The data is a 64-bit double-precision floating-point (**double**).

**CUDNN_DATA_HALF**

The data is a 16-bit floating-point.

**CUDNN_DATA_INT8**

The data is an 8-bit signed integer.

**CUDNN_DATA_UINT8**

The data is an 8-bit unsigned integer.

**CUDNN_DATA_INT32**

The data is a 32-bit signed integer.

**CUDNN_DATA_INT8x4**

The data is 32-bit elements each composed of 4 8-bit signed integers. This data type is only supported with tensor format **CUDNN_TENSOR_NCHW_VECT_C**.

**CUDNN_DATA_INT8x32**

The data is 32-element vectors, each element being an 8-bit signed integer. This data type is only supported with the tensor format **CUDNN_TENSOR_NCHW_VECT_C**. Moreover, this data type can only be used with **algo 1**, meaning, **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**. For more information, see cudnnConvolutionFwdAlgo_t.

**CUDNN_DATA_UINT8x4**

The data is 32-bit elements each composed of 4 8-bit unsigned integers. This data type is only supported with tensor format **CUDNN_TENSOR_NCHW_VECT_C**.

# 3.20. cudnnDeterminism_t

**cudnnDeterminism_t** is an enumerated type used to indicate if the computed results are deterministic (reproducible). For more information, see Reproducibility (determinism).

**Values**

**CUDNN_NON_DETERMINISTIC**

Results are not guaranteed to be reproducible.

**CUDNN_DETERMINISTIC**

Results are guaranteed to be reproducible.

## 3.21. cudnnDirectionMode_t

**cudnnDirectionMode_t** is an enumerated type used to specify the recurrence pattern in the **cudnnRNNForwardInference()**, **cudnnRNNForwardTraining()**, **cudnnRNNBackwardData()** and **cudnnRNNBackwardWeights()** routines.

**Values**

**CUDNN_UNIDIRECTIONAL**

The network iterates recurrently from the first input to the last.

**CUDNN_BIDIRECTIONAL**

Each layer of the network iterates recurrently from the first input to the last and separately from the last input to the first. The outputs of the two are concatenated at each iteration giving the output of the layer.

## 3.22. cudnnDivNormMode_t

**cudnnDivNormMode_t** is an enumerated type used to specify the mode of operation in **cudnnDivisiveNormalizationForward()** and **cudnnDivisiveNormalizationBackward()**.

**Values**

**CUDNN_DIVNORM_PRECOMPUTED_MEANS**

The means tensor data pointer is expected to contain means or other kernel convolution values precomputed by the user. The means pointer can also be **NULL**, in that case, it's considered to be filled with zeroes. This is equivalent to spatial LRN.

In the backward pass, the means are treated as independent inputs and the gradient over means is computed independently. In this mode, to yield a net gradient over the entire LCN computational graph, the **destDiffMeans** result should be backpropagated through the user's means layer (which can be implemented using average pooling) and added to the **destDiffData** tensor produced by **cudnnDivisiveNormalizationBackward()**.

## 3.23. cudnnDropoutDescriptor_t

**cudnnDropoutDescriptor_t** is a pointer to an opaque structure holding the description of a dropout operation. **cudnnCreateDropoutDescriptor()** is used to create one instance, **cudnnSetDropoutDescriptor()** is used to initialize this

instance, **cudnnDestroyDropoutDescriptor()** is used to destroy this instance, **cudnnGetDropoutDescriptor()** is used to query fields of a previously initialized instance, **cudnnRestoreDropoutDescriptor()** is used to restore an instance to a previously saved off state.

## 3.24. cudnnErrQueryMode_t

**cudnnErrQueryMode_t** is an enumerated type passed to **cudnnQueryRuntimeError()** to select the remote kernel error query mode.

**Values**

**CUDNN_ERRQUERY_RAWCODE**
Read the error storage location regardless of the kernel completion status.

**CUDNN_ERRQUERY_NONBLOCKING**
Report if all tasks in the user stream of the cuDNN handle were completed. If that is the case, report the remote kernel error code.

**CUDNN_ERRQUERY_BLOCKING**
Wait for all tasks to complete in the user stream before reporting the remote kernel error code.

## 3.25. cudnnFilterDescriptor_t

**cudnnFilterDescriptor_t** is a pointer to an opaque structure holding the description of a filter dataset. **cudnnCreateFilterDescriptor()** is used to create one instance, and **cudnnSetFilter4dDescriptor()** or **cudnnSetFilterNdDescriptor()** must be used to initialize this instance.

## 3.26. cudnnFoldingDirection_t

**cudnnFoldingDirection_t** is an enumerated type used to select the folding direction. For more information, see cudnnTensorTransformDescriptor_t.

| Member | Description |
|---|---|
| CUDNN_TRANSFORM_FOLD = 0U | Selects folding. |
| CUDNN_TRANSFORM_UNFOLD = 1U | Selects unfolding. |

## 3.27. cudnnFusedOps_t

The **cudnnFusedOps_t** type is an enumerated type to select a specific sequence of computations to perform in the fused operations.

| Member | Description |
|---|---|
| `CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS = 0` | On a per-channel basis, performs these operations in this order: scale, add bias, activation, convolution, and generate `batchnorm` statistics. |
| | |
| `CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD = 1` | On a per-channel basis, performs these operations in this order: scale, add bias, activation, convolution backward weights, and generate `batchnorm` statistics. |
| CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD<br>Output<br>Input<br>x<br>equivalent scale<br>equivalent bias<br>dw<br>wgrad<br>$y_1$<br>ReLU<br>$y_0$<br>Scale & Bias<br>$y_1 = ReLU(y_0)$<br>$y_0 = scale(x)+bias$<br>dy | |
| `CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING = 2` | Computes the equivalent scale and bias from `ySum`, `ySqSum` and learned `scale`, `bias`.<br>Optionally update running statistics and generate saved stats |
| `CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE = 3` | Computes the equivalent scale and bias from the learned running statistics and the learned scale, bias. |
| `CUDNN_FUSED_CONV_SCALE_BIAS_ADD_ACTIVATION = 4` | On a per-channel basis, performs these operations in this order: convolution, scale, add bias, element-wise addition with another tensor, and activation. |
| `CUDNN_FUSED_SCALE_BIAS_ADD_ACTIVATION_GEN_BITMASK = 5` | On a per-channel basis, performs these operations in this order: scale and bias on one tensor, scale, and bias on a second tensor, element-wise addition of these two tensors, and on the resulting tensor perform activation, and generate activation bit mask. |
| `CUDNN_FUSED_DACTIVATION_FORK_DBATCHNORM = 6` | On a per-channel basis, performs these operations in this order: backward activation, fork (meaning, write out gradient for the residual branch), and backward batch norm. |

# 3.28. cudnnFusedOpsConstParamLabel_t

The **cudnnFusedOpsConstParamLabel_t** is an enumerated type for the selection of the type of the **cudnnFusedOps** descriptor. For more information, see cudnnSetFusedOpsConstParamPackAttribute.

```
typedef enum {
 CUDNN_PARAM_XDESC                          = 0,
 CUDNN_PARAM_XDATA_PLACEHOLDER              = 1,
 CUDNN_PARAM_BN_MODE                        = 2,
 CUDNN_PARAM_BN_EQSCALEBIAS_DESC            = 3,
 CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER         = 4,
 CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER          = 5,
 CUDNN_PARAM_ACTIVATION_DESC                = 6,
 CUDNN_PARAM_CONV_DESC                      = 7,
 CUDNN_PARAM_WDESC                          = 8,
 CUDNN_PARAM_WDATA_PLACEHOLDER              = 9,
 CUDNN_PARAM_DWDESC                         = 10,
 CUDNN_PARAM_DWDATA_PLACEHOLDER             = 11,
 CUDNN_PARAM_YDESC                          = 12,
 CUDNN_PARAM_YDATA_PLACEHOLDER              = 13,
 CUDNN_PARAM_DYDESC                         = 14,
 CUDNN_PARAM_DYDATA_PLACEHOLDER             = 15,
 CUDNN_PARAM_YSTATS_DESC                    = 16,
 CUDNN_PARAM_YSUM_PLACEHOLDER               = 17,
 CUDNN_PARAM_YSQSUM_PLACEHOLDER             = 18,
 CUDNN_PARAM_BN_SCALEBIAS_MEANVAR_DESC      = 19,
 CUDNN_PARAM_BN_SCALE_PLACEHOLDER           = 20,
 CUDNN_PARAM_BN_BIAS_PLACEHOLDER            = 21,
 CUDNN_PARAM_BN_SAVED_MEAN_PLACEHOLDER      = 22,
 CUDNN_PARAM_BN_SAVED_INVSTD_PLACEHOLDER    = 23,
 CUDNN_PARAM_BN_RUNNING_MEAN_PLACEHOLDER    = 24,
 CUDNN_PARAM_BN_RUNNING_VAR_PLACEHOLDER     = 25,
 CUDNN_PARAM_ZDESC                          = 26,
 CUDNN_PARAM_ZDATA_PLACEHOLDER              = 27,
 CUDNN_PARAM_BN_Z_EQSCALEBIAS_DESC          = 28,
 CUDNN_PARAM_BN_Z_EQSCALE_PLACEHOLDER       = 29,
 CUDNN_PARAM_BN_Z_EQBIAS_PLACEHOLDER        = 30,
 CUDNN_PARAM_ACTIVATION_BITMASK_DESC        = 31,
 CUDNN_PARAM_ACTIVATION_BITMASK_PLACEHOLDER = 32,
 CUDNN_PARAM_DXDESC                         = 33,
 CUDNN_PARAM_DXDATA_PLACEHOLDER             = 34,
 CUDNN_PARAM_DZDESC                         = 35,
 CUDNN_PARAM_DZDATA_PLACEHOLDER             = 36,
 CUDNN_PARAM_BN_DSCALE_PLACEHOLDER          = 37,
 CUDNN_PARAM_BN_DBIAS_PLACEHOLDER           = 38,
 } cudnnFusedOpsConstParamLabel_t;
```

| Short-form used | Stands for |
|---|---|
| Setter | cudnnSetFusedOpsConstParamPackAttribute |
| Getter | cudnnGetFusedOpsConstParamPackAttribute |
| **X_PointerPlaceHolder_t** | cudnnFusedOpsPointerPlaceHolder_t |
| **X_** prefix in the *Attribute* column | Stands for **CUDNN_PARAM_** in the enumerator name |

Table 2 **CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS**

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in cudnnFusedOp_t | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| X_XDESC | In the setter, the *param should be xDesc, a pointer to a previously initialized cudnnTensorDescriptor_t. | Tensor descriptor describing the size, layout, and datatype of the x (input) tensor. | NULL |
| X_XDATA_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t*. | Describes whether xData pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. | CUDNN_PTR_NULL |
| X_BN_MODE | In the setter, the *param should be a pointer to a previously initialized cudnnBatchNormMode_t*. | Describes the mode of operation for the scale, bias and the statistics.<br>As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias, and statistics are all per-channel. | CUDNN_BATCHNORM_PER_ACTIVATION |
| X_BN_EQSCALEBIAS_DESC | In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t. | Tensor descriptor describing the size, layout, and datatype of the batchNorm equivalent scale and bias tensors. The shapes must match the mode specified in CUDNN_PARAM_BN_MODE. If set to NULL, both scale and bias operation will become a NOP. | NULL |
| X_BN_EQSCALE_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t*. | Describes whether batchnorm equivalent scale pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, then the scale operation becomes a NOP. | CUDNN_PTR_NULL |

| For the attribute **CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS** in **cudnnFusedOp_t** | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| **X_BN_EQBIAS_PLACEHOLDER** | In the setter, the ***param** should be a pointer to a previously initialized **X_PointerPlaceHolder_t***. | Describes whether batchnorm equivalent bias pointer in the **VariantParamPack** will be **NULL**, or if not, user promised pointer alignment *****.<br>If set to **CUDNN_PTR_NULL**, then the bias operation becomes a NOP. | **CUDNN_PTR_NULL** |
| **X_ACTIVATION_DESC** | In the setter, the ***param** should be a pointer to a previously initialized cudnnActivationDescriptor_t*. | Describes the activation operation.<br>As of 7.6.0, only activation mode of **CUDNN_ACTIVATION_RELU** and **CUDNN_ACTIVATION_IDENTITY** are supported. If set to **NULL** or if the activation mode is set to **CUDNN_ACTIVATION_IDENTITY**, then the activation in the op sequence becomes a NOP. | **NULL** |
| **X_CONV_DESC** | In the setter, the ***param** should be a pointer to a previously initialized cudnnConvolutionDescriptor_t*. | Describes the convolution operation. | **NULL** |
| **X_WDESC** | In the setter, the ***param** should be a pointer to a previously initialized cudnnFilterDescriptor_t*. | Filter descriptor describing the size, layout and datatype of the **w** (filter) tensor. | **NULL** |
| **X_WDATA_PLACEHOLDER** | In the setter, the ***param** should be a pointer to a previously initialized **X_PointerPlaceHolder_t***. | Describes whether **w** (filter) tensor pointer in the **VariantParamPack** will be **NULL**, or if not, user promised pointer alignment *****. | **CUDNN_PTR_NULL** |
| **X_YDESC** | In the setter, the ***param** should be a pointer to a previously initialized cudnnTensorDescriptor_t*. | Tensor descriptor describing the size, layout and datatype of the **y** (output) tensor. | **NULL** |

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in cudnnFusedOp_t | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| X_YDATA_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether y (output) tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. | CUDNN_PTR_NULL |
| X_YSTATS_DESC | In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t | Tensor descriptor describing the size, layout and datatype of the sum of y and sum of y square tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE.<br>If set to NULL, the y statistics generation operation will be become a NOP. | NULL |
| X_YSUM_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether sum of y pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, the y statistics generation operation will be become a NOP. | CUDNN_PTR_NULL |
| X_YSQSUM_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether sum of y square pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, the y statistics generation operation will be become a NOP. | CUDNN_PTR_NULL |

- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_NULL, then the device pointer in the VariantParamPack need to be NULL as well.
- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_ELEM_ALIGNED or CUDNN_PTR_16B_ALIGNED, then the device

pointer in the **VariantParamPack** may not be **NULL** and need to be at least element-aligned or 16 bytes-aligned, respectively.

As of cuDNN 7.6.0, if the conditions in Table 3 are met, then the fully fused fast path will be triggered. Otherwise, a slower partially fused path will be triggered.

## Table 3 Conditions for Fully Fused Fast Path (Forward)

| Parameter | Condition |
|---|---|
| Device compute capability | Need to be one of **7.0**, **7.2** or **7.5**. |
| **CUDNN_PARAM_XDESC**<br>**CUDNN_PARAM_XDATA_PLACEHOLDER** | Tensor is 4 dimensional<br>Datatype is **CUDNN_DATA_HALF**<br>Layout is **NHWC** fully packed<br>Alignment is **CUDNN_PTR_16B_ALIGNED**<br>Tensor's **C** dimension is a multiple of 8. |
| **CUDNN_PARAM_BN_EQSCALEBIAS_DESC**<br>**CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER**<br>**CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER** | If either one of scale and bias operation is not turned into a NOP:<br>Tensor is 4 dimensional with shape 1xCx1x1<br>Datatype is **CUDNN_DATA_HALF**<br>Layout is fully packed<br>Alignment is **CUDNN_PTR_16B_ALIGNED** |
| **CUDNN_PARAM_CONV_DESC**<br>**CUDNN_PARAM_WDESC**<br>**CUDNN_PARAM_WDATA_PLACEHOLDER** | Convolution descriptor's mode needs to be **CUDNN_CROSS_CORRELATION**.<br>Convolution descriptor's **dataType** needs to be **CUDNN_DATA_FLOAT**.<br>Convolution descriptor's **dilationA** is (1,1).<br>Convolution descriptor's group count needs to be **1**.<br>Convolution descriptor's **mathType** needs to be **CUDNN_TENSOR_OP_MATH** or **CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION**.<br>Filter is in **NHWC** layout<br>Filter's data type is **CUDNN_DATA_HALF**<br>Filter's K dimension is a multiple of 32<br>Filter size RxS is either 1x1 or 3x3<br>If filter size RxS is 1x1, convolution descriptor's **padA** needs to be (0,0) and **filterStrideA** needs to be (1,1).<br>Filter's alignment is **CUDNN_PTR_16B_ALIGNED** |
| **CUDNN_PARAM_YDESC**<br>**CUDNN_PARAM_YDATA_PLACEHOLDER** | Tensor is 4 dimensional<br>Datatype is **CUDNN_DATA_HALF**<br>Layout is **NHWC** fully packed<br>Alignment is **CUDNN_PTR_16B_ALIGNED** |

| Parameter | Condition |
| --- | --- |
| CUDNN_PARAM_YSTATS_DESC<br>CUDNN_PARAM_YSUM_PLACEHOLDER<br>CUDNN_PARAM_YSQSUM_PLACEHOLDER | If the generate statistics operation is not turned into a NOP:<br>Tensor is 4 dimensional with shape 1xKx1x1<br>Datatype is CUDNN_DATA_FLOAT<br>Layout is fully packed<br>Alignment is CUDNN_PTR_16B_ALIGNED |

## Table 4 CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in cudnnFusedOp_t | | | |
| --- | --- | --- | --- |
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| X_XDESC | In the setter, the *param should be xDesc, a pointer to a previously initialized cudnnTensorDescriptor_t. | Tensor descriptor describing the size, layout and datatype of the x (input) tensor | NULL |
| X_XDATA_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether xData pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. | CUDNN_PTR_NULL |
| X_BN_MODE | In the setter, the *param should be a pointer to a previously initialized cudnnBatchNormMode_t*. | Describes the mode of operation for the scale, bias and the statistics.<br>As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias, and statistics are all per-channel. | CUDNN_BATCHNORM_PER_ACTIVATION |
| X_BN_EQSCALEBIAS_DESC | In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t. | Tensor descriptor describing the size, layout and datatype of the batchNorm equivalent scale and bias tensors. The shapes must match the mode specified in CUDNN_PARAM_BN_MODE. If set to NULL, both scale and bias | NULL |

| For the attribute `CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD` in `cudnnFusedOp_t` | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| | | operation will become a NOP. | |
| `X_BN_EQSCALE_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether batchnorm equivalent scale pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`.<br>If set to `CUDNN_PTR_NULL`, then the scale operation becomes a NOP. | `CUDNN_PTR_NULL` |
| `X_BN_EQBIAS_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether batchnorm equivalent bias pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`.<br>If set to `CUDNN_PTR_NULL`, then the bias operation becomes a NOP. | `CUDNN_PTR_NULL` |
| `X_ACTIVATION_DESC` | In the setter, the `*param` should be a pointer to a previously initialized cudnnActivationDescriptor_t*. | Describes the activation operation.<br>As of 7.6.0, only activation mode of `CUDNN_ACTIVATION_RELU` and `CUDNN_ACTIVATION_IDENTITY` is supported. If set to `NULL` or if the activation mode is set to `CUDNN_ACTIVATION_IDENTITY`, then the activation in the op sequence becomes a NOP. | `NULL` |
| `X_CONV_DESC` | In the setter, the `*param` should be a pointer to a previously initialized cudnnConvolutionDescriptor_t*. | Describes the convolution operation. | `NULL` |
| `X_DWDESC` | In the setter, the `*param` should be a pointer to a | Filter descriptor describing the size, layout and datatype of | `NULL` |

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in cudnnFusedOp_t | | | |
|---|---|---|---|
| Attribute | Expected Descriptor Type Passed in, in the Setter | Description | Default Value After Creation |
| | previously initialized cudnnFilterDescriptor_t* | the dw (filter gradient output) tensor. | |
| X_DWDATA_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether dw (filter gradient output) tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. | CUDNN_PTR_NULL |
| X_DYDESC | In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t* | Tensor descriptor describing the size, layout and datatype of the dy (gradient input) tensor. | NULL |
| X_DYDATA_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether dy (gradient input) tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. | CUDNN_PTR_NULL |

- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_NULL, then the device pointer in the VariantParamPack needs to be NULL as well.
- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_ELEM_ALIGNED or CUDNN_PTR_16B_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

As of cuDNN 7.6.0, if the conditions in Table 5 are met, then the fully fused fast path will be triggered. Otherwise a slower partially fused path will be triggered.

## Table 5 Conditions for Fully Fused Fast Path (Backward)

| Parameter | Condition |
|---|---|
| Device compute capability | Needs to be one of 7.0, 7.2 or 7.5. |
| CUDNN_PARAM_XDESC<br>CUDNN_PARAM_XDATA_PLACEHOLDER | Tensor is 4 dimensional<br>Datatype is CUDNN_DATA_HALF<br>Layout is NHWC fully packed<br>Alignment is CUDNN_PTR_16B_ALIGNED<br>Tensor's C dimension is a multiple of 8. |

| Parameter | Condition |
|---|---|
| CUDNN_PARAM_BN_EQSCALEBIAS_DESC<br>CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER<br>CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER | If either one of scale and bias operation is not turned into a NOP:<br>Tensor is 4 dimensional with shape 1xCx1x1<br>Datatype is CUDNN_DATA_HALF<br>Layout is fully packed<br>Alignment is CUDNN_PTR_16B_ALIGNED |
| CUDNN_PARAM_CONV_DESC<br>CUDNN_PARAM_DWDESC<br>CUDNN_PARAM_DWDATA_PLACEHOLDER | Convolution descriptor's mode needs to be CUDNN_CROSS_CORRELATION.<br>Convolution descriptor's dataType needs to be CUDNN_DATA_FLOAT.<br>Convolution descriptor's dilationA is (1,1)<br>Convolution descriptor's group count needs to be 1.<br>Convolution descriptor's mathType needs to be CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION.<br>Filter gradient is in NHWC layout<br>Filter gradient's data type is CUDNN_DATA_HALF<br>Filter gradient's K dimension is a multiple of 32.<br>Filter gradient size RxS is either 1x1 or 3x3<br>If filter gradient size RxS is 1x1, convolution descriptor's padA needs to be (0,0) and filterStrideA needs to be (1,1).<br>Filter gradient's alignment is CUDNN_PTR_16B_ALIGNED |
| CUDNN_PARAM_DYDESC<br>CUDNN_PARAM_DYDATA_PLACEHOLDER | Tensor is 4 dimensional<br>Datatype is CUDNN_DATA_HALF<br>Layout is NHWC fully packed<br>Alignment is CUDNN_PTR_16B_ALIGNED |

## Table 6 CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | |
|---|---|---|---|
| Attribute | Expected Descriptor Type Passed in, in the Setter | Description | Default Value After Creation |
| X_BN_MODE | In the setter, the *param should be a pointer to a previously initialized cudnnBatchNormMode_t*. | Describes the mode of operation for the scale, bias and the statistics.<br>As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL | CUDNN_BATCHNORM_PER_ACTIVATION |

| For the attribute `CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING` in `cudnnFusedOp_t` | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| | | and `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` are supported, meaning, scale, bias and statistics are all per-channel. | |
| `X_YSTATS_DESC` | In the setter, the `*param` should be a pointer to a previously initialized `cudnnTensorDescriptor_t*`. | Tensor descriptor describing the size, layout and datatype of the sum of `y` and sum of `y` square tensors. The shapes need to match the mode specified in `CUDNN_PARAM_BN_MODE`. | `NULL` |
| `X_YSUM_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether sum of `y` pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_YSQSUM_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether sum of `y` square pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_BN_SCALEBIAS_MEANVAR_DESC` | In the setter, the `*param` should be a pointer to a previously initialized cudnnTensorDescriptor_t | A common tensor descriptor describing the size, layout and datatype of the batchNorm trained scale, bias and statistics tensors. The shapes need to match the mode specified in `CUDNN_PARAM_BN_MODE` (similar to the `bnScaleBiasMeanVarDesc` field in the `cudnnBatchNormalization*` API). | `NULL` |
| `X_BN_SCALE_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether the batchNorm trained scale pointer in the `VariantParamPack` will be `NULL`, or if not, | `CUDNN_PTR_NULL` |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | |
|---|---|---|---|
| Attribute | Expected Descriptor Type Passed in, in the Setter | Description | Default Value After Creation |
| | | user promised pointer alignment *.<br>If the output of BN_EQSCALE is not needed, then this is not needed and may be NULL. | |
| X_BN_BIAS_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t*. | Describes whether the batchNorm trained bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If neither output of BN_EQSCALE or BN_EQBIAS is needed, then this is not needed and may be NULL. | CUDNN_PTR_NULL |
| X_BN_SAVED_MEAN_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t*. | Describes whether the batchNorm saved mean pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, then the computation for this output becomes a NOP. | CUDNN_PTR_NULL |
| X_BN_SAVED_INVSTD_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t*. | Describes whether the batchNorm saved inverse standard deviation pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, then the computation for this output becomes a NOP. | CUDNN_PTR_NULL |
| X_BN_RUNNING_MEAN_PLACEHOLDER | In the setter, the *param should be a pointer to a | Describes whether the batchNorm running mean pointer in the | CUDNN_PTR_NULL |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| | previously initialized `X_PointerPlaceHolder_t*`. | `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`.<br>If set to `CUDNN_PTR_NULL`, then the computation for this output becomes a NOP. | |
| `X_BN_RUNNING_VAR_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether the batchNorm running variance pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`.<br>If set to `CUDNN_PTR_NULL`, then the computation for this output becomes a NOP. | `CUDNN_PTR_NULL` |
| `X_BN_EQSCALEBIAS_DESC` | In the setter, the `*param` should be a pointer to a previously initialized cudnnTensorDescriptor_t*. | Tensor descriptor describing the size, layout and datatype of the batchNorm equivalent scale and bias tensors. The shapes need to match the mode specified in `CUDNN_PARAM_BN_MODE`.<br>If neither output of `BN_EQSCALE` or `BN_EQBIAS` is needed, then this is not needed and may be `NULL`. | `NULL` |
| `X_BN_EQSCALE_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*`. | Describes whether batchnorm equivalent scale pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`.<br>If set to `CUDNN_PTR_NULL`, then the computation for this output becomes a NOP. | `CUDNN_PTR_NULL` |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| X_BN_EQBIAS_PLACEHOLDER | In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t* | Describes whether batchnorm equivalent bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.<br>If set to CUDNN_PTR_NULL, then the computation for this output becomes a NOP. | CUDNN_PTR_NULL |

## Table 7 CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOp_t | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| X_BN_MODE | In the setter, the *param should be a pointer to a previously initialized cudnnBatchNormMode_t*. | Describes the mode of operation for the scale, bias and the statistics.<br>As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias and statistics are all per-channel. | CUDNN_BATCHNORM_PER_ACTIVATION |
| X_BN_SCALEBIAS_MEANVAR_DESC | In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t | A common tensor descriptor describing the size, layout and datatype of the batchNorm trained scale, bias and statistics tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE (similar to the bnScaleBiasMeanVarDesc field in the | NULL |

| For the attribute `CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE` in `cudnnFusedOp_t` | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| | | `cudnnBatchNormalization*` API). | |
| `X_BN_SCALE_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*` | Describes whether the batchNorm trained scale pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_BN_BIAS_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*` | Describes whether the batchNorm trained bias pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_BN_RUNNING_MEAN_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*` | Describes whether the batchNorm running mean pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_BN_RUNNING_VAR_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*` | Describes whether the batchNorm running variance pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |
| `X_BN_EQSCALEBIAS_DESC` | In the setter, the `*param` should be a pointer to a previously initialized cudnnTensorDescriptor_t* | Tensor descriptor describing the size, layout and datatype of the batchNorm equivalent scale and bias tensors. The shapes need to match the mode specified in `CUDNN_PARAM_BN_MODE`. | `NULL` |
| `X_BN_EQSCALE_PLACEHOLDER` | In the setter, the `*param` should be a pointer to a previously initialized `X_PointerPlaceHolder_t*` | Describes whether batchnorm equivalent scale pointer in the `VariantParamPack` will be `NULL`, or if not, user promised pointer alignment `*`. | `CUDNN_PTR_NULL` |

| For the attribute **CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE** in **cudnnFusedOp_t** | | | |
|---|---|---|---|
| **Attribute** | **Expected Descriptor Type Passed in, in the Setter** | **Description** | **Default Value After Creation** |
| | | If set to **CUDNN_PTR_NULL**, then the computation for this output becomes a NOP. | |
| **X_BN_EQBIAS_PLACEHOLDER** | In the setter, the ***param** should be a pointer to a previously initialized **X_PointerPlaceHolder_t*** | Describes whether batchnorm equivalent bias pointer in the **VariantParamPack** will be **NULL**, or if not, user promised pointer alignment *****. If set to **CUDNN_PTR_NULL**, then the computation for this output becomes a NOP. | **CUDNN_PTR_NULL** |

## 3.29. cudnnFusedOpsConstParamPack_t

**cudnnFusedOpsConstParamPack_t** is a pointer to an opaque structure holding the description of the **cudnnFusedOps** constant parameters. Use the function cudnnCreateFusedOpsConstParamPack to create one instance of this structure, and the function cudnnDestroyFusedOpsConstParamPack to destroy a previously-created descriptor.

## 3.30. cudnnFusedOpsPlan_t

**cudnnFusedOpsPlan_t** is a pointer to an opaque structure holding the description of the **cudnnFusedOpsPlan**. This descriptor contains the plan information, including the problem type and size, which kernels should be run, and the internal workspace partition. Use the function cudnnCreateFusedOpsPlan to create one instance of this structure, and the function cudnnDestroyFusedOpsPlan to destroy a previously-created descriptor.

## 3.31. cudnnFusedOpsPointerPlaceHolder_t

cudnnFusedOpsPointerPlaceHolder_t is an enumerated type used to select the alignment type of the **cudnnFusedOps** descriptor pointer.

| Member | Description |
| --- | --- |
| **CUDNN_PTR_NULL = 0** | Indicates that the pointer to the tensor in the **variantPack** will be **NULL**. |
| **CUDNN_PTR_ELEM_ALIGNED = 1** | Indicates that the pointer to the tensor in the **variantPack** will not be **NULL**, and will have element alignment. |
| **CUDNN_PTR_16B_ALIGNED = 2** | Indicates that the pointer to the tensor in the **variantPack** will not be **NULL**, and will have 16 byte alignment. |

# 3.32. cudnnFusedOpsVariantParamLabel_t

The **cudnnFusedOpsVariantParamLabel_t** is an enumerated type that is used to set the buffer pointers. These buffer pointers can be changed in each iteration.

```
typedef enum {
 CUDNN_PTR_XDATA                              = 0,
 CUDNN_PTR_BN_EQSCALE                         = 1,
 CUDNN_PTR_BN_EQBIAS                          = 2,
 CUDNN_PTR_WDATA                              = 3,
 CUDNN_PTR_DWDATA                             = 4,
 CUDNN_PTR_YDATA                              = 5,
 CUDNN_PTR_DYDATA                             = 6,
 CUDNN_PTR_YSUM                               = 7,
 CUDNN_PTR_YSQSUM                             = 8,
 CUDNN_PTR_WORKSPACE                          = 9,
 CUDNN_PTR_BN_SCALE                           = 10,
 CUDNN_PTR_BN_BIAS                            = 11,
 CUDNN_PTR_BN_SAVED_MEAN                      = 12,
 CUDNN_PTR_BN_SAVED_INVSTD                    = 13,
 CUDNN_PTR_BN_RUNNING_MEAN                    = 14,
 CUDNN_PTR_BN_RUNNING_VAR                     = 15,
 CUDNN_PTR_ZDATA                              = 16,
 CUDNN_PTR_BN_Z_EQSCALE                       = 17,
 CUDNN_PTR_BN_Z_EQBIAS                        = 18,
 CUDNN_PTR_ACTIVATION_BITMASK                 = 19,
 CUDNN_PTR_DXDATA                             = 20,
 CUDNN_PTR_DZDATA                             = 21,
 CUDNN_PTR_BN_DSCALE                          = 22,
 CUDNN_PTR_BN_DBIAS                           = 23,
 CUDNN_SCALAR_SIZE_T_WORKSPACE_SIZE_IN_BYTES  = 100,
 CUDNN_SCALAR_INT64_T_BN_ACCUMULATION_COUNT   = 101,
 CUDNN_SCALAR_DOUBLE_BN_EXP_AVG_FACTOR        = 102,
 CUDNN_SCALAR_DOUBLE_BN_EPSILON               = 103,
 } cudnnFusedOpsVariantParamLabel_t;
```

## Table 8 Legend For Tables in This Section

| Short-form used | Stands for |
| --- | --- |
| Setter | cudnnSetFusedOpsVariantParamPackAttribute |
| Getter | cudnnGetFusedOpsVariantParamPackAttribute |
| **x_** prefix in the **Attribute key** column | Stands for **CUDNN_PTR_** or **CUDNN_SCALAR_** in the enumerator name. |

Table 9 **CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS**

| For the attribute `CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS` in `cudnnFusedOp_t` | | | | |
|---|---|---|---|---|
| **Attribute key** | **Expected Descriptor Type Passed in, in the Setter** | **I/O Type** | **Description** | **Default Value** |
| `X_XDATA` | `void *` | input | Pointer to `x` (input) tensor on device, need to agree with previously set `CUDNN_PARAM_XDATA_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_BN_EQSCALE` | `void *` | input | Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set `CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_BN_EQBIAS` | `void *` | input | Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set `CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_WDATA` | `void *` | input | Pointer to `w` (filter) tensor on device, need to agree with previously set `CUDNN_PARAM_WDATA_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_YDATA` | `void *` | output | Pointer to `y` (output) tensor on device, need to agree with previously set `CUDNN_PARAM_YDATA_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_YSUM` | `void *` | output | Pointer to sum of `y` tensor on device, need to agree with previously set `CUDNN_PARAM_YSUM_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_YSQSUM` | `void *` | output | Pointer to sum of `y` square tensor on device, need to agree with previously set `CUDNN_PARAM_YSQSUM_PLACEHOLDER` attribute `*`. | `NULL` |
| `X_WORKSPACE` | `void *` | input | Pointer to user allocated workspace on device. Can be `NULL` if the workspace size requested is `0`. | `NULL` |
| `X_SIZE_T_WORKSPACE_SIZE_IN_BYTES` | `size_t *` | input | Pointer to a `size_t` value in host memory describing | `0` |

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| Attribute key | Expected Descriptor Type Passed in, in the Setter | I/O Type | Description | Default Value |
| | | | the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan. | |

- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_NULL, then the device pointer in the VariantParamPack needs to be NULL as well
- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_ELEM_ALIGNED or CUDNN_PTR_16B_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 10 CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| Attribute key | Expected Descriptor Type Passed in, in the Setter | I/O Type | Description | Default Value |
| X_XDATA | void * | input | Pointer to x (input) tensor on device, need to agree with previously set CUDNN_PARAM_XDATA_PLACEHOLDER attribute *. | NULL |
| X_BN_EQSCALE | void * | input | Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *. | NULL |
| X_BN_EQBIAS | void * | input | Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *. | NULL |

| For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| Attribute key | Expected Descriptor Type Passed in, in the Setter | I/O Type | Description | Default Value |
| X_DWDATA | void * | output | Pointer to dw (filter gradient output) tensor on device, need to agree with previously set CUDNN_PARAM_WDATA_PLACEHOLDER attribute *. | NULL |
| X_DYDATA | void * | input | Pointer to dy (gradient input) tensor on device, need to agree with previously set CUDNN_PARAM_YDATA_PLACEHOLDER attribute *. | NULL |
| X_WORKSPACE | void * | input | Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0. | NULL |
| X_SIZE_T_WORKSPACE_SIZE_IN_BYTES | size_t * | input | Pointer to a size_t value in host memory describing the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan. | 0 |

- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_NULL, then the device pointer in the VariantParamPack needs to be NULL as well.
- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_ELEM_ALIGNED or CUDNN_PTR_16B_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

## Table 11 CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| **Attribute key** | **Expected Descriptor Type Passed in, in the Setter** | **I/O Type** | **Description** | **Default Value** |
| X_YSUM | void * | input | Pointer to sum of y tensor on device, need to agree with previously set CUDNN_PARAM_YSUM_PLACEHOLDER attribute *. | NULL |
| X_YSQSUM | void * | input | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_YSQSUM_PLACEHOLDER attribute *. | NULL |
| X_BN_SCALE | void * | input | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_SCALE_PLACEHOLDER attribute *. | NULL |
| X_BN_BIAS | void * | input | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_BIAS_PLACEHOLDER attribute *. | NULL |
| X_BN_SAVED_MEAN | void * | output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_SAVED_MEAN_PLACEHOLDER attribute *. | NULL |
| X_BN_SAVED_INVSTD | void * | output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_SAVED_INVSTD_PLACEHOLDER attribute *. | NULL |
| X_BN_RUNNING_MEAN | void * | input/ output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_MEAN_PLACEHOLDER attribute *. | NULL |
| X_BN_RUNNING_VAR | void * | input/ output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_VAR_PLACEHOLDER attribute *. | NULL |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| Attribute key | Expected Descriptor Type Passed in, in the Setter | I/O Type | Description | Default Value |
| X_BN_EQSCALE | void * | output | Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *. | NULL |
| X_BN_EQBIAS | void * | output | Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *. | NULL |
| X_INT64_T_BN_ACCUMULATION_COUNT | int64_t * | input | Pointer to a scalar value in int64_t on host memory.<br>This value should describe the number of tensor elements accumulated in the sum of y and sum of y square tensors.<br>For example, in the single GPU use case, if the mode is CUDNN_BATCHNORM_SPATIAL or CUDNN_BATCHNORM_SPATIAL_PERSISTENT, the value should be equal to N*H*W of the tensor from which the statistics are calculated.<br>In multi-GPU use case, if all-reduce has been performed on the sum of y and sum of y square tensors, this value should be the sum of the single GPU accumulation count on each of the GPUs. | 0 |
| X_DOUBLE_BN_EXP_AVG_FACTOR | double * | input | Pointer to a scalar value in double on host memory.<br>Factor used in the moving average computation. See exponentialAverageFactor in cudnnBatchNormalization* APIs. | 0.0 |
| X_DOUBLE_BN_EPSILON | double * | input | Pointer to a scalar value in double on host memory.<br>A conditioning constant used in the batch normalization formula. Its value should be equal to or greater than the value defined | 0.0 |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| **Attribute key** | **Expected Descriptor Type Passed in, in the Setter** | **I/O Type** | **Description** | **Default Value** |
| | | | for CUDNN_BN_MIN_EPSILON in cudnn.h. See exponentialAverageFactor in cudnnBatchNormalization* APIs. | |
| X_WORKSPACE | void * | input | Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0. | NULL |
| X_SIZE_T_WORKSPACE_SIZE_IN_BYTES | size_t * | input | Pointer to a size_t value in host memory describing the user allocated workspace size in bytes. The amount need to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan. | 0 |

- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_NULL, then the device pointer in the VariantParamPack need to be NULL as well.
- If the corresponding pointer placeholder in ConstParamPack is set to CUDNN_PTR_ELEM_ALIGNED or CUDNN_PTR_16B_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 12 CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| **Attribute key** | **Expected Descriptor Type Passed in, in the Setter** | **I/O Type** | **Description** | **Default Value** |
| X_BN_SCALE | void * | input | Pointer to sum of y square tensor on device, need to agree with previously set | NULL |

| For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOp_t | | | | |
|---|---|---|---|---|
| Attribute key | Expected Descriptor Type Passed in, in the Setter | I/O Type | Description | Default Value |
| | | | CUDNN_PARAM_BN_SCALE_PLACEHOLDER attribute *. | |
| X_BN_BIAS | void * | input | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_BIAS_PLACEHOLDER attribute *. | NULL |
| X_BN_RUNNING_MEAN | void * | input/ output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_MEAN_PLACEHOLDER attribute *. | NULL |
| X_BN_RUNNING_VAR | void * | input/ output | Pointer to sum of y square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_VAR_PLACEHOLDER attribute *. | NULL |
| X_BN_EQSCALE | void * | output | Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *. | NULL |
| X_BN_EQBIAS | void * | output | Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *. | NULL |
| X_DOUBLE_BN_EPSILON | double * | input | Pointer to a scalar value in double on host memory.<br>A conditioning constant used in the batch normalization formula. Its value should be equal to or greater than the value defined for CUDNN_BN_MIN_EPSILON in cudnn.h.<br>See exponentialAverageFactor in cudnnBatchNormalization* APIs. | 0.0 |
| X_WORKSPACE | void * | input | Pointer to user allocated workspace on device. Can be | NULL |

| For the attribute `CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE` in `cudnnFusedOp_t` | | | | |
|---|---|---|---|---|
| **Attribute key** | **Expected Descriptor Type Passed in, in the Setter** | **I/O Type** | **Description** | **Default Value** |
| | | | **NULL** if the workspace size requested is 0. | |
| X_SIZE_T_WORKSPACE_SIZE_IN_BYTES | size_t * | input | Pointer to a **size_t** value in host memory describing the user allocated workspace size in bytes. The amount need to be equal or larger than the amount requested in **cudnnMakeFusedOpsPlan**. | 0 |

- If the corresponding pointer placeholder in **ConstParamPack** is set to **CUDNN_PTR_NULL**, then the device pointer in the **VariantParamPack** needs to be **NULL** as well.
- If the corresponding pointer placeholder in **ConstParamPack** is set to **CUDNN_PTR_ELEM_ALIGNED** or **CUDNN_PTR_16B_ALIGNED**, then the device pointer in the **VariantParamPack** may not be **NULL** and needs to be at least element-aligned or 16 bytes-aligned, respectively.

## 3.33. cudnnFusedOpsVariantParamPack_t

**cudnnFusedOpsVariantParamPack_t** is a pointer to an opaque structure holding the description of the **cudnnFusedOps** variant parameters. Use the function cudnnCreateFusedOpsVariantParamPack to create one instance of this structure, and the function cudnnDestroyFusedOpsVariantParamPack to destroy a previously-created descriptor.

## 3.34. cudnnHandle_t

**cudnnHandle_t** is a pointer to an opaque structure holding the cuDNN library context. The cuDNN library context must be created using **cudnnCreate()** and the returned handle must be passed to all subsequent library function calls. The context should be destroyed at the end using **cudnnDestroy()**. The context is associated with only one GPU device, the current device at the time of the call to **cudnnCreate()**. However, multiple contexts can be created on the same GPU device.

# 3.35. cudnnIndicesType_t

**cudnnIndicesType_t** is an enumerated type used to indicate the data type for the indices to be computed by the **cudnnReduceTensor()** routine. This enumerated type is used as a field for the **cudnnReduceTensorDescriptor_t** descriptor.

**Values**

**CUDNN_32BIT_INDICES**

Compute unsigned int indices.

**CUDNN_64BIT_INDICES**

Compute unsigned long indices.

**CUDNN_16BIT_INDICES**

Compute unsigned short indices.

**CUDNN_8BIT_INDICES**

Compute unsigned char indices.

# 3.36. cudnnLossNormalizationMode_t

**cudnnLossNormalizationMode_t** is an enumerated type that controls the input normalization mode for a loss function. This type can be used with cudnnSetCTCLossDescriptorEx.

**Values**

**CUDNN_LOSS_NORMALIZATION_NONE**

The input **probs** of cudnnCTCLoss function is expected to be the normalized probability, and the output **gradients** is the gradient of loss with respect to the unnormalized probability.

**CUDNN_LOSS_NORMALIZATION_SOFTMAX**

The input **probs** of cudnnCTCLoss function is expected to be the unnormalized activation from the previous layer, and the output **gradients** is the gradient with respect to the activation. Internally the probability is computed by softmax normalization.

# 3.37. cudnnLRNMode_t

**cudnnLRNMode_t** is an enumerated type used to specify the mode of operation in **cudnnLRNCrossChannelForward()** and **cudnnLRNCrossChannelBackward()**.

**Values**

**CUDNN_LRN_CROSS_CHANNEL_DIM1**

LRN computation is performed across tensor's dimension **dimA[1]**.

## 3.38. cudnnMathType_t

**cudnnMathType_t** is an enumerated type used to indicate if the use of Tensor Core operations is permitted a given library routine.

**Values**

**CUDNN_DEFAULT_MATH**

Tensor Core operations are not used.

**CUDNN_TENSOR_OP_MATH**

The use of Tensor Core operations is permitted.

**CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION**

Enables the use of FP32 tensors for both input and output.

## 3.39. cudnnMultiHeadAttnWeightKind_t

**cudnnMultiHeadAttnWeightKind_t** is an enumerated type that specifies a group of weights or biases in the **cudnnGetMultiHeadAttnWeights()** function.

**Values**

**CUDNN_MH_ATTN_Q_WEIGHTS**

Selects the input projection weights for **queries**.

**CUDNN_MH_ATTN_K_WEIGHTS**

Selects the input projection weights for **keys**.

**CUDNN_MH_ATTN_V_WEIGHTS**

Selects the input projection weights for **values**.

**CUDNN_MH_ATTN_O_WEIGHTS**

Selects the output projection weights.

**CUDNN_MH_ATTN_Q_BIASES**

Selects the input projection biases for **queries**.

**CUDNN_MH_ATTN_K_BIASES**

Selects the input projection biases for **keys**.

**CUDNN_MH_ATTN_V_BIASES**

Selects the input projection biases for **values**.

**CUDNN_MH_ATTN_O_BIASES**

Selects the output projection biases.

## 3.40. cudnnNanPropagation_t

**cudnnNanPropagation_t** is an enumerated type used to indicate if a given routine should propagate **Nan** numbers. This enumerated type is used as a field for the **cudnnActivationDescriptor_t** descriptor and **cudnnPoolingDescriptor_t** descriptor.

**Values**

**CUDNN_NOT_PROPAGATE_NAN**

**Nan** numbers are not propagated.

**CUDNN_PROPAGATE_NAN**

**Nan** numbers are propagated.

## 3.41. cudnnOpTensorDescriptor_t

**cudnnOpTensorDescriptor_t** is a pointer to an opaque structure holding the description of a Tensor Core operation, used as a parameter to **cudnnOpTensor()**. **cudnnCreateOpTensorDescriptor()** is used to create one instance, and **cudnnSetOpTensorDescriptor()** must be used to initialize this instance.

## 3.42. cudnnOpTensorOp_t

**cudnnOpTensorOp_t** is an enumerated type used to indicate the Tensor Core operation to be used by the **cudnnOpTensor()** routine. This enumerated type is used as a field for the **cudnnOpTensorDescriptor_t** descriptor.

**Values**

**CUDNN_OP_TENSOR_ADD**

The operation to be performed is addition.

**CUDNN_OP_TENSOR_MUL**

The operation to be performed is multiplication.

**CUDNN_OP_TENSOR_MIN**

The operation to be performed is a minimum comparison.

**CUDNN_OP_TENSOR_MAX**

The operation to be performed is a maximum comparison.

**CUDNN_OP_TENSOR_SQRT**

The operation to be performed is square root, performed on only the **A** tensor.

**CUDNN_OP_TENSOR_NOT**

The operation to be performed is negation, performed on only the **A** tensor.

## 3.43. cudnnPersistentRNNPlan_t

**cudnnPersistentRNNPlan_t** is a pointer to an opaque structure holding a plan to execute a dynamic persistent RNN. **cudnnCreatePersistentRNNPlan()** is used to create and initialize one instance.

## 3.44. cudnnPoolingDescriptor_t

**cudnnPoolingDescriptor_t** is a pointer to an opaque structure holding the description of a pooling operation. **cudnnCreatePoolingDescriptor()** is used to create one instance, and **cudnnSetPoolingNdDescriptor()** or **cudnnSetPooling2dDescriptor()** must be used to initialize this instance.

## 3.45. cudnnPoolingMode_t

**cudnnPoolingMode_t** is an enumerated type passed to **cudnnSetPoolingDescriptor()** to select the pooling method to be used by **cudnnPoolingForward()** and **cudnnPoolingBackward()**.

**Values**

**CUDNN_POOLING_MAX**

The maximum value inside the pooling window is used.

**CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING**

Values inside the pooling window are averaged. The number of elements used to calculate the average includes spatial locations falling in the padding region.

**CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING**

Values inside the pooling window are averaged. The number of elements used to calculate the average excludes spatial locations falling in the padding region.

**CUDNN_POOLING_MAX_DETERMINISTIC**

The maximum value inside the pooling window is used. The algorithm used is deterministic.

## 3.46. cudnnReduceTensorDescriptor_t

**cudnnReduceTensorDescriptor_t** is a pointer to an opaque structure holding the description of a tensor reduction operation, used as a parameter to **cudnnReduceTensor()**. **cudnnCreateReduceTensorDescriptor()** is used to create

one instance, and **cudnnSetReduceTensorDescriptor()** must be used to initialize this instance.

## 3.47. cudnnReduceTensorIndices_t

**cudnnReduceTensorIndices_t** is an enumerated type used to indicate whether indices are to be computed by the **cudnnReduceTensor()** routine. This enumerated type is used as a field for the **cudnnReduceTensorDescriptor_t** descriptor.

**Values**

**CUDNN_REDUCE_TENSOR_NO_INDICES**

Do not compute indices.

**CUDNN_REDUCE_TENSOR_FLATTENED_INDICES**

Compute indices. The resulting indices are relative, and flattened.

## 3.48. cudnnReduceTensorOp_t

**cudnnReduceTensorOp_t** is an enumerated type used to indicate the Tensor Core operation to be used by the **cudnnReduceTensor()** routine. This enumerated type is used as a field for the **cudnnReduceTensorDescriptor_t** descriptor.

**Values**

**CUDNN_REDUCE_TENSOR_ADD**

The operation to be performed is addition.

**CUDNN_REDUCE_TENSOR_MUL**

The operation to be performed is multiplication.

**CUDNN_REDUCE_TENSOR_MIN**

The operation to be performed is a minimum comparison.

**CUDNN_REDUCE_TENSOR_MAX**

The operation to be performed is a maximum comparison.

**CUDNN_REDUCE_TENSOR_AMAX**

The operation to be performed is a maximum comparison of absolute values.

**CUDNN_REDUCE_TENSOR_AVG**

The operation to be performed is averaging.

**CUDNN_REDUCE_TENSOR_NORM1**

The operation to be performed is addition of absolute values.

**CUDNN_REDUCE_TENSOR_NORM2**

The operation to be performed is a square root of sum of squares.

**CUDNN_REDUCE_TENSOR_MUL_NO_ZEROS**

The operation to be performed is multiplication, not including elements of value zero.

## 3.49. cudnnReorderType_t

```
typedef enum {
 CUDNN_DEFAULT_REORDER = 0,
 CUDNN_NO_REORDER      = 1,
 } cudnnReorderType_t;
```

**cudnnReorderType_t** is an enumerated type to set the convolution reordering type. The reordering type can be set by cudnnSetConvolutionReorderType and its status can be read by cudnnGetConvolutionReorderType.

## 3.50. cudnnRNNAlgo_t

**cudnnRNNAlgo_t** is an enumerated type used to specify the algorithm used in the **cudnnRNNForwardInference()**, **cudnnRNNForwardTraining()**, **cudnnRNNBackwardData()** and **cudnnRNNBackwardWeights()** routines.

**Values**

**CUDNN_RNN_ALGO_STANDARD**

Each RNN layer is executed as a sequence of operations. This algorithm is expected to have robust performance across a wide range of network parameters.

**CUDNN_RNN_ALGO_PERSIST_STATIC**

The recurrent parts of the network are executed using a *persistent kernel* approach. This method is expected to be fast when the first dimension of the input tensor is small (meaning, a small minibatch).

**CUDNN_RNN_ALGO_PERSIST_STATIC** is only supported on devices with compute capability >= 6.0.

**CUDNN_RNN_ALGO_PERSIST_DYNAMIC**

The recurrent parts of the network are executed using a *persistent kernel* approach. This method is expected to be fast when the first dimension of the input tensor is small (ie. a small minibatch). When using **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** persistent kernels are prepared at runtime and are able to optimize using the specific parameters of the network and active GPU. As such, when using **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** a one-time plan preparation stage must be executed. These plans can then be reused in repeated calls with the same model parameters.

The limits on the maximum number of hidden units supported when using **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** are significantly higher than the limits when using **CUDNN_RNN_ALGO_PERSIST_STATIC**, however throughput is likely to significantly reduce when exceeding the maximums supported by **CUDNN_RNN_ALGO_PERSIST_STATIC**. In this regime, this method will still outperform **CUDNN_RNN_ALGO_STANDARD** for some cases.

**CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is only supported on devices with compute capability >= 6.0 on Linux machines.

## 3.51. cudnnRNNBiasMode_t

**cudnnRNNBiasMode_t** is an enumerated type used to specify the number of bias vectors for RNN functions. See the description of the cudnnRNNMode_t enumerated type for the equations for each cell type based on the bias mode.

**Values**

**CUDNN_RNN_NO_BIAS**

Applies RNN cell formulas that do not use biases.

**CUDNN_RNN_SINGLE_INP_BIAS**

Applies RNN cell formulas that use one input bias vector in the input GEMM.

**CUDNN_RNN_DOUBLE_BIAS**

Applies RNN cell formulas that use two bias vectors.

**CUDNN_RNN_SINGLE_REC_BIAS**

Applies RNN cell formulas that use one recurrent bias vector in the recurrent GEMM.

## 3.52. cudnnRNNClipMode_t

**cudnnRNNClipMode_t** is an enumerated type used to select the LSTM cell clipping mode. It is used with **cudnnRNNSetClip()**, **cudnnRNNGetClip()** functions, and internally within LSTM cells.

**Values**

**CUDNN_RNN_CLIP_NONE**

Disables LSTM cell clipping.

**CUDNN_RNN_CLIP_MINMAX**

Enables LSTM cell clipping.

## 3.53. cudnnRNNDataDescriptor_t

**cudnnRNNDataDescriptor_t** is a pointer to an opaque structure holding the description of an RNN data set. The function **cudnnCreateRNNDataDescriptor()** is used to create one instance, and **cudnnSetRNNDataDescriptor()** must be used to initialize this instance.

## 3.54. cudnnRNNDataLayout_t

**cudnnRNNDataLayout_t** is an enumerated type used to select the RNN data layout. It is used used in the API calls cudnnGetRNNDataDescriptor and cudnnSetRNNDataDescriptor.

**Values**

**CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED**

Data layout is padded, with outer stride from one time-step to the next.

**CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_PACKED**

The sequence length is sorted and packed as in basic RNN API.

**CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED**

Data layout is padded, with outer stride from one batch to the next.

## 3.55. cudnnRNNDescriptor_t

**cudnnRNNDescriptor_t** is a pointer to an opaque structure holding the description of an RNN operation. **cudnnCreateRNNDescriptor()** is used to create one instance, and **cudnnSetRNNDescriptor()** must be used to initialize this instance.

## 3.56. cudnnRNNInputMode_t

**cudnnRNNInputMode_t** is an enumerated type used to specify the behavior of the first layer in the **cudnnRNNForwardInference()**, **cudnnRNNForwardTraining()**, **cudnnRNNBackwardData()** and **cudnnRNNBackwardWeights()** routines.

**Values**

**CUDNN_LINEAR_INPUT**

A biased matrix multiplication is performed at the input of the first recurrent layer.

**CUDNN_SKIP_INPUT**

No operation is performed at the input of the first recurrent layer. If **CUDNN_SKIP_INPUT** is used the leading dimension of the input tensor must be equal to the hidden state size of the network.

## 3.57. cudnnRNNMode_t

**cudnnRNNMode_t** is an enumerated type used to specify the type of network used in the cudnnRNNForwardInference, cudnnRNNForwardTraining, cudnnRNNBackwardData and cudnnRNNBackwardWeights routines.

**Values**

**CUDNN_RNN_RELU**

A single-gate recurrent neural network with a ReLU activation function.

In the forward pass, the output $h_t$ for a given iteration can be computed from the recurrent input $h_{t-1}$ and the previous layer input $x_t$, given the matrices **W, R** and the bias vectors, where **ReLU(x) = max(x, 0)**.

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** (default mode), then the following equation with biases $b_W$ and $b_R$ applies:

$$h_t = ReLU(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**, then the following equation with bias **b** applies:

$$h_t = ReLU(W_i x_t + R_i h_{t-1} + b_i)$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_NO_BIAS**, then the following equation applies:

$$h_t = ReLU(W_i x_t + R_i h_{t-1})$$

**CUDNN_RNN_TANH**

A single-gate recurrent neural network with a **tanh** activation function.

In the forward pass, the output $h_t$ for a given iteration can be computed from the recurrent input $h_{t-1}$ and the previous layer input $x_t$, given the matrices **W, R** and the bias vectors, and where **tanh** is the hyperbolic tangent function.

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** (default mode), then the following equation with biases$b_W$ and $b_R$ applies:

$$h_t = tanh(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**, then the following equation with bias **b** applies:

$$h_t = tanh(W_i x_t + R_i h_{t-1} + b_i)$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_NO_BIAS**, then the following equation applies:

$$h_t = tanh(W_i x_t + R_i h_{t-1})$$

**CUDNN_LSTM**

A four-gate Long Short-Term Memory network with no peephole connections.

In the forward pass, the output $h_t$ and cell output $c_t$ for a given iteration can be computed from the recurrent input $h_{t-1}$, the cell input $c_{t-1}$ and the previous layer input $x_t$, given the matrices **W, R** and the bias vectors.

In addition, the following applies:

- **σ** is the sigmoid operator such that: $\sigma(x) = 1 / (1 + e^{-x})$,
- ∘ represents a point-wise multiplication,
- **tanh** is the hyperbolic tangent function, and
- $i_t$, $f_t$, $o_t$, $c'_t$ represent the input, forget, output and new gates respectively.

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** (default mode), then the following equations with biases $b_W$ and $b_R$ apply:

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$
$$f_t = \sigma(W_f x_t + R_f h_{t-1} + b_{Wf} + b_{Rf})$$
$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_{Wo} + b_{Ro})$$

$$
\begin{aligned}
c'_t &= \tanh(W_c x_t + R_c h_{t-1} + b_{Wc} + b_{Rc}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ c'_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**, then the following equations with bias **b** apply:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_i) \\
f_t &= \sigma(W_f x_t + R_f h_{t-1} + b_f) \\
o_t &= \sigma(W_o x_t + R_o h_{t-1} + b_o) \\
c'_t &= \tanh(W_c x_t + R_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ c'_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_NO_BIAS**, then the following equations apply:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + R_i h_{t-1}) \\
f_t &= \sigma(W_f x_t + R_f h_{t-1}) \\
o_t &= \sigma(W_o x_t + R_o h_{t-1}) \\
c'_t &= \tanh(W_c x_t + R_c h_{t-1}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ c'_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

**CUDNN_GRU**

A three-gate network consisting of Gated Recurrent Units.

In the forward pass, the output $h_t$ for a given iteration can be computed from the recurrent input $h_{t-1}$ and the previous layer input $x_t$ given matrices **W, R** and the bias vectors.

In addition, the following applies:

- $\sigma$ is the sigmoid operator such that: $\sigma(x) = 1 / (1 + e^{-x})$,
- $\circ$ represents a point-wise multiplication,
- **tanh** is the hyperbolic tangent function, and
- $i_t$, $r_t$, $h'_t$ represent the input, reset, new gates respectively.

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** (default mode), then the following equations with biases $b_W$ and $b_R$ apply:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ru}) \\
r_t &= \sigma(W_r x_t + R_r h_{t-1} + b_{Wr} + b_{Rr}) \\
h'_t &= \tanh(W_h x_t + r_t \circ (R_h h_{t-1} + b_{Rh}) + b_{Wh}) \\
h_t &= (1 - i_t) \circ h'_t + i_t \circ h_{t-1}
\end{aligned}
$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS**, then the following equations with bias **b** apply:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_i) \\
r_t &= \sigma(W_r x_t + R_r h_{t-1} + b_r) \\
h'_t &= \tanh(W_h x_t + r_t \circ (R_h h_{t-1}) + b_{Wh}) \\
h_t &= (1 - i_t) \circ h'_t + i_t \circ h_{t-1}
\end{aligned}
$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_REC_BIAS**, then the following equations with bias **b** apply:

$$
\begin{aligned}
i_t &= \sigma(W_i x_t + R_i h_{t-1} + b_i) \\
r_t &= \sigma(W_r x_t + R_r h_{t-1} + b_r) \\
h'_t &= \tanh(W_h x_t + r_t \circ (R_h h_{t-1} + b_{Rh})) \\
h_t &= (1 - i_t) \circ h'_t + i_t \circ h_{t-1}
\end{aligned}
$$

If **cudnnRNNBiasMode_t biasMode** in **rnnDesc** is **CUDNN_RNN_NO_BIAS**, then the following equations apply:

$$i_t = \sigma(W_i x_t + R_i h_{t-1})$$
$$r_t = \sigma(W_r x_t + R_r h_{t-1})$$
$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1}))$$
$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

# 3.58. cudnnRNNPaddingMode_t

**cudnnRNNPaddingMode_t** is an enumerated type used to enable or disable the padded input/output.

**Values**

**CUDNN_RNN_PADDED_IO_DISABLED**
Disables the padded input/output.

**CUDNN_RNN_PADDED_IO_ENABLED**
Enables the padded input/output.

# 3.59. cudnnSamplerType_t

**cudnnSamplerType_t** is an enumerated type passed to **cudnnSetSpatialTransformerNdDescriptor()** to select the sampler type to be used by **cudnnSpatialTfSamplerForward()** and **cudnnSpatialTfSamplerBackward()**.

**Values**

**CUDNN_SAMPLER_BILINEAR**
Selects the bilinear sampler.

# 3.60. cudnnSeqDataAxis_t

**cudnnSeqDataAxis_t** is an enumerated type that indexes active dimensions in the **dimA[]** argument that is passed to the **cudnnSetSeqDataDescriptor()** function to configure the sequence data descriptor of type **cudnnSeqDataDescriptor_t**.

**cudnnSeqDataAxis_t** constants are also used in the **axis[]** argument of the **cudnnSetSeqDataDescriptor()** call to define the layout of the sequence data buffer in memory.

See **cudnnSetSeqDataDescriptor()** for a detailed description on how to use the **cudnnSeqDataAxis_t** enumerated type.

The **CUDNN_SEQDATA_DIM_COUNT** macro defines the number of constants in the **cudnnSeqDataAxis_t** enumerated type. This value is currently set to **4**.

**Values**

**CUDNN_SEQDATA_TIME_DIM**
Identifies the **TIME** (sequence length) dimension or specifies the **TIME** in the data layout.

**CUDNN_SEQDATA_BATCH_DIM**
Identifies the **BATCH** dimension or specifies the **BATCH** in the data layout.

**CUDNN_SEQDATA_BEAM_DIM**
Identifies the **BEAM** dimension or specifies the **BEAM** in the data layout.

**CUDNN_SEQDATA_VECT_DIM**
Identifies the **VECT** (vector) dimension or specifies the **VECT** in the data layout.

## 3.61. cudnnSeqDataDescriptor_t

**cudnnSeqDataDescriptor_t** is a pointer to an opaque structure holding parameters of the sequence data container or buffer. The sequence data container is used to store fixed size vectors defined by the **VECT** dimension. Vectors are arranged in additional three dimensions: **TIME**, **BATCH** and **BEAM**.

The **TIME** dimension is used to bundle vectors into sequences of vectors. The actual sequences can be shorter than the **TIME** dimension, therefore, additional information is needed about each sequence length and how unused (padding) vectors should be saved.

It is assumed that the sequence data container is fully packed. The **TIME**, **BATCH** and **BEAM** dimensions can be in any order when vectors are traversed in the ascending order of addresses. Six data layouts (permutation of **TIME**, **BATCH** and **BEAM**) are possible.

The **cudnnSeqDataDescriptor_t** object holds the following parameters:

- data type used by vectors
- **TIME**, **BATCH**, **BEAM** and **VECT** dimensions
- data layout
- the length of each sequence along the **TIME** dimension
- an optional value to be copied to output padding vectors

Use the **cudnnCreateSeqDataDescriptor()** function to create one instance of the sequence data descriptor object and **cudnnDestroySeqDataDescriptor()** to delete a previously created descriptor. Use the **cudnnSetSeqDataDescriptor()** function to configure the descriptor.

This descriptor is used by multi-head attention API functions.

## 3.62. cudnnSoftmaxAlgorithm_t

**cudnnSoftmaxAlgorithm_t** is used to select an implementation of the softmax function used in **cudnnSoftmaxForward()** and **cudnnSoftmaxBackward()**.

### Values

**CUDNN_SOFTMAX_FAST**

This implementation applies the straightforward softmax operation.

**CUDNN_SOFTMAX_ACCURATE**

This implementation scales each point of the softmax input domain by its maximum value to avoid potential floating point overflows in the softmax evaluation.

**CUDNN_SOFTMAX_LOG**

This entry performs the log softmax operation, avoiding overflows by scaling each point in the input domain as in **CUDNN_SOFTMAX_ACCURATE**.

## 3.63. cudnnSoftmaxMode_t

**cudnnSoftmaxMode_t** is used to select over which data the **cudnnSoftmaxForward()** and **cudnnSoftmaxBackward()** are computing their results.

**Values**

**CUDNN_SOFTMAX_MODE_INSTANCE**

The softmax operation is computed per image (**N**) across the dimensions C,H,W.

**CUDNN_SOFTMAX_MODE_CHANNEL**

The softmax operation is computed per spatial location (**H,W**) per image (**N**) across the dimension **C**.

## 3.64. cudnnSpatialTransformerDescriptor_t

**cudnnSpatialTransformerDescriptor_t** is a pointer to an opaque structure holding the description of a spatial transformation operation. **cudnnCreateSpatialTransformerDescriptor()** is used to create one instance, **cudnnSetSpatialTransformerNdDescriptor()** is used to initialize this instance, and **cudnnDestroySpatialTransformerDescriptor()** is used to destroy this instance.

## 3.65. cudnnStatus_t

**cudnnStatus_t** is an enumerated type used for function status returns. All cuDNN library functions return their status, which can be one of the following values:

**Values**

**CUDNN_STATUS_SUCCESS**

The operation completed successfully.

**CUDNN_STATUS_NOT_INITIALIZED**

The cuDNN library was not initialized properly. This error is usually returned when a call to **cudnnCreate()** fails or when **cudnnCreate()** has not been called prior to calling another cuDNN routine. In the former case, it is usually due to an error in the CUDA Runtime API called by **cudnnCreate()** or by an error in the hardware setup.

**CUDNN_STATUS_ALLOC_FAILED**

Resource allocation failed inside the cuDNN library. This is usually caused by an internal **cudaMalloc()** failure.

To correct: prior to the function call, deallocate previously allocated memory as much as possible.

**CUDNN_STATUS_BAD_PARAM**

An incorrect value or parameter was passed to the function.

To correct: ensure that all the parameters being passed have valid values.

**CUDNN_STATUS_ARCH_MISMATCH**

The function requires a feature absent from the current GPU device. Note that cuDNN only supports devices with compute capabilities greater than or equal to 3.0.

To correct: compile and run the application on a device with appropriate compute capability.

**CUDNN_STATUS_MAPPING_ERROR**

An access to GPU memory space failed, which is usually caused by a failure to bind a texture.

To correct: prior to the function call, unbind any previously bound textures.

Otherwise, this may indicate an internal error/bug in the library.

**CUDNN_STATUS_EXECUTION_FAILED**

The GPU program failed to execute. This is usually caused by a failure to launch some cuDNN kernel on the GPU, which can occur for multiple reasons.

To correct: check that the hardware, an appropriate version of the driver, and the cuDNN library are correctly installed.

Otherwise, this may indicate an internal error/bug in the library.

**CUDNN_STATUS_INTERNAL_ERROR**

An internal cuDNN operation failed.

**CUDNN_STATUS_NOT_SUPPORTED**

The functionality requested is not presently supported by cuDNN.

**CUDNN_STATUS_LICENSE_ERROR**

The functionality requested requires some license and an error was detected when trying to check the current licensing. This error can happen if the license is not present or is expired or if the environment variable **NVIDIA_LICENSE_FILE** is not set properly.

**CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING**

Runtime library required by RNN calls (**libcuda.so** or **nvcuda.dll**) cannot be found in predefined search paths.

**CUDNN_STATUS_RUNTIME_IN_PROGRESS**

Some tasks in the user stream are not completed.

**CUDNN_STATUS_RUNTIME_FP_OVERFLOW**

Numerical overflow occurred during the GPU kernel execution.

## 3.66. cudnnTensorDescriptor_t

**cudnnCreateTensorDescriptor_t** is a pointer to an opaque structure holding the description of a generic n-D dataset. **cudnnCreateTensorDescriptor()** is used to create one instance, and one of the routines **cudnnSetTensorNdDescriptor()**, **cudnnSetTensor4dDescriptor()** or **cudnnSetTensor4dDescriptorEx()** must be used to initialize this instance.

## 3.67. cudnnTensorFormat_t

**cudnnTensorFormat_t** is an enumerated type used by **cudnnSetTensor4dDescriptor()** to create a tensor with a pre-defined layout. For a detailed explanation of how these tensors are arranged in memory, see Data Layout Formats.

**Values**

**CUDNN_TENSOR_NCHW**

This tensor format specifies that the data is laid out in the following order: batch size, feature maps, rows, columns. The strides are implicitly defined in such a way that the data are contiguous in memory with no padding between images, feature maps, rows, and columns; the columns are the inner dimension and the images are the outermost dimension.

**CUDNN_TENSOR_NHWC**

This tensor format specifies that the data is laid out in the following order: batch size, rows, columns, feature maps. The strides are implicitly defined in such a way that the data are contiguous in memory with no padding between images, rows, columns, and feature maps; the feature maps are the inner dimension and the images are the outermost dimension.

**CUDNN_TENSOR_NCHW_VECT_C**

This tensor format specifies that the data is laid out in the following order: batch size, feature maps, rows, columns. However, each element of the tensor is a vector of multiple feature maps. The length of the vector is carried by the data type of the tensor. The strides are implicitly defined in such a way that the data are contiguous in memory with no padding between images, feature maps, rows, and columns; the columns are the inner dimension and the images are the outermost dimension. This format is only supported with tensor data types **CUDNN_DATA_INT8x4**, **CUDNN_DATA_INT8x32**, and **CUDNN_DATA_UINT8x4**.

The **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.

## 3.68. cudnnTensorTransformDescriptor_t

**cudnnTensorTransformDescriptor_t** is an opaque structure containing the description of the tensor transform. Use the cudnnCreateTensorTransformDescriptor function to create an instance of this descriptor, and cudnnDestroyTensorTransformDescriptor function to destroy a previously created instance.

## 3.69. cudnnWgradMode_t

**cudnnWgradMode_t** is an enumerated type that selects how buffers holding gradients of the loss function, computed with respect to trainable parameters, are updated. Currently, this type is used by the **cudnnGetMultiHeadAttnWeights()** function only.

**Values**

**CUDNN_WGRAD_MODE_ADD**

A weight gradient component corresponding to a new batch of inputs is added to previously evaluated weight gradients. Before using this mode, the buffer holding weight gradients should be initialized to zero. Alternatively, the first API call outputting to an uninitialized buffer should use the **CUDNN_WGRAD_MODE_SET** option.

**CUDNN_WGRAD_MODE_SET**

A weight gradient component, corresponding to a new batch of inputs, overwrites previously stored weight gradients in the output buffer.

# Chapter 4. CUDNN API REFERENCE

This chapter describes the API of all the routines of the cuDNN library.

## 4.1. cudnnActivationBackward

```
cudnnStatus_t cudnnActivationBackward(
    cudnnHandle_t                    handle,
    cudnnActivationDescriptor_t      activationDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    yDesc,
    const void                      *y,
    const cudnnTensorDescriptor_t    dyDesc,
    const void                      *dy,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    dxDesc,
    void                            *dx)
```

This routine computes the gradient of a neuron activation function.

- In-place operation is allowed for this routine; meaning **dy** and **dx** pointers may be equal. However, this requires the corresponding tensor descriptors to be identical (particularly, the strides of the input and output must match for an in-place operation to be allowed).
- All tensor formats are supported for 4 and 5 dimensions, however, the best performance is obtained when the strides of **yDesc** and **xDesc** are equal and **HW-packed**. For more than 5 dimensions the tensors must have their spatial dimensions packed.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**activationDesc**

*Input*. Activation descriptor. See cudnnActivationDescriptor_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**yDesc**

*Input*. Handle to the previously initialized input tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**y**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **yDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dyDesc**.

**xDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**x**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **xDesc**.

**dxDesc**

*Input*. Handle to the previously initialized output differential tensor descriptor.

**dx**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **dxDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The strides **nStride, cStride, hStride, wStride** of the input differential tensor and output differential tensor differ and in-place operation is used.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimensions **n, c, h, w** of the input tensor and output tensor differ.
- The **datatype** of the input tensor and output tensor differs.
- The strides **nStride, cStride, hStride, wStride** of the input tensor and the input differential tensor differ.
- The strides **nStride, cStride, hStride, wStride** of the output tensor and the output differential tensor differ.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.2. cudnnActivationForward

```
cudnnStatus_t cudnnActivationForward(
    cudnnHandle_t handle,
    cudnnActivationDescriptor_t     activationDesc,
    const void                     *alpha,
    const cudnnTensorDescriptor_t   xDesc,
    const void                     *x,
    const void                     *beta,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y)
```

This routine applies a specified neuron activation function element-wise over each input value.

- In-place operation is allowed for this routine; meaning, **xData** and **yData** pointers may be equal. However, this requires **xDesc** and **yDesc** descriptors to be identical (particularly, the strides of the input and output must match for an in-place operation to be allowed).
- All tensor formats are supported for 4 and 5 dimensions, however, the best performance is obtained when the strides of **xDesc** and **yDesc** are equal and **HW-packed**. For more than 5 dimensions the tensors must have their spatial dimensions packed.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**activationDesc**

*Input*. Activation descriptor. For more information, see cudnnActivationDescriptor_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The parameter **mode** has an invalid enumerant value.
- The dimensions **n, c, h, w** of the input tensor and output tensor differ.
- The **datatype** of the input tensor and output tensor differs.
- The strides **nStride, cStride, hStride, wStride** of the input tensor and output tensor differ and in-place operation is used (meaning, **x** and **y** pointers are equal).

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.3. cudnnAddTensor

```
cudnnStatus_t cudnnAddTensor(
    cudnnHandle_t                     handle,
```

```
    const void                          *alpha,
    const cudnnTensorDescriptor_t        aDesc,
    const void                          *A,
    const void                          *beta,
    const cudnnTensorDescriptor_t        cDesc,
    void                                *C)
```

This function adds the scaled values of a bias tensor to another tensor. Each dimension of the bias tensor **A** must match the corresponding dimension of the destination tensor **C** or must be equal to 1. In the latter case, the same value from the bias tensor for those dimensions will be used to blend into the **C** tensor.

Up to dimension 5, all tensor formats are supported. Beyond those dimensions, this routine is not supported

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with the prior value in the destination tensor as follows:

```
dstValue = alpha[0]*srcValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**aDesc**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**A**

*Input*. Pointer to data of the tensor described by the **aDesc** descriptor.

**cDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**C**

*Input/Output*. Pointer to data of the tensor described by the **cDesc** descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function executed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

The dimensions of the bias tensor refer to an amount of data that is incompatible with the output tensor dimensions or the **dataType** of the two tensor descriptors are different.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.4. cudnnBatchNormalizationBackward

```
cudnnStatus_t cudnnBatchNormalizationBackward(
      cudnnHandle_t                    handle,
      cudnnBatchNormMode_t             mode,
      const void                      *alphaDataDiff,
      const void                      *betaDataDiff,
      const void                      *alphaParamDiff,
      const void                      *betaParamDiff,
      const cudnnTensorDescriptor_t    xDesc,
      const void                      *x,
      const cudnnTensorDescriptor_t    dyDesc,
      const void                      *dy,
      const cudnnTensorDescriptor_t    dxDesc,
      void                            *dx,
      const cudnnTensorDescriptor_t    bnScaleBiasDiffDesc,
      const void                      *bnScale,
      void                            *resultBnScaleDiff,
      void                            *resultBnBiasDiff,
      double                           epsilon,
      const void                      *savedMean,
      const void                      *savedInvVariance)
```

This function performs the backward batch normalization layer computation. This layer is based on the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, S. Ioffe, C. Szegedy, 2015. .

- Only 4D and 5D tensors are supported.
- The **epsilon** value has to be the same during training, backpropagation, and inference.
- Higher performance can be obtained when **HW-packed** tensors are used for all of **x, dy, dx.**

For more information, see **cudnnDeriveBNTensorDescriptor** for the secondary tensor descriptor generation for the parameters used in this function.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**mode**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

***alphaDataDiff**, ***betaDataDiff**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the gradient output **dx** with a prior value in the destination tensor as follows:

```
dstValue = alphaDataDiff[0]*resultValue + betaDataDiff[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

***alphaParamDiff**, ***betaParamDiff**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the gradient outputs **resultBnScaleDiff** and **resultBnBiasDiff** with prior values in the destination tensor as follows:

```
dstValue = alphaParamDiff[0]*resultValue + betaParamDiff[0]*priorDstValue
```

For more information, see Scaling Parameters.

**xDesc, dxDesc, dyDesc**

*Inputs*. Handles to the previously initialized tensor descriptors.

***x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**, for the layer's **x** data.

***dy**

*Inputs*. Data pointer to GPU memory associated with the tensor descriptor **dyDesc**, for the backpropagated differential **dy** input.

***dx**

*Inputs*. Data pointer to GPU memory associated with the tensor descriptor **dxDesc**, for the resulting differential output with respect to **x**.

**bnScaleBiasDiffDesc**

*Input*. Shared tensor descriptor for the following five tensors: **bnScale, resultBnScaleDiff, resultBnBiasDiff, savedMean, savedInvVariance**. The dimensions for this tensor descriptor are dependent on normalization mode. For more information, see cudnnDeriveBNTensorDescriptor.

The data type of this tensor descriptor must be **float** for FP16 and FP32 input tensors, and **double** for FP64 input tensors.

***bnScale**

*Input*. Pointer in the device memory for the batch normalization **scale** parameter (in the original paper the quantity **scale** is referred to as gamma).

The **bnBias** parameter is not needed for this layer's computation.

**resultBnScaleDiff, resultBnBiasDiff**

*Outputs*. Pointers in device memory for the resulting scale and bias differentials computed by this routine. Note that these scale and bias gradients are weight gradients specific to this batch normalization operation, and by definition are not backpropagated.

**epsilon**

*Input*. Epsilon value used in batch normalization formula. Its value should be equal to or greater than the value defined for **CUDNN_BN_MIN_EPSILON** in **cudnn.h**. The same **epsilon** value should be used in forward and backward functions.

***savedMean, *savedInvVariance**

*Inputs*. Optional cache parameters containing saved intermediate results that were computed during the forward pass. For this to work correctly, the layer's **x** and **bnScale** data have to remain unchanged until this backward function is called.

Both these parameters can be **NULL** but only at the same time. It is recommended to use this cache since the memory overhead is relatively small.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- Any of the pointers **alpha, beta, x, dy, dx, bnScale, resultBnScaleDiff, resultBnBiasDiff** is **NULL**.
- The number of **xDesc** or **yDesc** or **dxDesc** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- **bnScaleBiasDiffDesc** dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Exactly one of **savedMean**, **savedInvVariance** pointers is **NULL**.
- **epsilon** value is less than **CUDNN_BN_MIN_EPSILON**.

- Dimensions or data types mismatch for any pair of **xDesc, dyDesc, dxDesc**.

# 4.5. cudnnBatchNormalizationBackwardEx

```
cudnnStatus_t cudnnBatchNormalizationBackwardEx (
    cudnnHandle_t                         handle,
    cudnnBatchNormMode_t                  mode,
    cudnnBatchNormOps_t                   bnOps,
    const void                            *alphaDataDiff,
    const void                            *betaDataDiff,
    const void                            *alphaParamDiff,
    const void                            *betaParamDiff,
    const cudnnTensorDescriptor_t         xDesc,
    const void                            *xData,
    const cudnnTensorDescriptor_t         yDesc,
    const void                            *yData,
    const cudnnTensorDescriptor_t         dyDesc,
    const void                            *dyData,
    const cudnnTensorDescriptor_t         dzDesc,
    void                                  *dzData,
    const cudnnTensorDescriptor_t         dxDesc,
    void                                  *dxData,
    const cudnnTensorDescriptor_t         dBnScaleBiasDesc,
    const void                            *bnScaleData,
    const void                            *bnBiasData,
    void                                  *dBnScaleData,
    void                                  *dBnBiasData,
    double                                epsilon,
    const void                            *savedMean,
    const void                            *savedInvVariance,
    const cudnnActivationDescriptor_t     activationDesc,
    void                                  *workspace,
    size_t                                workSpaceSizeInBytes
    void                                  *reserveSpace
    size_t                                reserveSpaceSizeInBytes);
```

This function is an extension of the cudnnBatchNormalizationBackward for performing the backward batch normalization layer computation with a fast NHWC semi-persistent kernel. This API will trigger the new semi-persistent NHWC kernel when the following conditions are true:

- All tensors, namely, **x, y, dz, dy, dx** must be NHWC-fully packed, and must be of the type **CUDNN_DATA_HALF**.
- The tensor C dimension should be a multiple of 4.
- The input parameter **mode** must be set to **CUDNN_BATCHNORM_SPATIAL_PERSISTENT**.
- **workspace** is not **NULL**.
- **workSpaceSizeInBytes** is equal or larger than the amount required by **cudnnGetBatchNormalizationBackwardExWorkspaceSize()**.
- **reserveSpaceSizeInBytes** is equal or larger than the amount required by **cudnnGetBatchNormalizationTrainingExReserveSpaceSize()**.
- The content in **reserveSpace** stored by cudnnBatchNormalizationForwardTrainingEx must be preserved.

If **workspace** is **NULL** and **workSpaceSizeInBytes** of zero is passed in, this API will function exactly like the non-extended function **cudnnBatchNormalizationBackward**.

This **workspace** is not required to be clean. Moreover, the **workspace** does not have to remain unchanged between the forward and backward pass, as it is not used for passing any information.

This extended function can accept a ***workspace** pointer to the GPU workspace, and **workSpaceSizeInBytes**, the size of the workspace, from the user.

The **bnOps** input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

Only 4D and 5D tensors are supported. The **epsilon** value has to be the same during the training, the backpropagation, and the inference.

When the tensor layout is NCHW, higher performance can be obtained when HW-packed tensors are used for **x, dy, dx**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**mode**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**bnOps**

*Input*. Mode of operation for the fast NHWC kernel. For more information, see cudnnBatchNormOps_t. This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

***alphaDataDiff, *betaDataDiff**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the gradient output **dx** with a prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

***alphaParamDiff, *betaParamDiff**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the gradient outputs **dBnScaleData** and **dBnBiasData** with prior values in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc, *x, yDesc, *yData, dyDesc, *dyData, dzDesc, *dzData, dxDesc, *dx/dt**

*Inputs*. Tensor descriptors and pointers in the device memory for the layer's **x** data, backpropagated differential **dy** (inputs), the optional **y** input data, the optional **dz**

output, and the **dx** output, which is the resulting differential with respect to **x**. For more information, see cudnnTensorDescriptor_t.

**dBnScaleBiasDesc**

*Input*. Shared tensor descriptor for the following six tensors: **bnScaleData**, **bnBiasData**, **dBnScaleData**, **dBnBiasData**, **savedMean**, and **savedInvVariance**. For more information, see cudnnDeriveBNTensorDescriptor.

The dimensions for this tensor descriptor are dependent on normalization mode.

Note: The data type of this tensor descriptor must be **float** for FP16 and FP32 input tensors and **double** for FP64 input tensors.

For more information, see cudnnTensorDescriptor_t.

***bnScaleData**

*Input*. Pointer in the device memory for the batch normalization scale parameter (in the original paper the quantity scale is referred to as gamma).

***bnBiasData**

*Input*. Pointers in the device memory for the batch normalization bias parameter (in the original paper bias is referred to as beta). This parameter is used only when activation should be performed.

***dBnScaleData, dBnBiasData**

*Inputs*. Pointers in the device memory for the gradients of **bnScaleData** and **bnBiasData**, respectively.

**epsilon**

*Input*. Epsilon value used in batch normalization formula. Its value should be equal to or greater than the value defined for **CUDNN_BN_MIN_EPSILON** in **cudnn.h**. The same epsilon value should be used in forward and backward functions.

***savedMean, *savedInvVariance**

*Inputs*. Optional cache parameters containing saved intermediate results computed during the forward pass. For this to work correctly, the layer's **x** and **bnScaleData**, **bnBiasData** data has to remain unchanged until this backward function is called. Note that both these parameters can be **NULL** but only at the same time. It is recommended to use this cache since the memory overhead is relatively small.

**activationDesc**

*Input*. The tensor descriptor for the activation operation.

**workspace**

*Input*. Pointer to the GPU workspace. If **workspace** is **NULL** and **workSpaceSizeInBytes** of zero is passed in, then this API will function exactly like the non-extended function cudnnBatchNormalizationBackward.

**workSpaceSizeInBytes**

*Input*. The size of the workspace. It must be large enough to trigger the fast NHWC semi-persistent kernel by this function.

**`*reserveSpace`**

*Input*. Pointer to the GPU workspace for the **`reserveSpace`**.

**`reserveSpaceSizeInBytes`**

*Input*. The size of the **`reserveSpace`**. It must be equal or larger than the amount required by **`cudnnGetBatchNormalizationTrainingExReserveSpaceSize()`**.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The computation was performed successfully.

**`CUDNN_STATUS_NOT_SUPPORTED`**

The function does not support the provided configuration.

**`CUDNN_STATUS_BAD_PARAM`**

At least one of the following conditions are met:

- Any of the pointers **`alphaDataDiff, betaDataDiff, alphaParamDiff, betaParamDiff, x, dy, dx, bnScale, resultBnScaleDiff, resultBnBiasDiff`** is **`NULL`**.
- The number of **`xDesc`** or **`yDesc`** or **`dxDesc`** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- **`dBnScaleBiasDesc`** dimensions not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Exactly one of **`savedMean`**, **`savedInvVariance`** pointers is **`NULL`**.
- **`epsilon`** value is less than **`CUDNN_BN_MIN_EPSILON`**.
- Dimensions or data types mismatch for any pair of **`xDesc`**, **`dyDesc`**, **`dxDesc`**.

## 4.6. cudnnBatchNormalizationForwardInference

```
cudnnStatus_t cudnnBatchNormalizationForwardInference(
      cudnnHandle_t                     handle,
      cudnnBatchNormMode_t              mode,
      const void                       *alpha,
      const void                       *beta,
      const cudnnTensorDescriptor_t     xDesc,
      const void                       *x,
      const cudnnTensorDescriptor_t     yDesc,
      void                             *y,
      const cudnnTensorDescriptor_t     bnScaleBiasMeanVarDesc,
      const void                       *bnScale,
      const void                       *bnBias,
      const void                       *estimatedMean,
      const void                       *estimatedVariance,
      double                            epsilon)
```

This function performs the forward batch normalization layer computation for the inference phase. This layer is based on the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, S. Ioffe, C. Szegedy, 2015.

- Only 4D and 5D tensors are supported.
- The input transformation performed by this function is defined as:

```
y = beta*y + alpha *[bnBias + (bnScale * (x-estimatedMean)/
sqrt(epsilon + estimatedVariance)]
```

- The **epsilon** value has to be the same during training, backpropagation and inference.
- For the training phase, use cudnnBatchNormalizationForwardTraining.
- Higher performance can be obtained when HW-packed tensors are used for all of **x** and **dx**.

For more information, see cudnnDeriveBNTensorDescriptor for the secondary tensor descriptor generation for the parameters used in this function.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**mode**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**alpha, beta**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc, yDesc**

*Input*. Handles to the previously initialized tensor descriptors.

***x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**, for the layer's **x** input data.

***y**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **yDesc**, for the **y**output of the batch normalization layer.

**bnScaleBiasMeanVarDesc, bnScale, bnBias**

*Inputs*. Tensor descriptors and pointers in device memory for the batch normalization scale and bias parameters (in the original paper bias is referred to as beta and scale as gamma).

**estimatedMean, estimatedVariance**

*Inputs*. Mean and variance tensors (these have the same descriptor as the bias and scale). The **resultRunningMean** and **resultRunningVariance**, accumulated during the training phase from the **cudnnBatchNormalizationForwardTraining()** call, should be passed as inputs here.

**epsilon**

*Input*. Epsilon value used in the batch normalization formula. Its value should be equal to or greater than the value defined for **CUDNN_BN_MIN_EPSILON** in **cudnn.h**.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the pointers **alpha, beta, x, y, bnScale, bnBias, estimatedMean, estimatedInvVariance** is **NULL**.
- The number of **xDesc** or **yDesc** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported.)
- **bnScaleBiasMeanVarDesc** dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- **epsilon** value is less than **CUDNN_BN_MIN_EPSILON**.
- Dimensions or data types mismatch for **xDesc**, **yDesc**.

## 4.7. cudnnBatchNormalizationForwardTraining

```
cudnnStatus_t cudnnBatchNormalizationForwardTraining(
      cudnnHandle_t                    handle,
      cudnnBatchNormMode_t             mode,
      const void                      *alpha,
      const void                      *beta,
      const cudnnTensorDescriptor_t    xDesc,
      const void                      *x,
      const cudnnTensorDescriptor_t    yDesc,
```

```
    void                              *y,
    const cudnnTensorDescriptor_t      bnScaleBiasMeanVarDesc,
    const void                        *bnScale,
    const void                        *bnBias,
    double                             exponentialAverageFactor,
    void                              *resultRunningMean,
    void                              *resultRunningVariance,
    double                             epsilon,
    void                              *resultSaveMean,
    void                              *resultSaveInvVariance)
```

This function performs the forward batch normalization layer computation for the training phase. This layer is based on the paper *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, S. Ioffe, C. Szegedy, 2015.

- Only 4D and 5D tensors are supported.
- The `epsilon` value has to be the same during training, backpropagation, and inference.
- For the inference phase, use `cudnnBatchNormalizationForwardInference`.
- Higher performance can be obtained when HW-packed tensors are used for both `x` and `y`.

See cudnnDeriveBNTensorDescriptor for the secondary tensor descriptor generation for the parameters used in this function.

### Parameters

**`handle`**

Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**`mode`**

Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**`alpha, beta`**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**`xDesc, yDesc`**

Tensor descriptors and pointers in device memory for the layer's **`x`** and **`y`** data. For more information, see cudnnTensorDescriptor_t.

**`*x`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`xDesc`**, for the layer's **`x`** input data.

***y**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **yDesc**, for the **y**output of the batch normalization layer.

**bnScaleBiasMeanVarDesc**

Shared tensor descriptor **desc** for the secondary tensor that was derived by cudnnDeriveBNTensorDescriptor. The dimensions for this tensor descriptor are dependent on the normalization mode.

**bnScale, bnBias**

*Inputs*. Pointers in device memory for the batch normalization scale and bias parameters (in the original paper bias is referred to as beta and scale as gamma). Note that **bnBias** parameter can replace the previous layer's bias parameter for improved efficiency.

**exponentialAverageFactor**

*Input*. Factor used in the moving average computation as follows:

```
runningMean = runningMean*(1-factor) + newMean*factor
```

Use a **factor=1/(1+n)** at **N**-th call to the function to get Cumulative Moving Average (CMA) behavior such that:

```
CMA[n] = (x[1]+...+x[n])/n
```

This is proved below:

Writing

```
CMA[n+1] = (n*CMA[n]+x[n+1])/(n+1)
= ((n+1)*CMA[n]-CMA[n])/(n+1) + x[n+1]/(n+1)
= CMA[n]*(1-1/(n+1))+x[n+1]*1/(n+1)
= CMA[n]*(1-factor) + x(n+1)*factor
```

**resultRunningMean, resultRunningVariance**

*Inputs/Outputs*. Running mean and variance tensors (these have the same descriptor as the bias and scale). Both of these pointers can be **NULL** but only at the same time. The value stored in **resultRunningVariance** (or passed as an input in inference mode) is the sample variance and is the moving average of **variance[x]** where the variance is computed either over batch or spatial+batch dimensions depending on the mode. If these pointers are not **NULL**, the tensors should be initialized to some reasonable values or to 0.

**epsilon**

*Input*. Epsilon value used in the batch normalization formula. Its value should be equal to or greater than the value defined for **CUDNN_BN_MIN_EPSILON** in **cudnn.h**. The same **epsilon** value should be used in forward and backward functions.

**resultSaveMean, resultSaveInvVariance**

*Outputs*. Optional cache to save intermediate results computed during the forward pass. These buffers can be used to speed up the backward pass when supplied to the cudnnBatchNormalizationBackward function. The intermediate results stored in **resultSaveMean** and **resultSaveInvVariance** buffers should not be used

directly by the user. Depending on the batch normalization mode, the results stored in **resultSaveInvVariance** may vary. For the cache to work correctly, the input layer data must remain unchanged until the backward function is called. Note that both parameters can be **NULL** but only at the same time. In such a case, intermediate statistics will not be saved, and cudnnBatchNormalizationBackward will have to re-compute them. It is recommended to use this cache as the memory overhead is relatively small because these tensors have a much lower product of dimensions than the data tensors.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the pointers **alpha, beta, x, y, bnScale, bnBias** is **NULL**.
- The number of **xDesc** or **yDesc** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- **bnScaleBiasMeanVarDesc** dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Exactly one of **resultSaveMean**, **resultSaveInvVariance** pointers are **NULL**.
- Exactly one of **resultRunningMean**, **resultRunningInvVariance** pointers are **NULL**.
- **epsilon** value is less than **CUDNN_BN_MIN_EPSILON**.
- Dimensions or data types mismatch for **xDesc**, **yDesc**

## 4.8. cudnnBatchNormalizationForwardTrainingEx

```
cudnnStatus_t cudnnBatchNormalizationForwardTrainingEx(
   cudnnHandle_t                        handle,
   cudnnBatchNormMode_t                 mode,
   cudnnBatchNormOps_t                  bnOps,
   const void                           *alpha,
   const void                           *beta,
   const cudnnTensorDescriptor_t        xDesc,
   const void                           *xData,
   const cudnnTensorDescriptor_t        zDesc,
   const void                           *zData,
   const cudnnTensorDescriptor_t        yDesc,
   void                                 *yData,
   const cudnnTensorDescriptor_t        bnScaleBiasMeanVarDesc,
   const void                           *bnScaleData,
   const void                           *bnBiasData,
```

```
    double                              exponentialAverageFactor,
    void                                *resultRunningMeanData,
    void                                *resultRunningVarianceData,
    double                              epsilon,
    void                                *saveMean,
    void                                *saveInvVariance,
    const cudnnActivationDescriptor_t   activationDesc,
    void                                *workspace,
    size_t                              workSpaceSizeInBytes
    void                                *reserveSpace
    size_t                              reserveSpaceSizeInBytes);
```

This function is an extension of the **cudnnBatchNormalizationForwardTraining()** for performing the forward batch normalization layer computation.

This API will trigger the new semi-persistent NHWC kernel when the following conditions are true:

- All tensors, namely, **x, y, dz, dy, dx** must be NHWC-fully packed and must be of the type **CUDNN_DATA_HALF**.
- The tensor **C** dimension should be a multiple of 4.
- The input parameter **mode** must be set to **CUDNN_BATCHNORM_SPATIAL_PERSISTENT**.
- **workspace** is not **NULL**.
- **workSpaceSizeInBytes** is equal or larger than the amount required by **cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()**.
- **reserveSpaceSizeInBytes** is equal or larger than the amount required by **cudnnGetBatchNormalizationTrainingExReserveSpaceSize()**.
- The content in **reserveSpace** stored by cudnnBatchNormalizationForwardTrainingEx must be preserved.

If **workspace** is **NULL** and **workSpaceSizeInBytes** of zero is passed in, this API will function exactly like the non-extended function **cudnnBatchNormalizationForwardTraining()**.

This workspace is not required to be clean. Moreover, the workspace does not have to remain unchanged between the forward and backward pass, as it is not used for passing any information.

This extended function can accept a ***workspace** pointer to the GPU workspace, and **workSpaceSizeInBytes**, the size of the workspace, from the user.

The **bnOps** input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

Only 4D and 5D tensors are supported. The **epsilon** value has to be the same during the training, the backpropagation, and the inference.

When the tensor layout is NCHW, higher performance can be obtained when HW-packed tensors are used for **x, dy, dx**.

### Parameters

**`handle`**

Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**`mode`**

Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**`bnOps`**

*Input*. Mode of operation for the fast NHWC kernel. See cudnnBatchNormOps_t. This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

**`*alpha`, `*beta`**

*Inputs*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**`xDesc`, `*xData`, `zDesc`, `*zData`, `yDesc`, `*yData`**

Tensor descriptors and pointers in device memory for the layer's **`x`** and **`y`** data, and for the optional **`z`** tensor input for residual addition to the result of the batch normalization operation, prior to the activation. The optional tensor input **`z`** should be exactly the same size as **`x`** and the final output **`y`**. This **`z`** input is element-wise added to the output of batch normalization. This addition optionally happens after batch normalization and before the activation. For more information, see cudnnTensorDescriptor_t .

**`bnScaleBiasMeanVarDesc`**

Shared tensor descriptor **`desc`** for the secondary tensor that was derived by cudnnDeriveBNTensorDescriptor. The dimensions for this tensor descriptor are dependent on the normalization mode.

**`*bnScaleData`, `*bnBiasData`**

*Inputs*. Pointers in device memory for the batch normalization scale and bias parameters (in the original paper, bias is referred to as beta and scale as gamma). Note that **`bnBiasData`** parameter can replace the previous layer's bias parameter for improved efficiency.

**`exponentialAverageFactor`**

*Input*. Factor used in the moving average computation as follows:

```
runningMean = runningMean*(1-factor) + newMean*factor
```

Use a **factor=1/(1+n)** at **N**-th call to the function to get Cumulative Moving Average (CMA) behavior such that:

```
CMA[n] = (x[1]+...+x[n])/n
```

This is proved below:

Writing

```
CMA[n+1] = (n*CMA[n]+x[n+1])/(n+1)
= ((n+1)*CMA[n]-CMA[n])/(n+1) + x[n+1]/(n+1)
= CMA[n]*(1-1/(n+1))+x[n+1]*1/(n+1)
= CMA[n]*(1-factor) + x(n+1)*factor
```

***resultRunningMeanData**, ***resultRunningVarianceData**

*Inputs/Outputs*. Pointers to the running mean and running variance data. Both these pointers can be **NULL** but only at the same time. The value stored in **resultRunningVarianceData** (or passed as an input in inference mode) is the sample variance and is the moving average of **variance[x]** where the variance is computed either over batch or spatial+batch dimensions depending on the mode. If these pointers are not **NULL**, the tensors should be initialized to some reasonable values or to 0.

**epsilon**

*Input*. Epsilon value used in the batch normalization formula. Its value should be equal to or greater than the value defined for **CUDNN_BN_MIN_EPSILON** in **cudnn.h**. The same **epsilon** value should be used in forward and backward functions.

***saveMean**, ***saveInvVariance**

*Outputs*. Optional cache parameters containing saved intermediate results computed during the forward pass. For this to work correctly, the layer's **x** and **bnScaleData**, **bnBiasData** data has to remain unchanged until this backward function is called. Note that both these parameters can be **NULL** but only at the same time. It is recommended to use this cache since the memory overhead is relatively small.

**activationDesc**

*Input*. The tensor descriptor for the activation operation. When the **bnOps** input is set to either **CUDNN_BATCHNORM_OPS_BN_ACTIVATION** or **CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION** then this activation is used.

***workspace**, **workSpaceSizeInBytes**

*Inputs*. ***workspace** is a pointer to the GPU workspace, and **workSpaceSizeInBytes** is the size of the workspace. When ***workspace** is not **NULL** and ***workSpaceSizeInBytes** is large enough, and the tensor layout is NHWC and the data type configuration is supported, then this function will trigger a new semi-persistent NHWC kernel for batch normalization. The workspace is not required to be clean. Also, the workspace does not need to remain unchanged between the forward and backward passes.

***reserveSpace**

*Input*. Pointer to the GPU workspace for the **reserveSpace**.

**reserveSpaceSizeInBytes**

*Input*. The size of the **reserveSpace**. Must be equal or larger than the amount required by **cudnnGetBatchNormalizationTrainingExReserveSpaceSize()**.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the pointers **alpha, beta, x, y, bnScaleData, bnBiasData** is **NULL**.
- The number of **xDesc** or **yDesc** tensor descriptor dimensions is not within the [4,5] range (only 4D and 5D tensors are supported).
- **bnScaleBiasMeanVarDesc** dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Exactly one of **saveMean**, **saveInvVariance** pointers are **NULL**.
- Exactly one of **resultRunningMeanData**, **resultRunningInvVarianceData** pointers are **NULL**.
- **epsilon** value is less than **CUDNN_BN_MIN_EPSILON**.
- Dimensions or data types mismatch for **xDesc**, **yDesc**

## 4.9. cudnnConvolutionBackwardBias

```
cudnnStatus_t cudnnConvolutionBackwardBias(
    cudnnHandle_t                       handle,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       dyDesc,
    const void                         *dy,
    const void                         *beta,
    const cudnnTensorDescriptor_t       dbDesc,
    void                               *db)
```

This function computes the convolution function gradient with respect to the bias, which is the sum of every element belonging to the same feature map across all of the images of the input tensor. Therefore, the number of elements produced is equal to the number of features maps of the input tensor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**dyDesc**

*Input*. Handle to the previously initialized input tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dyDesc**.

**dbDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**db**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **dbDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The operation was launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **n, height, width** of the output tensor is not 1.
- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors is different.

## 4.10. cudnnConvolutionBackwardData

```
cudnnStatus_t cudnnConvolutionBackwardData(
    cudnnHandle_t                       handle,
    const void                         *alpha,
    const cudnnFilterDescriptor_t       wDesc,
```

```
    const void                          *w,
    const cudnnTensorDescriptor_t        dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionBwdDataAlgo_t        algo,
    void                                *workSpace,
    size_t                               workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnTensorDescriptor_t        dxDesc,
    void                                *dx)
```

This function computes the convolution data gradient of the tensor **dy**, where **y** is the output of the forward convolution in **cudnnConvolutionForward()**. It uses the specified **algo**, and returns the results in the output tensor **dx**. Scaling factors **alpha** and **beta** can be used to scale the computed result or accumulate with the current **dx**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor. For more information, see cudnnFilterDescriptor_t.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**dy**

*Input*. Data pointer to GPU memory associated with the input differential tensor descriptor **dyDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor. For more information, see cudnnConvolutionDescriptor_t.

**algo**

*Input*. Enumerant that specifies which backward data convolution algorithm should be used to compute the results. For more information, see cudnnConvolutionBwdDataAlgo_t.

**workSpace**

*Input*. Data pointer to GPU memory to a workspace needed to able to execute the specified algorithm. If no workspace is needed for a particular algorithm, that pointer can be nil.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**dx**

*Input/Output*. Data pointer to GPU memory associated with the output tensor descriptor **dxDesc** that carries the result.

### Supported configurations

This function supports the following combinations of data types for **wDesc**, **dyDesc**, **convDesc**, and **dxDesc**.

| Data Type Configurations | wDesc, dyDesc and dxDesc Data Type | convDesc Data Type |
| --- | --- | --- |
| TRUE_HALF_CONFIG (only supported on architectures with true FP16 support, meaning, compute capability 5.3 and later) | CUDNN_DATA_HALF | CUDNN_DATA_HALF |
| PSEUDO_HALF_CONFIG | CUDNN_DATA_HALF | CUDNN_DATA_FLOAT |
| FLOAT_CONFIG | CUDNN_DATA_FLOAT | CUDNN_DATA_FLOAT |
| DOUBLE_CONFIG | CUDNN_DATA_DOUBLE | CUDNN_DATA_DOUBLE |

### Supported algorithms

Specifying a separate algorithm can cause changes in performance, support and computation determinism. See the following for a list of algorithm options, and their respective supported parameters and deterministic behavior.

The table below shows the list of the supported 2D and 3D convolutions. The 2D convolutions are described first, followed by the 3D convolutions.

For the following terms, the short-form versions shown in the parentheses are used in the table below, for brevity:

- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0 (_ALGO_0)`
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1 (_ALGO_1)`
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT (_FFT)`
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING (_FFT_TILING)`
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD (_WINOGRAD)`
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED (_WINOGRAD_NONFUSED)`
- `CUDNN_TENSOR_NCHW (_NCHW)`
- `CUDNN_TENSOR_NHWC (_NHWC)`
- `CUDNN_TENSOR_NCHW_VECT_C (_NCHW_VECT_C)`

### Table 13 For 2D convolutions: `wDesc: _NHWC`

| Filter descriptor `wDesc: _NHWC` (see cudnnTensorFormat_t) | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| `_ALGO_1` | | NHWC HWC-packed | NHWC HWC-packed | `TRUE_HALF_CONFIG`<br>`PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG` | |

### Table 14 For 2D convolutions: `wDesc: _NCHW`

| Filter descriptor `wDesc: _NCHW`. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| `_ALGO_0` | No | NCHW CHW-packed | All except `_NCHW_VECT_C`. | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` | **Dilation**: greater than 0 for all dimensions<br>`convDesc` Group Count |

| Filter descriptor wDesc: _NCHW. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for dyDesc | Tensor Formats Supported for dxDesc | Data Type Configurations Supported | Important |
| | | | | | Support: Greater than 0 |
| _ALGO_1 | Yes | NCHW CHW-packed | All except _NCHW_VECT_C. | TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG | Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0 |
| _FFT | Yes | NCHW CHW-packed | NCHW HW-packed | PSEUDO_HALF_CONFIG FLOAT_CONFIG | Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0 dxDesc feature map height + 2 * convDesc zero-padding height must equal 256 or less dxDesc feature map width + 2 * convDesc zero-padding width must equal 256 or less convDesc vertical and horizontal filter stride must equal 1 |

| Filter descriptor `wDesc: _NCHW`. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| | | | | | `wDesc` filter height must be greater than `convDesc` zero-padding height<br>`wDesc` filter width must be greater than `convDesc` zero-padding width |
| `_FFT_TILING` | Yes | NCHW CHW-packed | NCHW HW-packed | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` is also supported when the task can be handled by 1D FFT, meaning, one of the filter dimensions, width or height is 1. | **Dilation:** 1 for all dimensions<br>`convDesc` Group Count Support: Greater than 0<br>When neither of `wDesc` filter dimension is 1, the filter width and height must not be larger than 32<br>When either of `wDesc` filter dimension is 1, the largest filter dimension should not exceed 256<br>`convDesc` vertical and |

| Filter descriptor `wDesc: _NCHW`. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| | | | | | horizontal filter stride must equal 1 when either the filter width or filter height is 1, otherwise, the stride can be 1 or 2<br>`wDesc` filter height must be greater than `convDesc` zero-padding height<br>`wDesc` filter width must be greater than `convDesc` zero-padding width |
| `_WINOGRAD` | Yes | NCHW CHW-packed | All except `_NCHW_VECT_C`. | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`convDesc` Group Count Support: Greater than 0<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`wDesc` filter height must be 3 |

| Filter descriptor `wDesc: _NCHW`. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| | | | | | `wDesc` filter width must be 3 |
| `_WINOGRAD_NONFUSED` | Yes | NCHW CHW-packed | All except `_NCHW_VECT_C`. | `TRUE_HALF_CONFIG` `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`convDesc` Group Count Support: Greater than 0<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`wDesc` filter (height, width) must be (3,3) or (5,5)<br>If `wDesc` filter (height, width) is (5,5) then the data type config `TRUE_HALF_CONFIG` is not supported |

## Table 15 For 3D convolutions: `wDesc: _NCHW`

| Filter descriptor `wDesc: _NCHW`. | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| `_ALGO_0` | Yes | NCDHW CDHW-packed | All except `_NCDHW_VECT_C` | `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` `DOUBLE_CONFIG` | **Dilation:** greater than 0 for all dimensions `convDesc` Group Count Support: Greater than 0 |
| `_ALGO_1` | Yes | NCDHW fully-packed | NCDHW fully-packed | `TRUE_HALF_CONFIG` `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` `DOUBLE_CONFIG` | **Dilation**: 1 for all dimensions `convDesc` Group Count Support: Greater than 0 |
| `_FFT_TILING` | Yes | NCDHW CDHW-packed | NCDHW DHW-packed | `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` `DOUBLE_CONFIG` | **Dilation:** 1 for all dimensions `convDesc` Group Count Support: Greater than 0 `wDesc` filter height must equal 16 or less `wDesc` filter width must equal 16 or less `wDesc` filter depth must equal 16 or less `convDesc` must have all filter |

| Filter descriptor **wDesc: _NCHW.** | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for **dyDesc** | Tensor Formats Supported for **dxDesc** | Data Type Configurations Supported | Important |
| | | | | | strides equal to 1<br>**wDesc** filter height must be greater than **convDesc** zero-padding height<br>**wDesc** filter width must be greater than **convDesc** zero-padding width<br>**wDesc** filter depth must be greater than **convDesc** zero-padding width |

### Returns

**CUDNN_STATUS_SUCCESS**

The operation was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- At least one of the following is **NULL**: **handle**, **dyDesc**, **wDesc**, **convDesc**, **dxDesc**, **dy**, **w**, **dx**, **alpha**, **beta**
- **wDesc** and **dyDesc** have a non-matching number of dimensions
- **wDesc** and **dxDesc** have a non-matching number of dimensions
- **wDesc** has fewer than three number of dimensions
- **wDesc**, **dxDesc**, and **dyDesc** have a non-matching data type.
- **wDesc** and **dxDesc** have a non-matching number of input feature maps per image (or group in case of grouped convolutions).

- **dyDesc** spatial sizes do not match with the expected size as determined by **cudnnGetConvolutionNdForwardOutputDim**

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- **dyDesc** or **dxDesc** have a negative tensor striding
- **dyDesc**, **wDesc** or **dxDesc** has a number of dimensions that is not 4 or 5
- The chosen algo does not support the parameters provided; see above for an exhaustive list of parameters that support each algo
- **dyDesc** or **wDesc** indicate an output channel count that isn't a multiple of group count (if group count has been set in **convDesc**).

**CUDNN_STATUS_MAPPING_ERROR**

An error occurs during the texture binding of the filter data or the input differential tensor data

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.11. cudnnConvolutionBackwardFilter

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                       handle,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *x,
    const cudnnTensorDescriptor_t       dyDesc,
    const void                         *dy,
    const cudnnConvolutionDescriptor_t  convDesc,
    cudnnConvolutionBwdFilterAlgo_t     algo,
    void                               *workSpace,
    size_t                              workSpaceSizeInBytes,
    const void                         *beta,
    const cudnnFilterDescriptor_t       dwDesc,
    void                               *dw)
```

This function computes the convolution weight (filter) gradient of the tensor **dy**, where **y** is the output of the forward convolution in **cudnnConvolutionForward()**. It uses the specified **algo**, and returns the results in the output tensor **dw**. Scaling factors **alpha** and **beta** can be used to scale the computed result or accumulate with the current **dw**.

## Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**dy**

*Input*. Data pointer to GPU memory associated with the backpropagation gradient tensor descriptor **dyDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor. For more information, see cudnnConvolutionDescriptor_t.

**algo**

*Input*. Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, see cudnnConvolutionBwdFilterAlgo_t.

**workSpace**

*Input*. Data pointer to GPU memory to a workspace needed to able to execute the specified algorithm. If no workspace is needed for a particular algorithm, that pointer can be nil.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

**dwDesc**

*Input*. Handle to a previously initialized filter gradient descriptor. For more information, see cudnnFilterDescriptor_t.

**dw**

*Input/Output*. Data pointer to GPU memory associated with the filter gradient descriptor **dwDesc** that carries the result.

### Supported configurations

This function supports the following combinations of data types for **xDesc**, **dyDesc**, **convDesc**, and **dwDesc**.

| Data Type Configurations | xDesc, dyDesc, and dwDesc Data Type | convDesc Data Type |
|---|---|---|
| TRUE_HALF_CONFIG (only supported on architectures with true FP16 support, meaning, compute capability 5.3 and later) | CUDNN_DATA_HALF | CUDNN_DATA_HALF |
| PSEUDO_HALF_CONFIG | CUDNN_DATA_HALF | CUDNN_DATA_FLOAT |
| FLOAT_CONFIG | CUDNN_DATA_FLOAT | CUDNN_DATA_FLOAT |
| DOUBLE_CONFIG | CUDNN_DATA_DOUBLE | CUDNN_DATA_DOUBLE |

### Supported algorithms

Specifying a separate algorithm can cause changes in performance, support and computation determinism. See the following table for an exhaustive list of algorithm options and their respective supported parameters and deterministic behavior.

The table below shows the list of the supported 2D and 3D convolutions. The 2D convolutions are described first, followed by the 3D convolutions.

For the following terms, the short-form versions shown in the parentheses are used in the table below, for brevity:

- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0 (_ALGO_0)**
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1 (_ALGO_1)**
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3 (_ALGO_3)**
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT (_FFT)**
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING (_FFT_TILING)**
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED (_WINOGRAD_NONFUSED)**
- **CUDNN_TENSOR_NCHW (_NCHW)**
- **CUDNN_TENSOR_NHWC (_NHWC)**
- **CUDNN_TENSOR_NCHW_VECT_C (_NCHW_VECT_C)**

Table 16 For 2D convolutions: `dwDesc: _NHWC`

| Filter descriptor `dwDesc: _NHWC` (see cudnnTensorFormat_t) | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| `_ALGO_0` and `_ALGO_1` | | NHWC HWC-packed | NHWC HWC-packed | `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` | |

Table 17 For 2D convolutions: `wDesc: _NCHW`

| Filter descriptor `wDesc: _NCHW` | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| `_ALGO_0` | No | All except `_NCHW_VECT_C`. | NCHW CHW-packed | `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` `DOUBLE_CONFIG` | **Dilation:** greater than 0 for all dimensions `convDesc` Group Count Support: Greater than 0 This algo is not supported if output is of type `CUDNN_DATA_HALF` and the number of elements in `dw` is odd. |
| `_ALGO_1` | Yes | `_NCHW` or `_NHWC` | NCHW CHW-packed | `TRUE_HALF_CONFIG` `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` `DOUBLE_CONFIG` | **Dilation:** 1 for all dimensions `convDesc` Group Count |

| Filter descriptor `wDesc: _NCHW` | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for `dyDesc` | Tensor Formats Supported for `dxDesc` | Data Type Configurations Supported | Important |
| | | | | | Support: Greater than 0 |
| `_FFT` | Yes | NCHW CHW-packed | NCHW CHW-packed | `PSEUDO_HALF_CONFIG` `FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`convDesc` Group Count Support: Greater than 0<br>`xDesc` feature map height + 2 * `convDesc` zero-padding height must equal 256 or less<br>`xDesc` feature map width + 2 * `convDesc` zero-padding width must equal 256 or less<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`dwDesc` filter height must be greater than `convDesc` zero-padding height<br>`dwDesc` filter width must be |

| Filter descriptor wDesc: _NCHW | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for dyDesc | Tensor Formats Supported for dxDesc | Data Type Configurations Supported | Important |
| | | | | | greater than convDesc zero-padding width |
| _ALGO_3 | Yes | All except _NCHW_VECT_C | NCHW CHW-packed | PSEUDO_HALF_CONFIG<br>FLOAT_CONFIG<br>DOUBLE_CONFIG | **Dilation:** 1 for all dimensions<br>convDesc Group Count Support: Greater than 0 |
| _WINOGRAD_NONFUSED | Yes | All except _NCHW_VECT_C | NCHW CHW-packed | TRUE_HALF_CONFIG<br>PSEUDO_HALF_CONFIG<br>FLOAT_CONFIG | **Dilation:** 1 for all dimensions<br>convDesc Group Count Support: Greater than 0<br>convDesc vertical and horizontal filter stride must equal 1<br>wDesc filter (height, width) must be (3,3) or (5,5)<br>If wDesc filter (height, width) is (5,5), then the data type config TRUE_HALF_CONFIG is not supported. |

| Filter descriptor wDesc: _NCHW | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for dyDesc | Tensor Formats Supported for dxDesc | Data Type Configurations Supported | Important |
| _FFT_TILING | Yes | NCHW CHW-packed | NCHW CHW-packed | PSEUDO_HALF_CONFIG<br>FLOAT_CONFIG<br>DOUBLE_CONFIG | **Dilation:** 1 for all dimensions<br>convDesc Group Count Support: Greater than 0<br>xDesc width or height must equal 1<br>dyDesc width or height must equal 1 (the same dimension as in xDesc). The other dimension must be less than or equal to 256, meaning, the largest 1D tile size currently supported.<br>convDesc vertical and horizontal filter stride must equal 1<br>dwDesc filter height must be greater than convDesc zero-padding height |

| Filter descriptor wDesc: _NCHW | | | | | |
|---|---|---|---|---|---|
| Algo Name | Deterministic (Yes or No) | Tensor Formats Supported for dyDesc | Tensor Formats Supported for dxDesc | Data Type Configurations Supported | Important |
| | | | | | dwDesc filter width must be greater than convDesc zero-padding width |

Table 18 For 3D convolutions: **wDesc: _NCHW**

| Filter descriptor wDesc: _NCHW. | | | | | |
|---|---|---|---|---|---|
| Algo Name (3D Convolutions) | Deterministic (Yes or No) | Tensor Formats Supported for dyDesc | Tensor Formats Supported for dxDesc | Data Type Configurations Supported | Important |
| _ALGO_0 | No | All except _NCDHW_VECT_C | NCDHW CDHW-packed | PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG | **Dilation:** greater than 0 for all dimensions convDesc Group Count Support: Greater than 0 |
| _ALGO_3 | No | NCDHW fully-packed | NCDHW fully-packed | PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG | **Dilation:** greater than 0 for all dimensions convDesc Group Count Support: Greater than 0 |

### Returns

**CUDNN_STATUS_SUCCESS**

The operation was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- At least one of the following is NULL: **handle**, **xDesc**, **dyDesc**, **convDesc**, **dwDesc**, **xData**, **dyData**, **dwData**, **alpha**, **beta**
- **xDesc** and **dyDesc** have a non-matching number of dimensions
- **xDesc** and **dwDesc** have a non-matching number of dimensions
- **xDesc** has fewer than three number of dimensions
- **xDesc**, **dyDesc**, and **dwDesc** have a non-matching data type.
- **xDesc** and **dwDesc** have a non-matching number of input feature maps per image (or group in case of grouped convolutions).
- **yDesc** or **wDesc** indicate an output channel count that isn't a multiple of group count (if group count has been set in **convDesc**).

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- **xDesc** or **dyDesc** have negative tensor striding
- **xDesc**, **dyDesc** or **dwDesc** has a number of dimensions that is not 4 or 5
- The chosen algo does not support the parameters provided; see above for exhaustive list of parameter support for each algo

**CUDNN_STATUS_MAPPING_ERROR**

An error occurs during the texture binding of the filter data.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.12. cudnnConvolutionBiasActivationForward

```
cudnnStatus_t cudnnConvolutionBiasActivationForward(
    cudnnHandle_t                        handle,
    const void                          *alpha1,
    const cudnnTensorDescriptor_t        xDesc,
    const void                          *x,
    const cudnnFilterDescriptor_t        wDesc,
    const void                          *w,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionFwdAlgo_t            algo,
    void                                *workSpace,
    size_t                               workSpaceSizeInBytes,
    const void                          *alpha2,
    const cudnnTensorDescriptor_t        zDesc,
    const void                          *z,
    const cudnnTensorDescriptor_t        biasDesc,
    const void                          *bias,
    const cudnnActivationDescriptor_t    activationDesc,
    const cudnnTensorDescriptor_t        yDesc,
    void                                *y)
```

This function applies a bias and then an activation to the convolutions or cross-correlations of cudnnConvolutionForward(), returning results in **`y`**. The full computation follows the equation **`y = act ( alpha1 * conv(x) + alpha2 * z + bias )`**.

- The routine **`cudnnGetConvolution2dForwardOutputDim`** or **`cudnnGetConvolutionNdForwardOutputDim`** can be used to determine the proper dimensions of the output tensor descriptor **`yDesc`** with respect to **`xDesc`**, **`convDesc`**, and **`wDesc`**.
- Only the **`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`** algo is enabled with **`CUDNN_ACTIVATION_IDENTITY`**. In other words, in the **`cudnnActivationDescriptor_t`** structure of the input **`activationDesc`**, if the mode of the **`cudnnActivationMode_t`** field is set to the enum value **`CUDNN_ACTIVATION_IDENTITY`**, then the input **`cudnnConvolutionFwdAlgo_t`** of this function **`cudnnConvolutionBiasActivationForward()`** must be set to the enum value **`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`**. For more information, see **`cudnnSetActivationDescriptor()`**.

### Parameters

**`handle`**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**`alpha1, alpha2`**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**`xDesc`**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**`x`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`xDesc`**.

**`wDesc`**

*Input*. Handle to a previously initialized filter descriptor. For more information, see cudnnFilterDescriptor_t.

**`w`**

*Input*. Data pointer to GPU memory associated with the filter descriptor **`wDesc`**.

**`convDesc`**

*Input*. Previously initialized convolution descriptor. For more information, see cudnnConvolutionDescriptor_t.

**algo**

*Input*. Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, see cudnnConvolutionFwdAlgo_t.

**workSpace**

*Input*. Data pointer to GPU memory to a workspace needed to able to execute the specified algorithm. If no workspace is needed for a particular algorithm, that pointer can be nil.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

**zDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**z**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **zDesc**.

**biasDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**bias**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **biasDesc**.

**activationDesc**

*Input*. Handle to a previously initialized activation descriptor. For more information, see cudnnActivationDescriptor_t.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**y**

*Input/Output*. Data pointer to GPU memory associated with the tensor descriptor **yDesc** that carries the result of the convolution.

For the convolution step, this function supports the specific combinations of data types for **xDesc**, **wDesc**, **convDesc**, and **yDesc** as listed in the documentation of **cudnnConvolutionForward()**. The following table specifies the supported combinations of data types for **x**, **y**, **z**, **bias**, and **alpha1/alpha2**.

### Table 19 Supported combinations of data types ( **X** = **CUDNN_DATA**)

| x | w | y and z | bias | alpha1/alpha2 |
|---|---|---|---|---|
| X_DOUBLE | X_DOUBLE | X_DOUBLE | X_DOUBLE | X_DOUBLE |
| X_FLOAT | X_FLOAT | X_FLOAT | X_FLOAT | X_FLOAT |

| x | w | y and z | bias | alpha1/alpha2 |
|---|---|---|---|---|
| X_HALF | X_HALF | X_HALF | X_HALF | X_FLOAT |
| X_INT8 | X_INT8 | X_INT8 | X_FLOAT | X_FLOAT |
| X_INT8 | X_INT8 | X_FLOAT | X_FLOAT | X_FLOAT |
| X_INT8x4 | X_INT8x4 | X_INT8x4 | X_FLOAT | X_FLOAT |
| X_INT8x4 | X_INT8x4 | X_FLOAT | X_FLOAT | X_FLOAT |
| X_UINT8 | X_INT8 | X_INT8 | X_FLOAT | X_FLOAT |
| X_UINT8 | X_INT8 | X_FLOAT | X_FLOAT | X_FLOAT |
| X_UINT8x4 | X_INT8x4 | X_INT8x4 | X_FLOAT | X_FLOAT |
| X_UINT8x4 | X_INT8x4 | X_FLOAT | X_FLOAT | X_FLOAT |

### Returns

In addition to the error values listed by the documentation of **cudnnConvolutionForward()**, the possible error values returned by this function and their meanings are listed below.

**CUDNN_STATUS_SUCCESS**

The operation was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- At least one of the following is **NULL**: **zDesc**, **zData**, **biasDesc**, **bias**, **activationDesc**.
- The second dimension of **biasDesc** and the first dimension of **filterDesc** are not equal.
- **zDesc** and **destDesc** do not match.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. Some examples of non-supported configurations are as follows:

- The **mode** of **activationDesc** is neither **CUDNN_ACTIVATION_RELU** or **CUDNN_ACTIVATION_IDENTITY**.
- The **reluNanOpt** of **activationDesc** is not **CUDNN_NOT_PROPAGATE_NAN**.
- The second stride of **biasDesc** is not equal to one.
- The data type of **biasDesc** does not correspond to the data type of **yDesc** as listed in the above data types table.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.13. cudnnConvolutionForward

```
cudnnStatus_t cudnnConvolutionForward(
    cudnnHandle_t                       handle,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *x,
    const cudnnFilterDescriptor_t       wDesc,
    const void                         *w,
    const cudnnConvolutionDescriptor_t  convDesc,
    cudnnConvolutionFwdAlgo_t           algo,
    void                               *workSpace,
    size_t                              workSpaceSizeInBytes,
    const void                         *beta,
    const cudnnTensorDescriptor_t       yDesc,
    void                               *y)
```

This function executes convolutions or cross-correlations over **x** using filters specified with **w**, returning results in **y**. Scaling factors **alpha** and **beta** can be used to scale the input tensor and the output tensor respectively.

The routine **cudnnGetConvolution2dForwardOutputDim** or **cudnnGetConvolutionNdForwardOutputDim** can be used to determine the proper dimensions of the output tensor descriptor **yDesc** with respect to **xDesc**, **convDesc**, and **wDesc**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor. For more information, see cudnnFilterDescriptor_t.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor. For more information, see cudnnConvolutionDescriptor_t.

**algo**

*Input*. Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, see cudnnConvolutionFwdAlgo_t.

**workSpace**

*Input*. Data pointer to GPU memory to a workspace needed to able to execute the specified algorithm. If no workspace is needed for a particular algorithm, that pointer can be nil.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**y**

*Input/Output*. Data pointer to GPU memory associated with the tensor descriptor **yDesc** that carries the result of the convolution.

### Supported configurations

This function supports the following combinations of data types for **xDesc**, **wDesc**, **convDesc**, and **yDesc**.

## Table 20 Supported configurations

| Data Type Configurations | xDesc and wDesc | convDesc | yDesc |
|---|---|---|---|
| TRUE_HALF_CONFIG (only supported on architectures with true FP16 support, meaning, compute capability 5.3 and later) | CUDNN_DATA_HALF | CUDNN_DATA_HALF | CUDNN_DATA_HALF |
| PSEUDO_HALF_CONFIG | CUDNN_DATA_HALF | CUDNN_DATA_FLOAT | CUDNN_DATA_HALF |
| FLOAT_CONFIG | CUDNN_DATA_FLOAT | CUDNN_DATA_FLOAT | CUDNN_DATA_FLOAT |

| Data Type Configurations | xDesc and wDesc | convDesc | yDesc |
|---|---|---|---|
| DOUBLE_CONFIG | CUDNN_DATA_DOUBLE | CUDNN_DATA_DOUBLE | CUDNN_DATA_DOUBLE |
| INT8_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later) | CUDNN_DATA_INT8 | CUDNN_DATA_INT32 | CUDNN_DATA_INT8 |
| INT8_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later) | CUDNN_DATA_INT8 | CUDNN_DATA_INT32 | CUDNN_DATA_FLOAT |
| INT8x4_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later) | CUDNN_DATA_INT8x4 | CUDNN_DATA_INT32 | CUDNN_DATA_INT8x4 |
| INT8x4_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later) | CUDNN_DATA_INT8x4 | CUDNN_DATA_INT32 | CUDNN_DATA_FLOAT |
| UINT8x4_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later) | CUDNN_DATA_UINT8x4 | CUDNN_DATA_INT32 | CUDNN_DATA_UINT8x4 |
| UINT8x4_EXT_CONFIG (only supported | CUDNN_DATA_UINT8x4 | CUDNN_DATA_INT32 | CUDNN_DATA_FLOAT |

| Data Type Configurations | xDesc and wDesc | convDesc | yDesc |
|---|---|---|---|
| on architectures with DP4A support, meaning, compute capability 6.1 and later) | | | |

**Supported algorithms**

For this function, all algorithms perform deterministic computations. Specifying a separate algorithm can cause changes in performance and support.

The table below shows the list of the supported 2D and 3D convolutions. The 2D convolutions are described first, followed by the 3D convolutions.

For the following terms, the short-form versions shown in the paranthesis are used in the table below, for brevity:

- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM (_IMPLICIT_GEMM)
- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM (_IMPLICIT_PRECOMP_GEMM)
- CUDNN_CONVOLUTION_FWD_ALGO_GEMM (_GEMM)
- CUDNN_CONVOLUTION_FWD_ALGO_DIRECT (_DIRECT)
- CUDNN_CONVOLUTION_FWD_ALGO_FFT (_FFT)
- CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING (_FFT_TILING)
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD (_WINOGRAD)
- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED (_WINOGRAD_NONFUSED)
- CUDNN_TENSOR_NCHW (_NCHW)
- CUDNN_TENSOR_NHWC (_NHWC)
- CUDNN_TENSOR_NCHW_VECT_C (_NCHW_VECT_C)

Table 21 For 2D convolutions: **wDesc: _NCHW**

| **Filter descriptor `wDesc: _NCHW` (see cudnnTensorFormat_t)**<br>**`convDesc` Group count support: Greater than 0, for all algos.** | | | | |
|---|---|---|---|---|
| **Algo Name** | **Tensor Formats Supported for `xDesc`** | **Tensor Formats Supported for `yDesc`** | **Data Type Configurations Supported** | **Important** |
| `_IMPLICIT_GEMM` | All except `_NCHW_VECT_C`. | All except `_NCHW_VECT_C`. | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` | **Dilation:** Greater than 0 for all dimensions |
| `_IMPLICIT_PRECOMP_GEMM` | | | `TRUE_HALF_CONFIG`<br>`PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` | **Dilation:** 1 for all dimensions |
| `_GEMM` | | | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` | **Dilation:** 1 for all dimensions |
| `_FFT` | NCHW HW-packed | NCHW HW-packed | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`xDesc` feature map height + 2 * `convDesc` zero-padding height must equal 256 or less<br>`xDesc` feature map width + 2 * `convDesc` zero-padding width must equal 256 or less<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`wDesc` filter height must be greater |

| Filter descriptor `wDesc`: `_NCHW` (see cudnnTensorFormat_t)<br>`convDesc` Group count support: Greater than 0, for all algos. | | | | |
|---|---|---|---|---|
| Algo Name | Tensor Formats Supported for `xDesc` | Tensor Formats Supported for `yDesc` | Data Type Configurations Supported | Important |
| | | | | than `convDesc` zero-padding height<br>`wDesc` filter width must be greater than `convDesc` zero-padding width |
| `_FFT_TILING` | | | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG`<br>`DOUBLE_CONFIG` is also supported when the task can be handled by 1D FFT, meaning, one of the filter dimension, width or height is 1. | **Dilation:** 1 for all dimensions<br>When neither of `wDesc` filter dimension is 1, the filter width and height must not be larger than 32<br>When either of `wDesc` filter dimension is 1, the largest filter dimension should not exceed 256<br>`convDesc` vertical and horizontal filter stride must equal 1 when either the filter width or filter height is 1, otherwise the stride can be a 1 or 2<br>`wDesc` filter height must be greater than `convDesc` |

| Filter descriptor `wDesc: _NCHW` (see cudnnTensorFormat_t) `convDesc` Group count support: Greater than 0, for all algos. | | | | |
|---|---|---|---|---|
| Algo Name | Tensor Formats Supported for `xDesc` | Tensor Formats Supported for `yDesc` | Data Type Configurations Supported | Important |
| | | | | zero-padding height<br>`wDesc` filter width must be greater than `convDesc` zero-padding width |
| `_WINOGRAD` | All except `_NCHW_VECT_C`. | All except `_NCHW_VECT_C`. | `PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`wDesc` filter height must be 3<br>`wDesc` filter width must be 3 |
| `_WINOGRAD_NONFUSED` | | | `TRUE_HALF_CONFIG`<br>`PSEUDO_HALF_CONFIG`<br>`FLOAT_CONFIG` | **Dilation:** 1 for all dimensions<br>`convDesc` vertical and horizontal filter stride must equal 1<br>`wDesc` filter (height, width) must be (3,3) or (5,5)<br>If `wDesc` filter (height, width) is (5,5), then data type config `TRUE_HALF_CONFIG` is not supported. |
| `_DIRECT` | Currently not implemented in cuDNN. | | | |

Table 22 For 2D convolutions: **wDesc: _NCHWC**

| Filter descriptor `wDesc: _NCHWC`<br>`convDesc` Group count support: Greater than 0. | | | | |
|---|---|---|---|---|
| Algo Name | xDesc | yDesc | Data Type Configurations Supported | Important |
| _IMPLICIT_GEMM | NCHWC HWC-packed | NCHWC HWC-packed | PSEUDO_HALF_CONFIG<br>FLOAT_CONFIG | **Dilation:** Greater than 0 for all dimensions |

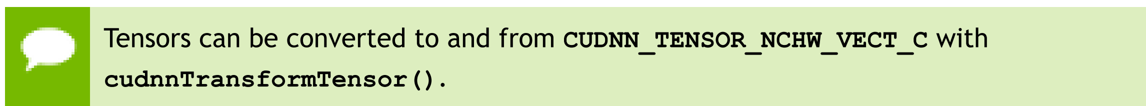
Table 23 For 2D convolutions: **wDesc: _NHWC**

| Filter descriptor `wDesc: _NHWC`<br>`convDesc` Group count support: Greater than 0. | | | | |
|---|---|---|---|---|
| Algo Name | xDesc | yDesc | Data Type Configurations Supported | Important |
| _IMPLICIT_PRECOMP_GEMM | NHWC | NHWC | INT8_CONFIG<br>INT8_EXT_CONFIG<br>INT8x4_CONFIG<br>INT8x4_EXT_CONFIG<br>UINT8x4_CONFIG<br>UINT8x4_EXT_CONFIG | **Dilation:** 1 for all dimensions<br>Input and output features maps must be a multiple of 4. |

Table 24 For 3D convolutions: **wDesc: _NCHW**

| Filter descriptor `wDesc: _NCHW`<br>`convDesc` Group count support: Greater than 0, for all algos. | | | | |
|---|---|---|---|---|
| Algo Name | xDesc | yDesc | Data Type Configurations Supported | Important |
| _IMPLICIT_GEMM | All except _NCHW_VECT_C. | All except _NCHW_VECT_C. | PSEUDO_HALF_CONFIG<br>FLOAT_CONFIG<br>DOUBLE_CONFIG | **Dilation:** Greater than 0 for all dimensions |
| _IMPLICIT_PRECOMP_GEMM | | | | **Dilation:** 1 for all dimensions |

| Filter descriptor `wDesc: _NCHW` `convDesc` Group count support: Greater than 0, for all algos. | | | | |
|---|---|---|---|---|
| Algo Name | `xDesc` | `yDesc` | Data Type Configurations Supported | Important |
| `_FFT_TILING` | NCDHW DHW-packed | NCDHW DHW-packed | | **Dilation:** 1 for all dimensions<br>`wDesc` filter height must equal 16 or less<br>`wDesc` filter width must equal 16 or less<br>`wDesc` filter depth must equal 16 or less<br>`convDesc` must have all filter strides equal to 1<br>`wDesc` filter height must be greater than `convDesc` zero-padding height<br>`wDesc` filter width must be greater than `convDesc` zero-padding width<br>`wDesc` filter depth must be greater than `convDesc` zero-padding width |


Tensors can be converted to and from `CUDNN_TENSOR_NCHW_VECT_C` with `cudnnTransformTensor()`.

### Returns

**CUDNN_STATUS_SUCCESS**

The operation was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- At least one of the following is **NULL**: handle, **xDesc**, **wDesc**, **convDesc**, **yDesc**, **xData**, **w**, **yData**, **alpha**, **beta**
- **xDesc** and **yDesc** have a non-matching number of dimensions
- **xDesc** and **wDesc** have a non-matching number of dimensions
- **xDesc** has fewer than three number of dimensions
- **xDesc**'s number of dimensions is not equal to **convDesc** array length + 2
- **xDesc** and **wDesc** have a non-matching number of input feature maps per image (or group in case of grouped convolutions)
- **yDesc** or **wDesc** indicate an output channel count that isn't a multiple of group count (if group count has been set in **convDesc**).
- **xDesc**, **wDesc**, and **yDesc** have a non-matching data type
- For some spatial dimension, **wDesc** has a spatial size that is larger than the input spatial size (including zero-padding size)

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- **xDesc** or **yDesc** have negative tensor striding
- **xDesc**, **wDesc**, or **yDesc** has a number of dimensions that is not 4 or 5
- **yDesc** spatial sizes do not match with the expected size as determined by **cudnnGetConvolutionNdForwardOutputDim**
- The chosen algo does not support the parameters provided; see above for an exhaustive list of parameters supported for each algo

**CUDNN_STATUS_MAPPING_ERROR**

An error occurred during the texture binding of the filter data.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.14. cudnnCreate

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

This function initializes the cuDNN library and creates a handle to an opaque structure holding the cuDNN library context. It allocates hardware resources on the host and device and must be called prior to making any other cuDNN library calls.

The cuDNN library handle is tied to the current CUDA device (context). To use the library on multiple devices, one cuDNN handle needs to be created for each device.

For a given device, multiple cuDNN handles with different configurations (for example, different current CUDA streams) may be created. Because **`cudnnCreate`** allocates some internal resources, the release of those resources by calling **`cudnnDestroy`** will implicitly call cudaDeviceSynchronize; therefore, the recommended best practice is to call **`cudnnCreate/cudnnDestroy`** outside of performance-critical code paths.

For multithreaded applications that use the same device from different threads, the recommended programming model is to create one (or a few, as is convenient) cuDNN handle(s) per thread and use that cuDNN handle for the entire life of the thread.

### Parameters

**`handle`**

*Output*. Pointer to pointer where to store the address to the allocated cuDNN handle. For more information, see cudnnHandle_t.

### Returns

**`CUDNN_STATUS_BAD_PARAM`**

Invalid (**`NULL`**) input pointer supplied.

**`CUDNN_STATUS_NOT_INITIALIZED`**

No compatible GPU found, CUDA driver not installed or disabled, CUDA runtime API initialization failed.

**`CUDNN_STATUS_ARCH_MISMATCH`**

NVIDIA GPU architecture is too old.

**`CUDNN_STATUS_ALLOC_FAILED`**

Host memory allocation failed.

**`CUDNN_STATUS_INTERNAL_ERROR`**

CUDA resource allocation failed.

**`CUDNN_STATUS_LICENSE_ERROR`**

cuDNN license validation failed (only when the feature is enabled).

**`CUDNN_STATUS_SUCCESS`**

cuDNN handle was created successfully.

## 4.15. cudnnCreateActivationDescriptor

```
cudnnStatus_t cudnnCreateActivationDescriptor(
        cudnnActivationDescriptor_t   *activationDesc)
```

This function creates an activation descriptor object by allocating the memory needed to hold its opaque structure. For more information, see cudnnActivationDescriptor_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.16. cudnnCreateAlgorithmDescriptor

```
cudnnStatus_t cudnnCreateAlgorithmDescriptor(
    cudnnAlgorithmDescriptor_t *algoDesc)
```

This function creates an algorithm descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.17. cudnnCreateAlgorithmPerformance

```
cudnnStatus_t cudnnCreateAlgorithmPerformance(
    cudnnAlgorithmPerformance_t *algoPerf,
    int                          numberToCreate)
```

This function creates multiple algorithm performance objects by allocating the memory needed to hold their opaque structures.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.18. cudnnCreateAttnDescriptor

```
cudnnStatus_t cudnnCreateAttnDescriptor(cudnnAttnDescriptor_t *attnDesc);
```

This function creates one instance of an opaque attention descriptor object by allocating the host memory for it and initializing all descriptor fields. The function writes **NULL** to **attnDesc** when the attention descriptor object cannot be allocated.

Use the **cudnnSetAttnDescriptor()** function to configure the attention descriptor and **cudnnDestroyAttnDescriptor()** to destroy it and release the allocated memory.

### Parameters

**attnDesc**

*Output*. Pointer where the address to the newly created attention descriptor should be written.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor object was created successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was encountered (**attnDesc=NULL**).

**CUDNN_STATUS_ALLOC_FAILED**

The memory allocation failed.

# 4.19. cudnnCreateConvolutionDescriptor

```
cudnnStatus_t cudnnCreateConvolutionDescriptor(
    cudnnConvolutionDescriptor_t *convDesc)
```

This function creates a convolution descriptor object by allocating the memory needed to hold its opaque structure. For more information, see cudnnConvolutionDescriptor_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.20. cudnnCreateCTCLossDescriptor

```
cudnnStatus_t cudnnCreateCTCLossDescriptor(
    cudnnCTCLossDescriptor_t* ctcLossDesc)
```

This function creates a CTC loss function descriptor.

### Parameters

**ctcLossDesc**

*Output*. CTC loss descriptor to be set. For more information, see cudnnCTCLossDescriptor_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

CTC loss descriptor passed to the function is invalid.

**CUDNN_STATUS_ALLOC_FAILED**

Memory allocation for this CTC loss descriptor failed.

## 4.21. cudnnCreateDropoutDescriptor

```
cudnnStatus_t cudnnCreateDropoutDescriptor(
    cudnnDropoutDescriptor_t    *dropoutDesc)
```

This function creates a generic dropout descriptor object by allocating the memory needed to hold its opaque structure. For more information, see cudnnDropoutDescriptor_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.22. cudnnCreateFilterDescriptor

```
cudnnStatus_t cudnnCreateFilterDescriptor(
    cudnnFilterDescriptor_t *filterDesc)
```

This function creates a filter descriptor object by allocating the memory needed to hold its opaque structure. For more information, see cudnnFilterDescriptor_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.23. cudnnCreateFusedOpsConstParamPack

```
cudnnStatus_t cudnnCreateFusedOpsConstParamPack(
 cudnnFusedOpsConstParamPack_t *constPack,
 cudnnFusedOps_t ops);
```

This function creates an opaque structure to store the various problem size information, such as the shape, layout and the type of tensors, and the descriptors for convolution and activation, for the selected sequence of **cudnnFusedOps** computations.

### Parameters

**constPack**

*Input*. The opaque structure that is created by this function. For more information, see cudnnFusedOpsConstParamPack_t.

**ops**

*Input*. The specific sequence of computations to perform in the **cudnnFusedOps** computations, as defined in the enumerant type cudnnFusedOps_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

If either **constPack** or **ops** is **NULL**.

**CUDNN_STATUS_SUCCESS**

If the descriptor is created successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

If the **ops** enum value is not supported or reserved for future use.

## 4.24. cudnnCreateFusedOpsPlan

```
cudnnStatus_t cudnnCreateFusedOpsPlan(
 cudnnFusedOpsPlan_t *plan,
 cudnnFusedOps_t ops);
```

This function creates the plan descriptor for the **cudnnFusedOps** computation. This descriptor contains the plan information, including the problem type and size, which kernels should be run, and the internal workspace partition.

### Parameters

**plan**

*Input*. A pointer to the instance of the descriptor created by this function.

**ops**

*Input*. The specific sequence of fused operations computations for which this plan descriptor should be created. For more information, see cudnnFusedOps_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

If either the input ***plan** is **NULL** or the **ops** input is not a valid **cudnnFusedOp** enum.

**CUDNN_STATUS_NOT_SUPPORTED**

The **ops** input provided is not supported.

**CUDNN_STATUS_SUCCESS**

The plan descriptor is created successfully.

## 4.25. cudnnCreateFusedOpsVariantParamPack

```
cudnnStatus_t cudnnCreateFusedOpsVariantParamPack(
 cudnnFusedOpsVariantParamPack_t *varPack,
 cudnnFusedOps_t ops);
```

This function creates a descriptor for **cudnnFusedOps** constant parameters.

### Parameters

**varPack**

*Input*. Pointer to the descriptor created by this function. For more information, see cudnnFusedOpsVariantParamPack_t.

**ops**

*Input*. The specific sequence of fused operations computations for which this descriptor should be created.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor is successfully created.

**CUDNN_STATUS_BAD_PARAM**

If any input is invalid.

## 4.26. cudnnCreateLRNDescriptor

```
cudnnStatus_t cudnnCreateLRNDescriptor(
            cudnnLRNDescriptor_t    *poolingDesc)
```

This function allocates the memory needed to hold the data needed for LRN and **DivisiveNormalization** layers operation and returns a descriptor used with subsequent layer forward and backward calls.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.27. cudnnCreateOpTensorDescriptor

```
cudnnStatus_t cudnnCreateOpTensorDescriptor(
    cudnnOpTensorDescriptor_t*  opTensorDesc)
```

This function creates a tensor pointwise math descriptor. For more information, see cudnnOpTensorDescriptor_t.

### Parameters

**opTensorDesc**

*Output*. Pointer to the structure holding the description of the Tensor Pointwise math such as add, multiply, and more.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

Tensor pointwise math descriptor passed to the function is invalid.

**CUDNN_STATUS_ALLOC_FAILED**

Memory allocation for this tensor pointwise math descriptor failed.

# 4.28. cudnnCreatePersistentRNNPlan

```
cudnnStatus_t cudnnCreatePersistentRNNPlan(
    cudnnRNNDescriptor_t        rnnDesc,
    const int                   minibatch,
    const cudnnDataType_t       dataType,
    cudnnPersistentRNNPlan_t   *plan)
```

This function creates a plan to execute persistent RNNs when using the **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** algo. This plan is tailored to the current GPU and problem hyperparameters. This function call is expected to be expensive

in terms of runtime and should be used infrequently. For more information, see cudnnRNNDescriptor_t, cudnnDataType_t, and cudnnPersistentRNNPlan_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

**CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING**

A prerequisite runtime library cannot be found.

**CUDNN_STATUS_NOT_SUPPORTED**

The current hyperparameters are invalid.

## 4.29. cudnnCreatePoolingDescriptor

```
cudnnStatus_t cudnnCreatePoolingDescriptor(
    cudnnPoolingDescriptor_t    *poolingDesc)
```

This function creates a pooling descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.30. cudnnCreateReduceTensorDescriptor

```
cudnnStatus_t cudnnCreateReduceTensorDescriptor(
 cudnnReduceTensorDescriptor_t* reduceTensorDesc)
```

This function creates a reduce tensor descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_BAD_PARAM**

**reduceTensorDesc** is a **NULL** pointer.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.31. cudnnCreateRNNDataDescriptor

```
cudnnStatus_t cudnnCreateRNNDataDescriptor(
    cudnnRNNDataDescriptor_t *RNNDataDesc)
```

This function creates a RNN data descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The RNN data descriptor object was created successfully.

**CUDNN_STATUS_BAD_PARAM**

**RNNDataDesc** is **NULL**.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.32. cudnnCreateRNNDescriptor

```
cudnnStatus_t cudnnCreateRNNDescriptor(
    cudnnRNNDescriptor_t    *rnnDesc)
```

This function creates a generic RNN descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

# 4.33. cudnnCreateSeqDataDescriptor

```
cudnnStatus_t cudnnCreateSeqDataDescriptor(cudnnSeqDataDescriptor_t
 *seqDataDesc);
```

This function creates one instance of an opaque sequence data descriptor object by allocating the host memory for it and initializing all descriptor fields. The function writes **NULL** to **seqDataDesc** when the sequence data descriptor object cannot be allocated.

Use the **cudnnSetSeqDataDescriptor()** function to configure the sequence data descriptor and **cudnnDestroySeqDataDescriptor()** to destroy it and release the allocated memory.

### Parameters

**seqDataDesc**
*Output*. Pointer where the address to the newly created sequence data descriptor should be written.

### Returns

**CUDNN_STATUS_SUCCESS**
The descriptor object was created successfully.

**CUDNN_STATUS_BAD_PARAM**
An invalid input argument was encountered (**seqDataDesc=NULL**).

**CUDNN_STATUS_ALLOC_FAILED**
The memory allocation failed.

## 4.34. cudnnCreateSpatialTransformerDescriptor

```
cudnnStatus_t cudnnCreateSpatialTransformerDescriptor(
    cudnnSpatialTransformerDescriptor_t *stDesc)
```

This function creates a generic spatial transformer descriptor object by allocating the memory needed to hold its opaque structure.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

## 4.35. cudnnCreateTensorDescriptor

```
cudnnStatus_t cudnnCreateTensorDescriptor(
    cudnnTensorDescriptor_t *tensorDesc)
```

This function creates a generic tensor descriptor object by allocating the memory needed to hold its opaque structure. The data is initialized to all zeros.

### Parameters

**tensorDesc**

*Input*. Pointer to pointer where the address to the allocated tensor descriptor object should be stored.

### Returns

**CUDNN_STATUS_BAD_PARAM**

Invalid input argument.

**CUDNN_STATUS_ALLOC_FAILED**

The resources could not be allocated.

**CUDNN_STATUS_SUCCESS**

The object was created successfully.

## 4.36. cudnnCreateTensorTransformDescriptor

```
cudnnStatus_t cudnnCreateTensorTransformDescriptor(
 cudnnTensorTransformDescriptor_t *transformDesc);
```

This function creates a Tensor transform descriptor object by allocating the memory needed to hold its opaque structure. The Tensor data is initialized to be all zero. Use the cudnnSetTensorTransformDescriptor function to initialize the descriptor created by this function.

### Parameters

**transformDesc**

*Output*. A pointer to an uninitialized tensor transform descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor object was created successfully.

**CUDNN_STATUS_BAD_PARAM**

The **transformDesc** is **NULL**.

**CUDNN_STATUS_ALLOC_FAILED**

The memory allocation failed.

## 4.37. cudnnCTCLoss

```
cudnnStatus_t cudnnCTCLoss(
    cudnnHandle_t                        handle,
    const   cudnnTensorDescriptor_t      probsDesc,
    const   void                        *probs,
```

```
    const   int                          *labels,
    const   int                          *labelLengths,
    const   int                          *inputLengths,
    void                                 *costs,
    const   cudnnTensorDescriptor_t       gradientsDesc,
    const   void                         *gradients,
    cudnnCTCLossAlgo_t                    algo,
    const   cudnnCTCLossDescriptor_t      ctcLossDesc,
    void                                 *workspace,
    size_t                               *workSpaceSizeInBytes)
```

This function returns the CTC costs and gradients, given the probabilities and labels.

This function has an inconsistent interface, for example, the **probs** input is probability normalized by softmax, but the **gradients** output is with respect to the unnormalized activation.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**probsDesc**

*Input*. Handle to the previously initialized probabilities tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**probs**

*Input*. Pointer to a previously initialized probabilities tensor. These input probabilities are normalized by softmax.

**labels**

*Input*. Pointer to a previously initialized labels list.

**labelLengths**

*Input*. Pointer to a previously initialized lengths list, to walk the above labels list.

**inputLengths**

*Input*. Pointer to a previously initialized list of the lengths of the timing steps in each batch.

**costs**

*Output*. Pointer to the computed costs of CTC.

**gradientsDesc**

*Input*. Handle to a previously initialized gradients tensor descriptor.

**gradients**

*Output*. Pointer to the computed gradients of CTC. These computed gradient outputs are with respect to the unnormalized activation.

**algo**

*Input*. Enumerant that specifies the chosen CTC loss algorithm. For more information, see cudnnCTCLossAlgo_t.

**ctcLossDesc**

*Input*. Handle to the previously initialized CTC loss descriptor. For more information, see cudnnCTCLossDescriptor_t.

**workspace**

*Input*. Pointer to GPU memory of a workspace needed to able to execute the specified algorithm.

**sizeInBytes**

*Input*. Amount of GPU memory needed as workspace to be able to execute the CTC loss computation with the specified **algo**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions of **probsDesc** do not match the dimensions of **gradientsDesc**.
- The **inputLengths** do not agree with the first dimension of **probsDesc**.
- The **workSpaceSizeInBytes** is not sufficient.
- The **labelLengths** is greater than 256.

**CUDNN_STATUS_NOT_SUPPORTED**

A compute or data type other than **FLOAT** was chosen, or an unknown algorithm type was chosen.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.38. cudnnDeriveBNTensorDescriptor

```
cudnnStatus_t cudnnDeriveBNTensorDescriptor(
    cudnnTensorDescriptor_t         derivedBnDesc,
    const cudnnTensorDescriptor_t   xDesc,
    cudnnBatchNormMode_t            mode)
```

This function derives a secondary tensor descriptor for the batch normalization **scale**, **invVariance**, **bnBias**, and **bnScale** subtensors from the layer's **x** data descriptor.

Use the tensor descriptor produced by this function as the **bnScaleBiasMeanVarDesc** parameter for the cudnnBatchNormalizationForwardInference and cudnnBatchNormalizationForwardTraining functions, and as the **bnScaleBiasDiffDesc** parameter in the cudnnBatchNormalizationBackward function.

The resulting dimensions will be:

- 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for **BATCHNORM_MODE_SPATIAL**
- 1xCxHxW for 4D and 1xCxDxHxW for 5D for **BATCHNORM_MODE_PER_ACTIVATION** mode

For **HALF** input data type the resulting tensor descriptor will have a **FLOAT** type. For other data types, it will have the same type as the input data.

- Only 4D and 5D tensors are supported.
- The **derivedBnDesc** should be first created using cudnnCreateTensorDescriptor.
- **xDesc** is the descriptor for the layer's **x** data and has to be setup with proper dimensions prior to calling this function.

### Parameters

**derivedBnDesc**

*Output*. Handle to a previously created tensor descriptor.

**xDesc**

*Input*. Handle to a previously created and initialized layer's **x** data descriptor.

**mode**

*Input*. Batch normalization layer mode of operation.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_BAD_PARAM**

Invalid Batch Normalization mode.

## 4.39. cudnnDestroy

```
cudnnStatus_t cudnnDestroy(cudnnHandle_t handle)
```

This function releases the resources used by the cuDNN handle. This function is usually the last call with a particular handle to the cuDNN handle. Because **cudnnCreate** allocates some internal resources, the release of those resources by calling **cudnnDestroy** will implicitly call **cudaDeviceSynchronize**; therefore, the recommended best practice is to call **cudnnCreate/cudnnDestroy** outside of performance-critical code paths.

### Parameters

**`handle`**

*Input*. Pointer to the cuDNN handle to be destroyed.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The cuDNN context destruction was successful.

**`CUDNN_STATUS_BAD_PARAM`**

Invalid (**`NULL`**) pointer supplied.

## 4.40. cudnnDestroyActivationDescriptor

```
cudnnStatus_t cudnnDestroyActivationDescriptor(
        cudnnActivationDescriptor_t activationDesc)
```

This function destroys a previously created activation descriptor object.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The object was destroyed successfully.

## 4.41. cudnnDestroyAlgorithmDescriptor

```
cudnnStatus_t cudnnDestroyAlgorithmDescriptor(
        cudnnActivationDescriptor_t algorithmDesc)
```

This function destroys a previously created algorithm descriptor object.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The object was destroyed successfully.

## 4.42. cudnnDestroyAlgorithmPerformance

```
cudnnStatus_t cudnnDestroyAlgorithmPerformance(
        cudnnAlgorithmPerformance_t     algoPerf)
```

This function destroys a previously created algorithm descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.43. cudnnDestroyAttnDescriptor

```
cudnnStatus_t cudnnDestroyAttnDescriptor(cudnnAttnDescriptor_t attnDesc);
```

This function destroys the attention descriptor object and releases its memory. The **attnDesc** argument can be **NULL**. Invoking **cudnnDestroyAttnDescriptor()** with a **NULL** argument is a no operation (NOP).

The **cudnnDestroyAttnDescriptor()** function is not able to detect if the **attnDesc** argument holds a valid address. Undefined behavior will occur in case of passing an invalid pointer, not returned by the **cudnnCreateAttnDescriptor()** function, or in the double deletion scenario of a valid address.

### Parameters

**attnDesc**

*Input.* Pointer to the attention descriptor object to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor was destroyed successfully.

## 4.44. cudnnDestroyConvolutionDescriptor

```
cudnnStatus_t cudnnDestroyConvolutionDescriptor(
    cudnnConvolutionDescriptor_t convDesc)
```

This function destroys a previously created convolution descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor was destroyed successfully.

## 4.45. cudnnDestroyCTCLossDescriptor

```
cudnnStatus_t cudnnDestroyCTCLossDescriptor(
    cudnnCTCLossDescriptor_t  ctcLossDesc)
```

This function destroys a CTC loss function descriptor object.

### Parameters

**ctcLossDesc**

*Input*. CTC loss function descriptor to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

## 4.46. cudnnDestroyDropoutDescriptor

```
cudnnStatus_t cudnnDestroyDropoutDescriptor(
    cudnnDropoutDescriptor_t dropoutDesc)
```

This function destroys a previously created dropout descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.47. cudnnDestroyFilterDescriptor

```
cudnnStatus_t cudnnDestroyFilterDescriptor(
    cudnnFilterDescriptor_t filterDesc)
```

This function destroys a previously created tensor 4D descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.48. cudnnDestroyFusedOpsConstParamPack

```
cudnnStatus_t cudnnDestroyFusedOpsConstParamPack(
 cudnnFusedOpsConstParamPack_t constPack);
```

This function destroys a previously-created cudnnFusedOpsConstParamPack_t structure.

### Parameters

**constPack**

*Input*. The **cudnnFusedOpsConstParamPack_t** structure that should be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**
If the descriptor is destroyed successfully.

**CUDNN_STATUS_INTERNAL_ERROR**
If the ops enum value is not supported or invalid.

## 4.49. cudnnDestroyFusedOpsPlan

```
cudnnStatus_t cudnnDestroyFusedOpsPlan(
 cudnnFusedOpsPlan_t plan);
```

This function destroys the plan descriptor provided.

### Parameters

**plan**
*Input*. The descriptor that should be destroyed by this function.

### Returns

**CUDNN_STATUS_SUCCESS**
If either the plan descriptor is **NULL** or the descriptor is successfully destroyed.

## 4.50. cudnnDestroyFusedOpsVariantParamPack

```
cudnnStatus_t cudnnDestroyFusedOpsVariantParamPack(
 cudnnFusedOpsVariantParamPack_t varPack);
```

This function destroys a previously-created descriptor for **cudnnFusedOps** constant parameters.

### Parameters

**varPack**
*Input*. The descriptor that should be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**
The descriptor is successfully destroyed.

## 4.51. cudnnDestroyLRNDescriptor

```
cudnnStatus_t cudnnDestroyLRNDescriptor(
    cudnnLRNDescriptor_t lrnDesc)
```

This function destroys a previously created LRN descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.52. cudnnDestroyOpTensorDescriptor

```
cudnnStatus_t cudnnDestroyOpTensorDescriptor(
    cudnnOpTensorDescriptor_t   opTensorDesc)
```

This function deletes a tensor pointwise math descriptor object.

### Parameters

**opTensorDesc**

*Input*. Pointer to the structure holding the description of the tensor pointwise math to be deleted.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

## 4.53. cudnnDestroyPersistentRNNPlan

```
cudnnStatus_t cudnnDestroyPersistentRNNPlan(
    cudnnPersistentRNNPlan_t plan)
```

This function destroys a previously created persistent RNN plan object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.54. cudnnDestroyPoolingDescriptor

```
cudnnStatus_t cudnnDestroyPoolingDescriptor(
    cudnnPoolingDescriptor_t poolingDesc)
```

This function destroys a previously created pooling descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

# 4.55. cudnnDestroyReduceTensorDescriptor

```
cudnnStatus_t cudnnDestroyReduceTensorDescriptor(
    cudnnReduceTensorDescriptor_t   tensorDesc)
```

This function destroys a previously created reduce tensor descriptor object. When the input pointer is **NULL**, this function performs no destroy operation.

### Parameters

**tensorDesc**

*Input*. Pointer to the reduce tensor descriptor object to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

# 4.56. cudnnDestroyRNNDataDescriptor

```
cudnnStatus_t cudnnDestroyRNNDataDescriptor(
    cudnnRNNDataDescriptor_t RNNDataDesc)
```

This function destroys a previously created RNN data descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The RNN data descriptor object was destroyed successfully.

# 4.57. cudnnDestroyRNNDescriptor

```
cudnnStatus_t cudnnDestroyRNNDescriptor(
    cudnnRNNDescriptor_t rnnDesc)
```

This function destroys a previously created RNN descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

# 4.58. cudnnDestroySeqDataDescriptor

```
cudnnStatus_t cudnnDestroySeqDataDescriptor(cudnnSeqDataDescriptor_t
 seqDataDesc);
```

This function destroys the sequence data descriptor object and releases its memory. The **seqDataDesc** argument can be **NULL**. Invoking **cudnnDestroySeqDataDescriptor()** with a **NULL** argument is a no operation (NOP).

The **cudnnDestroySeqDataDescriptor()** function is not able to detect if the **seqDataDesc** argument holds a valid address. Undefined behavior will occur in case of passing an invalid pointer, not returned by the **cudnnCreateSeqDataDescriptor()** function, or in the double deletion scenario of a valid address.

### Parameters

**seqDataDesc**

*Input*. Pointer to the sequence data descriptor object to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor was destroyed successfully.

# 4.59. cudnnDestroySpatialTransformerDescriptor

```
cudnnStatus_t cudnnDestroySpatialTransformerDescriptor(
    cudnnSpatialTransformerDescriptor_t stDesc)
```

This function destroys a previously created spatial transformer descriptor object.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

# 4.60. cudnnDestroyTensorDescriptor

```
cudnnStatus_t cudnnDestroyTensorDescriptor(cudnnTensorDescriptor_t tensorDesc)
```

This function destroys a previously created tensor descriptor object. When the input pointer is **NULL**, this function performs no destroy operation.

### Parameters

**tensorDesc**

*Input*. Pointer to the tensor descriptor object to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

## 4.61. cudnnDestroyTensorTransformDescriptor

```
cudnnStatus_t cudnnDestroyTensorTransformDescriptor(
 cudnnTensorTransformDescriptor_t transformDesc);
```

Destroys a previously created tensor transform descriptor.

### Parameters

**transformDesc**

*Input*. The tensor transform descriptor to be destroyed.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor was destroyed successfully.

## 4.62. cudnnDivisiveNormalizationBackward

```
cudnnStatus_t cudnnDivisiveNormalizationBackward(
    cudnnHandle_t                    handle,
    cudnnLRNDescriptor_t             normDesc,
    cudnnDivNormMode_t               mode,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *means,
    const void                      *dy,
    void                            *temp,
    void                            *temp2,
    const void                      *beta,
    const cudnnTensorDescriptor_t    dxDesc,
    void                            *dx,
    void                            *dMeans)
```

This function performs the backward **DivisiveNormalization** layer computation.

Supported tensor formats are NCHW for 4D and NCDHW for 5D with any non-overlapping non-negative strides. Only 4D and 5D tensors are supported.

### Parameters

**`handle`**

*Input*. Handle to a previously created cuDNN library descriptor.

**`normDesc`**

*Input*. Handle to a previously initialized LRN parameter descriptor (this descriptor is used for both LRN and **`DivisiveNormalization`** layers).

**`mode`**

*Input*. **`DivisiveNormalization`** layer mode of operation. Currently only **`CUDNN_DIVNORM_PRECOMPUTED_MEANS`** is implemented. Normalization is performed using the means input tensor that is expected to be precomputed by the user.

**`alpha, beta`**

*Input*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**`xDesc, x, means`**

*Input*. Tensor descriptor and pointers in device memory for the layer's x and means data. Note that the **`means`** tensor is expected to be precomputed by the user. It can also contain any valid values (not required to be actual **`means`**, and can be for instance a result of a convolution with a Gaussian kernel).

**`dy`**

*Input*. Tensor pointer in device memory for the layer's **`dy`** cumulative loss differential data (error backpropagation).

**`temp, temp2`**

*Workspace*. Temporary tensors in device memory. These are used for computing intermediate values during the backward pass. These tensors do not have to be preserved from forward to backward pass. Both use **`xDesc`** as a descriptor.

**`dxDesc`**

*Input*. Tensor descriptor for **`dx`** and **`dMeans`**.

**`dx, dMeans`**

*Output*. Tensor pointers (in device memory) for the layers resulting cumulative gradients **`dx`** and **`dMeans`** (**`dLoss/dx`** and **`dLoss/dMeans`**). Both share the same descriptor.

**Returns**

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the tensor pointers **x, dx, temp, tmep2, dy** is **NULL**.
- Number of any of the input or output tensor dimensions is not within the [4,5] range.
- Either alpha or beta pointer is **NULL**.
- A mismatch in dimensions between **xDesc** and **dxDesc**.
- LRN descriptor parameters are outside of their valid ranges.
- Any of the tensor strides is negative.

`CUDNN_STATUS_UNSUPPORTED`

The function does not support the provided configuration, for example, any of the input and output tensor strides mismatch (for the same dimension) is a non-supported configuration.

# 4.63. cudnnDivisiveNormalizationForward

```
cudnnStatus_t cudnnDivisiveNormalizationForward(
    cudnnHandle_t                    handle,
    cudnnLRNDescriptor_t             normDesc,
    cudnnDivNormMode_t               mode,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *means,
    void                            *temp,
    void                            *temp2,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This function performs the forward spatial **DivisiveNormalization** layer computation. It divides every value in a layer by the standard deviation of its spatial neighbors as described in *What is the Best Multi-Stage Architecture for Object Recognition*, Jarrett 2009, Local Contrast Normalization Layer section. Note that **DivisiveNormalization** only implements the **x/max(c, sigma_x)** portion of the computation, where **sigma_x** is the variance over the spatial neighborhood of **x**. The full LCN (Local Contrastive Normalization) computation can be implemented as a two-step process:

```
x_m = x-mean(x);
y = x_m/max(c, sigma(x_m));
```

The **x-mean(x)** which is often referred to as "subtractive normalization" portion of the computation can be implemented using cuDNN average pooling layer followed by a call to **addTensor**.

Supported tensor formats are NCHW for 4D and NCDHW for 5D with any non-overlapping non-negative strides. Only 4D and 5D tensors are supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**normDesc**

*Input*. Handle to a previously initialized LRN parameter descriptor. This descriptor is used for both LRN and **DivisiveNormalization** layers.

**divNormMode**

*Input*. **DivisiveNormalization** layer mode of operation. Currently only **CUDNN_DIVNORM_PRECOMPUTED_MEANS** is implemented. Normalization is performed using the means input tensor that is expected to be precomputed by the user.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc, yDesc**

*Input*. Tensor descriptor objects for the input and output tensors. Note that **xDesc** is shared between **x**, **means**, **temp**, and **temp2** tensors.

**x**

*Input*. Input tensor data pointer in device memory.

**means**

*Input*. Input means tensor data pointer in device memory. Note that this tensor can be **NULL** (in that case its values are assumed to be zero during the computation). This tensor also doesn't have to contain **means**, these can be any values, a frequently used variation is a result of convolution with a normalized positive kernel (such as Gaussian).

**temp, temp2**

*Workspace*. Temporary tensors in device memory. These are used for computing intermediate values during the forward pass. These tensors do not have to be preserved as inputs from forward to the backward pass. Both use **xDesc** as their descriptor.

**y**

*Output*. Pointer in device memory to a tensor for the result of the forward **DivisiveNormalization** computation.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the tensor pointers **x, y, temp, temp2** is **NULL**.
- Number of input tensor or output tensor dimensions is outside of [4,5] range.
- A mismatch in dimensions between any two of the input or output tensors.
- For in-place computation when pointers **x == y**, a mismatch in strides between the input data and output data tensors.
- Alpha or beta pointer is **NULL**.
- LRN descriptor parameters are outside of their valid ranges.
- Any of the tensor strides are negative.

**CUDNN_STATUS_UNSUPPORTED**

The function does not support the provided configuration, for example, any of the input and output tensor strides mismatch (for the same dimension) is a non-supported configuration.

## 4.64. cudnnDropoutBackward

```
cudnnStatus_t cudnnDropoutBackward(
    cudnnHandle_t                   handle,
    const cudnnDropoutDescriptor_t  dropoutDesc,
    const cudnnTensorDescriptor_t   dydesc,
    const void                     *dy,
    const cudnnTensorDescriptor_t   dxdesc,
    void                           *dx,
    void                           *reserveSpace,
    size_t                          reserveSpaceSizeInBytes)
```

This function performs backward dropout operation over **dy** returning results in **dx**. If during forward dropout operation value from **x** was propagated to **y** then during backward operation value from **dy** will be propagated to **dx**, otherwise, **dx** value will be set to **0**.

Better performance is obtained for fully packed tensors.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**dropoutDesc**

*Input*. Previously created dropout descriptor object.

**dyDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**dy**

*Input*. Pointer to data of the tensor described by the **dyDesc** descriptor.

**dxDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**dx**

*Output*. Pointer to data of the tensor described by the **dxDesc** descriptor.

**reserveSpace**

*Input*. Pointer to user-allocated GPU memory used by this function. It is expected that **reserveSpace** was populated during a call to **cudnnDropoutForward** and has not been changed.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided memory for the reserve space

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The number of elements of input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differs.
- The strides of the input tensor and output tensors differ and in-place operation is used (i.e., **x** and **y** pointers are equal).
- The provided **reserveSpaceSizeInBytes** is less then the value returned by **cudnnDropoutGetReserveSpaceSize**.
- **cudnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-**NULL** **states** argument.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.65. cudnnDropoutForward

```
cudnnStatus_t cudnnDropoutForward(
    cudnnHandle_t                      handle,
    const cudnnDropoutDescriptor_t     dropoutDesc,
    const cudnnTensorDescriptor_t      xdesc,
    const void                        *x,
    const cudnnTensorDescriptor_t      ydesc,
    void                              *y,
    void                              *reserveSpace,
    size_t                             reserveSpaceSizeInBytes)
```

This function performs forward dropout operation over **x** returning results in **y**. If **dropout** was used as a parameter to **cudnnSetDropoutDescriptor**, the approximately **dropout** fraction of **x** values will be replaced by a **0**, and the rest will be scaled by **1/(1-dropout)**. This function should not be running concurrently with another **cudnnDropoutForward** function using the same **states**.

- Better performance is obtained for fully packed tensors.
- This function should not be called during inference.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**dropoutDesc**

*Input*. Previously created dropout descriptor object.

**xDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**x**

*Input*. Pointer to data of the tensor described by the **xDesc** descriptor.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**y**

*Output*. Pointer to data of the tensor described by the **yDesc** descriptor.

**reserveSpace**

*Output*. Pointer to user-allocated GPU memory used by this function. It is expected that the contents of **reserveSpace** does not change between **cudnnDropoutForward** and **cudnnDropoutBackward** calls.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided memory for the reserve space.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The number of elements of input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differs.
- The strides of the input tensor and output tensors differ and in-place operation is used (i.e., **x** and **y** pointers are equal).
- The provided **reserveSpaceSizeInBytes** is less then the value returned by **cudnnDropoutGetReserveSpaceSize**.
- **cudnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-**NULL** **states** argument.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.66. cudnnDropoutGetReserveSpaceSize

```
cudnnStatus_t cudnnDropoutGetReserveSpaceSize(
    cudnnTensorDescriptor_t     xDesc,
    size_t                     *sizeInBytes)
```

This function is used to query the amount of reserve needed to run dropout with the input dimensions given by **xDesc**. The same reserve space is expected to be passed to **cudnnDropoutForward** and **cudnnDropoutBackward**, and its contents is expected to remain unchanged between **cudnnDropoutForward** and **cudnnDropoutBackward** calls.

### Parameters

**xDesc**

*Input*. Handle to a previously initialized tensor descriptor, describing input to a dropout operation.

**sizeInBytes**

*Output*. Amount of GPU memory needed as reserve space to be able to run dropout with an input tensor descriptor specified by **xDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

## 4.67. cudnnDropoutGetStatesSize

```
cudnnStatus_t cudnnDropoutGetStatesSize(
    cudnnHandle_t       handle,
    size_t             *sizeInBytes)
```

This function is used to query the amount of space required to store the states of the random number generators used by **cudnnDropoutForward** function.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**sizeInBytes**

*Output*. Amount of GPU memory needed to store random generator states.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

## 4.68. cudnnFindConvolutionBackwardDataAlgorithm

```
cudnnStatus_t cudnnFindConvolutionBackwardDataAlgorithm(
    cudnnHandle_t                         handle,
    const cudnnFilterDescriptor_t         wDesc,
    const cudnnTensorDescriptor_t         dyDesc,
    const cudnnConvolutionDescriptor_t    convDesc,
    const cudnnTensorDescriptor_t         dxDesc,
    const int                             requestedAlgoCount,
    int                                  *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t    *perfResults)
```

This function attempts all algorithms available for cudnnConvolutionBackwardData. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionBwdDataAlgoPerf_t. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionBackwardDataAlgorithmMaxCount.

- This function is host blocking.
- It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **wDesc**, **dyDesc**, or **dxDesc** is not allocated properly.
- **wDesc**, **dyDesc**, or **dxDesc** has fewer than 1 dimension.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

**CUDNN_STATUS_ALLOC_FAILED**

This function was unable to allocate memory to store sample input, filters and output.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

# 4.69. cudnnFindConvolutionBackwardDataAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionBackwardDataAlgorithmEx(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const void                            *w,
    const cudnnTensorDescriptor_t          dyDesc,
    const void                            *dy,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    void                                  *dx,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t     *perfResults,
    void                                  *workSpace,
    size_t                                 workSpaceSizeInBytes)
```

This function attempts all algorithms available for cudnnConvolutionBackwardData. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionBwdDataAlgoPerf_t. These metrics

are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionBackwardDataAlgorithmMaxCount.

This function is host blocking.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**dy**

*Input*. Data pointer to GPU memory associated with the filter descriptor **dyDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**dxDesc**

*Input/Output*. Data pointer to GPU memory associated with the tensor descriptor **dxDesc**. The content of this tensor will be overwritten with arbitrary values.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**workSpace**

*Input*. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a **workSpace** of 0 bytes.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **wDesc**, **dyDesc**, or **dxDesc** is not allocated properly.
- **wDesc**, **dyDesc**, or **dxDesc** has fewer than 1 dimension.
- **w**, **dy**, or **dx** is nil.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

## 4.70. cudnnFindConvolutionBackwardFilterAlgorithm

```
cudnnStatus_t cudnnFindConvolutionBackwardFilterAlgorithm(
cudnnHandle_t                        handle,
const cudnnTensorDescriptor_t        xDesc,
const cudnnTensorDescriptor_t        dyDesc,
const cudnnConvolutionDescriptor_t   convDesc,
const cudnnFilterDescriptor_t        dwDesc,
const int                            requestedAlgoCount,
int                                 *returnedAlgoCount,
cudnnConvolutionBwdFilterAlgoPerf_t *perfResults)
```

This function attempts all algorithms available for cudnnConvolutionBackwardFilter. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionBwdFilterAlgoPerf_t. These metrics are written in a sorted fashion where the first element has the lowest compute

time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionBackwardFilterAlgorithmMaxCount.

- This function is host blocking.
- It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **xDesc**, **dyDesc**, or **dwDesc** is not allocated properly.
- **xDesc**, **dyDesc**, or **dwDesc** has fewer than 1 dimension.
- Either **returnedCount** or **perfResults** is nil.

- **requestedCount** is less than 1.

**CUDNN_STATUS_ALLOC_FAILED**

This function was unable to allocate memory to store sample input, filters and output.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

# 4.71. cudnnFindConvolutionBackwardFilterAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionBackwardFilterAlgorithmEx(
    cudnnHandle_t                          handle,
    const cudnnTensorDescriptor_t          xDesc,
    const void                            *x,
    const cudnnTensorDescriptor_t          dyDesc,
    const void                            *dy,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnFilterDescriptor_t          dwDesc,
    void                                  *dw,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionBwdFilterAlgoPerf_t   *perfResults,
    void                                  *workSpace,
    size_t                                 workSpaceSizeInBytes)
```

This function attempts all algorithms available for cudnnConvolutionBackwardFilter. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionBwdFilterAlgoPerf_t. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionBackwardFilterAlgorithmMaxCount.

This function is host blocking.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**x**

*Input*. Data pointer to GPU memory associated with the filter descriptor **xDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dyDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor.

**dw**

*Input/Output*. Data pointer to GPU memory associated with the filter descriptor **dwDesc**. The content of this tensor will be overwritten with arbitrary values.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**workSpace**

*Input*. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a **workSpace** of 0 bytes.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **xDesc**, **dyDesc**, or **dwDesc** is not allocated properly.
- **xDesc**, **dyDesc**, or **dwDesc** has fewer than 1 dimension.
- **x**, **dy**, or **dw** is nil.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

# 4.72. cudnnFindConvolutionForwardAlgorithm

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithm(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults)
```

This function attempts all algorithms available for cudnnConvolutionForward. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionFwdAlgoPerf_t. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionForwardAlgorithmMaxCount.

- This function is host blocking.
- It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **xDesc**, **wDesc**, or **yDesc** is not allocated properly.
- **xDesc**, **wDesc**, or **yDesc** has fewer than 1 dimension.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

**CUDNN_STATUS_ALLOC_FAILED**

This function was unable to allocate memory to store sample input, filters and output.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

# 4.73. cudnnFindConvolutionForwardAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithmEx(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const cudnnFilterDescriptor_t      wDesc,
    const void                        *w,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    void                              *y,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults,
    void                              *workSpace,
    size_t                             workSpaceSizeInBytes)
```

This function attempts all algorithms available for cudnnConvolutionForward. It will attempt both the provided **convDesc mathType** and **CUDNN_DEFAULT_MATH** (assuming the two differ).

Algorithms without the **CUDNN_TENSOR_OP_MATH** availability will only be tried with **CUDNN_DEFAULT_MATH**, and returned as such.

Memory is allocated via **cudaMalloc()**. The performance metrics are returned in the user-allocated array of cudnnConvolutionFwdAlgoPerf_t. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API cudnnGetConvolutionForwardAlgorithmMaxCount.

This function is host blocking.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**y**

*Input/Output*. Data pointer to GPU memory associated with the tensor descriptor **yDesc**. The content of this tensor will be overwritten with arbitrary values.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**workSpace**

*Input*. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a **workSpace** of 0 bytes.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **xDesc**, **wDesc**, or **yDesc** is not allocated properly.
- **xDesc**, **wDesc**, or **yDesc** has fewer than 1 dimension.
- **x**, **w**, or **y** is nil.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

**CUDNN_STATUS_INTERNAL_ERROR**

At least one of the following conditions are met:

- The function was unable to allocate necessary timing objects.
- The function was unable to deallocate necessary timing objects.
- The function was unable to deallocate sample input, filters and output.

## 4.74. cudnnFindRNNBackwardDataAlgorithmEx

```
cudnnStatus_t cudnnFindRNNBackwardDataAlgorithmEx(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const int                          seqLength,
    const cudnnTensorDescriptor_t      *yDesc,
    const void                         *y,
    const cudnnTensorDescriptor_t      *dyDesc,
    const void                         *dy,
    const cudnnTensorDescriptor_t      dhyDesc,
    const void                         *dhy,
    const cudnnTensorDescriptor_t      dcyDesc,
    const void                         *dcy,
    const cudnnFilterDescriptor_t      wDesc,
    const void                         *w,
    const cudnnTensorDescriptor_t      hxDesc,
    const void                         *hx,
    const cudnnTensorDescriptor_t      cxDesc,
    const void                         *cx,
    const cudnnTensorDescriptor_t      *dxDesc,
    void                               *dx,
    const cudnnTensorDescriptor_t      dhxDesc,
    void                               *dhx,
    const cudnnTensorDescriptor_t      dcxDesc,
    void                               *dcx,
    const float                        findIntensity,
    const int                          requestedAlgoCount,
    int                                *returnedAlgoCount,
    cudnnAlgorithmPerformance_t        *perfResults,
    void                               *workspace,
    size_t                             workSpaceSizeInBytes,
    const void                         *reserveSpace,
    size_t                             reserveSpaceSizeInBytes)
```

This function attempts all available cuDNN algorithms for **cudnnRNNBackwardData**, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of **cudnnAlgorithmPerformance_t**. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

**y**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**dyDesc**

*Input*. An array of fully packed tensor descriptors describing the gradient at the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the second dimension of the tensor **n** in **dxDesc**.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **dyDesc**.

**dhyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dhy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dhyDesc**. If a **NULL** pointer is passed, the gradients at the final hidden state of the network will be initialized to zero.

**dcyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dcy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dcyDesc**. If a **NULL** pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**dxDesc**

*Input*. An array of fully packed tensor descriptors describing the gradient at the input of each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**dx**

*Output*. Data pointer to GPU memory associated with the tensor descriptors in the array **dxDesc**.

**dhxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dhx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dhxDesc**. If a **NULL** pointer is passed, the gradient at the hidden input of the network will not be set.

**dcxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **dxDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dcx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dcxDesc**. If a **NULL** pointer is passed, the gradient at the cell input of the network will not be set.

**findIntensity**

*Input*.This input was previously unused in versions prior to 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- Setting **findIntensity** within the range (0,1.] will set a percentage of the entire RNN search space to search. When **findIntensity** is set to 1.0, a full search is performed over all RNN parameters.
- When **findIntensity** is set to 0.0f, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- Setting **findIntensity** within the range [-1.,0) sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- Values outside the range [-1,1] are truncated to the range [-1,1], and then interpreted as per the above.
- Setting **findIntensity** to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

**Returns**

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **dhxDesc, wDesc, hxDesc, cxDesc, dcxDesc, dhyDesc, dcyDesc** or one of the descriptors in **yDesc, dxdesc, dydesc** is invalid.
- The descriptors in one of **yDesc, dxDesc, dyDesc, dhxDesc, wDesc, hxDesc, cxDesc, dcxDesc, dhyDesc, dcyDesc** has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.75. cudnnFindRNNBackwardWeightsAlgorithmEx

```
cudnnStatus_t cudnnFindRNNBackwardWeightsAlgorithmEx(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t    *xDesc,
    const void                       *x,
    const cudnnTensorDescriptor_t    hxDesc,
    const void                       *hx,
    const cudnnTensorDescriptor_t    *yDesc,
    const void                       *y,
    const float                      findIntensity,
    const int                        requestedAlgoCount,
    int                              *returnedAlgoCount,
    cudnnAlgorithmPerformance_t      *perfResults,
    const void                       *workspace,
    size_t                           workSpaceSizeInBytes,
    const cudnnFilterDescriptor_t    dwDesc,
    void                             *dw,
    const void                       *reserveSpace,
    size_t                           reserveSpaceSizeInBytes)
```

This function attempts all available cuDNN algorithms for **cudnnRNNBackwardWeights**, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of **cudnnAlgorithmPerformance_t**. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

**y**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**findIntensity**

*Input*.This input was previously unused in versions prior to 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- Setting **findIntensity** within the range (0,1.] will set a percentage of the entire RNN search space to search. When **findIntensity** is set to 1.0, a full search is performed over all RNN parameters.
- When **findIntensity** is set to 0.0f, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- Setting **findIntensity** within the range [-1.,0) sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- Values outside the range [-1,1] are truncated to the range [-1,1], and then interpreted as per the above.

- Setting **`findIntensity`** to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**`requestedAlgoCount`**

*Input*. The maximum number of elements to be stored in **`perfResults`**.

**`returnedAlgoCount`**

*Output*. The number of output elements stored in **`perfResults`**.

**`perfResults`**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**`workspace`**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**`workSpaceSizeInBytes`**

*Input*. Specifies the size in bytes of the provided **`workspace`**.

**`dwDesc`**

*Input*. Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**`dw`**

*Input/Output*. Data pointer to GPU memory associated with the filter descriptor **`dwDesc`**.

**`reserveSpace`**

*Input*. Data pointer to GPU memory to be used as a reserve space for this call.

**`reserveSpaceSizeInBytes`**

*Input*. Specifies the size in bytes of the provided **`reserveSpace`**.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The function launched successfully.

**`CUDNN_STATUS_NOT_SUPPORTED`**

The function does not support the provided configuration.

**`CUDNN_STATUS_BAD_PARAM`**

At least one of the following conditions are met:

- The descriptor **`rnnDesc`** is invalid.

- At least one of the descriptors **hxDesc, dwDesc** or one of the descriptors in **xDesc, yDesc** is invalid.
- The descriptors in one of **xDesc, hxDesc, yDesc, dwDesc** has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.76. cudnnFindRNNForwardInferenceAlgorithmEx

```
cudnnStatus_t cudnnFindRNNForwardInferenceAlgorithmEx(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t   *xDesc,
    const void                      *x,
    const cudnnTensorDescriptor_t    hxDesc,
    const void                      *hx,
    const cudnnTensorDescriptor_t    cxDesc,
    const void                      *cx,
    const cudnnFilterDescriptor_t    wDesc,
    const void                      *w,
    const cudnnTensorDescriptor_t    *yDesc,
    void                            *y,
    const cudnnTensorDescriptor_t    hyDesc,
    void                            *hy,
    const cudnnTensorDescriptor_t    cyDesc,
    void                            *cy,
    const float                     findIntensity,
    const int                       requestedAlgoCount,
    int                             *returnedAlgoCount,
    cudnnAlgorithmPerformance_t     *perfResults,
    void                            *workspace,
    size_t                           workSpaceSizeInBytes)
```

This function attempts all available cuDNN algorithms for **cudnnRNNForwardInference**, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of **cudnnAlgorithmPerformance_t**. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**. The data are expected to be packed contiguously with the first element of iteration **n+1** following directly from the last element of iteration **n**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **xDesc**.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**. The data are expected to be packed contiguously with the first element of iteration **n+1** following directly from the last element of iteration **n**.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**findIntensity**

*Input*.This input was previously unused in versions prior to 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- Setting **findIntensity** within the range (0,1.] will set a percentage of the entire RNN search space to search. When **findIntensity** is set to 1.0, a full search is performed over all RNN parameters.
- When **findIntensity** is set to 0.0f, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- Setting **findIntensity** within the range [-1.,0) sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been

heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.

- Values outside the range [-1,1] are truncated to the range [-1,1], and then interpreted as per the above.
- Setting **`findIntensity`** to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**`requestedAlgoCount`**

*Input*. The maximum number of elements to be stored in **`perfResults`**.

**`returnedAlgoCount`**

*Output*. The number of output elements stored in **`perfResults`**.

**`perfResults`**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**`workspace`**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**`workSpaceSizeInBytes`**

*Input*. Specifies the size in bytes of the provided **`workspace`**.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The function launched successfully.

**`CUDNN_STATUS_NOT_SUPPORTED`**

The function does not support the provided configuration.

**`CUDNN_STATUS_BAD_PARAM`**

At least one of the following conditions are met:

- The descriptor **`rnnDesc`** is invalid.
- At least one of the descriptors **`hxDesc, cxDesc, wDesc, hyDesc, cyDesc`** or one of the descriptors in **`xDesc, yDesc`** is invalid.
- The descriptors in one of **`xDesc, hxDesc, cxDesc, wDesc, yDesc, hyDesc, cyDesc`** have incorrect strides or dimensions.
- **`workSpaceSizeInBytes`** is too small.

**`CUDNN_STATUS_EXECUTION_FAILED`**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

## 4.77. cudnnFindRNNForwardTrainingAlgorithmEx

```
cudnnStatus_t cudnnFindRNNForwardTrainingAlgorithmEx(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t   *xDesc,
    const void                      *x,
    const cudnnTensorDescriptor_t    hxDesc,
    const void                      *hx,
    const cudnnTensorDescriptor_t    cxDesc,
    const void                      *cx,
    const cudnnFilterDescriptor_t    wDesc,
    const void                      *w,
    const cudnnTensorDescriptor_t   *yDesc,
    void                            *y,
    const cudnnTensorDescriptor_t    hyDesc,
    void                            *hy,
    const cudnnTensorDescriptor_t    cyDesc,
    void                            *cy,
    const float                     findIntensity,
    const int                       requestedAlgoCount,
    int                             *returnedAlgoCount,
    cudnnAlgorithmPerformance_t     *perfResults,
    void                            *workspace,
    size_t                           workSpaceSizeInBytes,
    void                            *reserveSpace,
    size_t                           reserveSpaceSizeInBytes)
```

This function attempts all available cuDNN algorithms for **cudnnRNNForwardTraining**, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of **cudnnAlgorithmPerformance_t**. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**xDesc**

*Input*. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **xDesc**.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**findIntensity**

*Input*.This input was previously unused in versions prior to 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- Setting **findIntensity** within the range (0,1.] will set a percentage of the entire RNN search space to search. When **findIntensity** is set to 1.0, a full search is performed over all RNN parameters.
- When **findIntensity** is set to 0.0f, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- Setting **findIntensity** within the range [-1.,0) sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- Values outside the range [-1,1] are truncated to the range [-1,1], and then interpreted as per the above.
- Setting **findIntensity** to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **hxDesc, cxDesc, wDesc, hyDesc, cyDesc** or one of the descriptors in **xDesc, yDesc** is invalid.
- The descriptors in one of **xDesc, hxDesc, cxDesc, wDesc, yDesc, hyDesc, cyDesc** have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.78. cudnnFusedOpsExecute

```
cudnnStatus_t cudnnFusedOpsExecute(
 cudnnHandle_t handle,
 const cudnnFusedOpsPlan_t plan,
 cudnnFusedOpsVariantParamPack_t varPack);
```

This function executes the sequence of **cudnnFusedOps** operations.

### Parameters

**handle**

*Input*. Pointer to the cuDNN library context.

**plan**

*Input*. Pointer to a previously-created and initialized plan descriptor.

**varPack**

*Input*. Pointer to the descriptor to the variant parameters pack.

### Returns

**CUDNN_STATUS_BAD_PARAM**

If the type of **cudnnFusedOps_t** in the plan descriptor is unsupported.

# 4.79. cudnnGetActivationDescriptor

```
cudnnStatus_t cudnnGetActivationDescriptor(
        const cudnnActivationDescriptor_t   activationDesc,
        cudnnActivationMode_t              *mode,
        cudnnNanPropagation_t              *reluNanOpt,
        double                             *coef)
```

This function queries a previously initialized generic activation descriptor object.

### Parameters

**activationDesc**

*Input*. Handle to a previously created activation descriptor.

**mode**

*Output*. Enumerant to specify the activation mode.

**reluNanOpt**

*Output*. Enumerant to specify the **Nan** propagation mode.

**coef**

*Output*. Floating point number to specify the clipping threshold when the activation mode is set to **CUDNN_ACTIVATION_CLIPPED_RELU** or to specify the alpha coefficient when the activation mode is set to **CUDNN_ACTIVATION_ELU**.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

## 4.80. cudnnGetAlgorithmDescriptor

```
cudnnStatus_t cudnnGetAlgorithmDescriptor(
        const cudnnAlgorithmDescriptor_t    algoDesc,
        cudnnAlgorithm_t                    *algorithm)
```

This function queries a previously initialized generic algorithm descriptor object.

### Parameters

**algorithmDesc**

*Input*. Handle to a previously created algorithm descriptor.

**algorithm**

*Input*. Struct to specify the algorithm.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

## 4.81. cudnnGetAlgorithmPerformance

```
cudnnStatus_t cudnnGetAlgorithmPerformance(
        const cudnnAlgorithmPerformance_t   algoPerf,
        cudnnAlgorithmDescriptor_t*         algoDesc,
        cudnnStatus_t*                      status,
        float*                              time,
        size_t*                             memory)
```

This function queries a previously initialized generic algorithm performance object.

### Parameters

**algoPerf**

*Input/Output*. Handle to a previously created algorithm performance object.

**algoDesc**

*Output*. The algorithm descriptor which the performance results describe.

**status**

*Output*. The cuDNN status returned from running the **algoDesc** algorithm.

**timecoef**

*Output*. The GPU time spent running the **algoDesc** algorithm.

**memory**

*Output*. The GPU memory needed to run the **algoDesc** algorithm.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

## 4.82. cudnnGetAlgorithmSpaceSize

```
cudnnStatus_t cudnnGetAlgorithmSpaceSize(
    cudnnHandle_t                handle,
    cudnnAlgorithmDescriptor_t  algoDesc,
    size_t*                      algoSpaceSizeInBytes)
```

This function queries for the amount of host memory needed to call **cudnnSaveAlgorithm**, much like the "get workspace size" functions query for the amount of device memory needed.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**algoDesc**

*Input*. A previously created algorithm descriptor.

**algoSpaceSizeInBytes**

*Output*. Amount of host memory needed as workspace to be able to save the metadata from the specified **algoDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the arguments is **NULL**.

## 4.83. cudnnGetAttnDescriptor

```
cudnnStatus_t cudnnGetAttnDescriptor(
 cudnnAttnDescriptor_t attnDesc,
```

```
cudnnAttnQueryMap_t *queryMap,
int *nHeads,
double *smScaler,
cudnnDataType_t *dataType,
cudnnDataType_t *computePrec,
cudnnMathType_t *mathType,
cudnnDropoutDescriptor_t *attnDropoutDesc,
cudnnDropoutDescriptor_t *postDropoutDesc,
int *qSize,
int *kSize,
int *vSize,
int *qProjSize,
int *kProjSize,
int *vProjSize,
int *oProjSize,
int *qoMaxSeqLength,
int *kvMaxSeqLength,
int *maxBatchSize,
int *maxBeamSize);
```

This function retrieves settings from the previously created attention descriptor. The user can assign **NULL** to any pointer except **attnDesc** when the retrieved value is not needed.

### Parameters

**attnDesc**
*Input*. Attention descriptor.

**attnMode**
*Output*. Pointer to the storage for binary attention flags.

**nHeads**
*Output*. Pointer to the storage for the number of attention heads.

**smScaler**
*Output*. Pointer to the storage for the softmax smoothing/sharpening coefficient.

**dataType**
*Output*. Data type for attention weights, sequence data inputs, and outputs.

**computePrec**
*Output*. Pointer to the storage for the compute precision.

**mathType**
*Output*. NVIDIA Tensor Core settings.

**attnDropoutDesc**
*Output*. Descriptor of the dropout operation applied to the softmax output.

**postDropoutDesc**
*Output*. Descriptor of the dropout operation applied to the multi-head attention output.

**qSize, kSize, vSize**
*Output*. **Q**, **K**, and **V** embedding vector lengths.

**qProjSize, kProjSize, vProjSize**
*Output*. **Q**, **K**, and **V** embedding vector lengths after input projections.

**oProjSize**
*Output*. Pointer to store the output vector length after projection.

**qoMaxSeqLength**

*Output*. Largest sequence length expected in sequence data descriptors related to **Q, O, dQ, dO** inputs and outputs.

**kvMaxSeqLength**

*Output*. Largest sequence length expected in sequence data descriptors related to **K, V, dK, dV** inputs and outputs.

**maxBatchSize**

*Output*. Largest batch size expected in the **cudnnSeqDataDescriptor_t** container.

**maxBeamSize**

*Output*. Largest beam size expected in the **cudnnSeqDataDescriptor_t** container.

### Returns

**CUDNN_STATUS_SUCCESS**

Requested attention descriptor fields were retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was found.

## 4.84. cudnnGetBatchNormalizationBackwardExWorkspaceSi

```
cudnnStatus_t cudnnGetBatchNormalizationBackwardExWorkspaceSize(
    cudnnHandle_t                       handle,
    cudnnBatchNormMode_t                mode,
    cudnnBatchNormOps_t                 bnOps,
    const cudnnTensorDescriptor_t       xDesc,
    const cudnnTensorDescriptor_t       yDesc,
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnTensorDescriptor_t       dzDesc,
    const cudnnTensorDescriptor_t       dxDesc,
    const cudnnTensorDescriptor_t       dBnScaleBiasDesc,
    const cudnnActivationDescriptor_t   activationDesc,
    size_t                              *sizeInBytes);
```

This function returns the amount of GPU memory workspace the user should allocate to be able to call cudnnBatchNormalizationBackwardEx function for the specified **bnOps** input setting. The workspace allocated will then be passed to the function **cudnnGetBatchNormalizationBackwardEx()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**mode**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**bnOps**

*Input*. Mode of operation for the fast NHWC kernel. For more information, see cudnnBatchNormOps_t. This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

**xDesc, yDesc, dyDesc, dzDesc, dxDesc**

Tensor descriptors and pointers in the device memory for the layer's **x** data, back propagated differential **dy** (inputs), the optional **y** input data, the optional **dz** output, and the **dx** output, which is the resulting differential with respect to **x**. For more information, see cudnnTensorDescriptor_t.

**dBnScaleBiasDesc**

*Input*. Shared tensor descriptor for the following six tensors: **bnScaleData, bnBiasData, dBnScaleData, dBnBiasData, savedMean,** and **savedInvVariance**. This is the shared tensor descriptor desc for the secondary tensor that was derived by cudnnDeriveBNTensorDescriptor. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be **float** for FP16 and FP32 input tensors, and **double** for FP64 input tensors.

**activationDesc**

*Input*. Tensor descriptor for the activation operation.

***sizeInBytes**

*Output*. Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the **cudnnGetBatchNormalizationBackwardEx()** function with the specified **bnOps** input setting.

## Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- Number of **xDesc**, **yDesc** or **dxDesc** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- **dBnScaleBiasDesc** dimensions not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Dimensions or data types mismatch for any pair of **xDesc, dyDesc, dxDesc**.

## 4.85. cudnnGetBatchNormalizationForwardTrainingExWorks

```
cudnnStatus_t cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize(
   cudnnHandle_t                          handle,
   cudnnBatchNormMode_t                   mode,
   cudnnBatchNormOps_t                    bnOps,
   const cudnnTensorDescriptor_t          xDesc,
   const cudnnTensorDescriptor_t          zDesc,
   const cudnnTensorDescriptor_t          yDesc,
   const cudnnTensorDescriptor_t          bnScaleBiasMeanVarDesc,
   const cudnnActivationDescriptor_t      activationDesc,
   size_t                                 *sizeInBytes);
```

This function returns the amount of GPU memory workspace the user should allocate to be able to call **cudnnGetBatchNormalizationForwardTrainingEx()** function for the specified **bnOps** input setting. The workspace allocated should then be passed by the user to the function **cudnnGetBatchNormalizationForwardTrainingEx()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**mode**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**bnOps**

*Input*. Mode of operation for the fast NHWC kernel. For more information, see cudnnBatchNormOps_t. This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

**xDesc, zDesc, yDesc**

Tensor descriptors and pointers in the device memory for the layer's **x** data, the optional **z** input data, and the **y** output. For more information, see cudnnTensorDescriptor_t.

**bnScaleBiasMeanVarDesc**

*Input*. Shared tensor descriptor for the following six tensors: **bnScaleData, bnBiasData, dBnScaleData, dBnBiasData, savedMean,** and **savedInvVariance**. This is the shared tensor descriptor desc for the secondary tensor that was derived by cudnnDeriveBNTensorDescriptor. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be **float** for FP16 and FP32 input tensors, and **double** for FP64 input tensors.

**activationDesc**

*Input*. Tensor descriptor for the activation operation. When the **bnOps** input is set to either **CUDNN_BATCHNORM_OPS_BN_ACTIVATION** or **CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION** then this activation is used.

***sizeInBytes**

*Output*. Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the **cudnnGetBatchNormalizationForwardTrainingEx()** function with the specified **bnOps** input setting.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- Number of **xDesc**, **yDesc** or **dxDesc** tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- **dBnScaleBiasDesc** dimensions not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- Dimensions or data types mismatch for **xDesc, yDesc**.

## 4.86. cudnnGetBatchNormalizationTrainingExReserveSpaceSize

```
cudnnStatus_t cudnnGetBatchNormalizationTrainingExReserveSpaceSize(
    cudnnHandle_t                       handle,
    cudnnBatchNormMode_t                mode,
    cudnnBatchNormOps_t                 bnOps,
    const cudnnActivationDescriptor_t   activationDesc,
    const cudnnTensorDescriptor_t       xDesc,
    size_t                              *sizeInBytes);
```

This function returns the amount of reserve GPU memory workspace the user should allocate for the batch normalization operation, for the specified **bnOps** input setting. In contrast to the **workspace**, the reserved space should be preserved between the forward and backward calls, and the data should not be altered.

### Parameters

**`handle`**

*Input*. Handle to a previously created cuDNN library descriptor. For more information, see cudnnHandle_t.

**`mode`**

*Input*. Mode of operation (spatial or per-activation). For more information, see cudnnBatchNormMode_t.

**`bnOps`**

*Input*. Mode of operation for the fast NHWC kernel. For more information, see cudnnBatchNormOps_t. This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

**`xDesc`**

Tensor descriptors for the layer's **`x`** data. For more information, see cudnnTensorDescriptor_t.

**`activationDesc`**

*Input*. Tensor descriptor for the activation operation.

**`*sizeInBytes`**

*Output*. Amount of GPU memory reserved.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The computation was performed successfully.

**`CUDNN_STATUS_NOT_SUPPORTED`**

The function does not support the provided configuration.

**`CUDNN_STATUS_BAD_PARAM`**

At least one of the following conditions are met:

- The **`xDesc`** tensor descriptor dimension is not within the [4,5] range (only 4D and 5D tensors are supported).

## 4.87. cudnnGetCallback

```
cudnnStatus_t cudnnGetCallback(
        unsigned            mask,
        void                **udata,
        cudnnCallback_t     fptr)
```

This function queries the internal states of cuDNN error reporting functionality.

### Parameters

**mask**

*Output*. Pointer to the address where the current internal error reporting message bit mask will be outputted.

**udata**

*Output*. Pointer to the address where the current internally stored **udata** address will be stored.

**fptr**

*Output*. Pointer to the address where the current internally stored **callback** function pointer will be stored. When the built-in default callback function is used, **NULL** will be outputted.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

If any of the input parameters are **NULL**.

## 4.88. cudnnGetConvolution2dDescriptor

```
cudnnStatus_t cudnnGetConvolution2dDescriptor(
    const cudnnConvolutionDescriptor_t  convDesc,
    int                                *pad_h,
    int                                *pad_w,
    int                                *u,
    int                                *v,
    int                                *dilation_h,
    int                                *dilation_w,
    cudnnConvolutionMode_t             *mode,
    cudnnDataType_t                    *computeType)
```

This function queries a previously initialized 2D convolution descriptor object.

### Parameters

**convDesc**

*Input/Output*. Handle to a previously created convolution descriptor.

**pad_h**

*Output*. Zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

**pad_w**

*Output*. Zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

**u**

*Output*. Vertical filter stride.

**v**

*Output*. Horizontal filter stride.

**dilation_h**

*Output*. Filter height dilation.

**dilation_w**

*Output*. Filter width dilation.

**mode**

*Output*. Convolution mode.

**computeType**

*Output*. Compute precision.

### Returns

**CUDNN_STATUS_SUCCESS**

The operation was successful.

**CUDNN_STATUS_BAD_PARAM**

The parameter **convDesc** is nil.

## 4.89. cudnnGetConvolution2dForwardOutputDim

```
cudnnStatus_t cudnnGetConvolution2dForwardOutputDim(
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       inputTensorDesc,
    const cudnnFilterDescriptor_t       filterDesc,
    int                                *n,
    int                                *c,
    int                                *h,
    int                                *w)
```

This function returns the dimensions of the resulting 4D tensor of a 2D convolution, given the convolution descriptor, the input tensor descriptor and the filter descriptor This function can help to setup the output tensor and allocate the proper amount of memory prior to launch the actual convolution.

Each dimension **h and w** of the output images is computed as follows:

```
    outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) )/
convolutionStride;
```

The dimensions provided by this routine must be strictly respected when calling **cudnnConvolutionForward()** or **cudnnConvolutionBackwardBias()**. Providing a smaller or larger output tensor is not supported by the convolution routines.

### Parameters

**convDesc**

*Input*. Handle to a previously created convolution descriptor.

**inputTensorDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**filterDesc**

*Input*. Handle to a previously initialized filter descriptor.

**n**

*Output*. Number of output images.

**c**

*Output*. Number of output feature maps per image.

**h**

*Output*. Height of each output feature map.

**w**

*Output*. Width of each output feature map.

### Returns

**CUDNN_STATUS_BAD_PARAM**

One or more of the descriptors has not been created correctly or there is a mismatch between the feature maps of **inputTensorDesc** and **filterDesc**.

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

## 4.90. cudnnGetConvolutionBackwardDataAlgorithm

```
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithm(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    cudnnConvolutionBwdDataPreference_t    preference,
    size_t                                 memoryLimitInBytes,
    cudnnConvolutionBwdDataAlgo_t         *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardData** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, please use **cudnnFindConvolutionBackwardDataAlgorithm**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**preference**

*Input*. Enumerant to express the preference criteria in terms of memory requirement and speed.

**memoryLimitInBytes**

*Input*. Specifies the maximum amount of GPU memory the user is willing to use as a workspace. This is currently a placeholder and is not used.

**algo**

*Output*. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.

# 4.91. cudnnGetConvolutionBackwardDataAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithm_v7(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionBwdDataAlgoPerf_t     *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardData** for the given layer specifications. This function will return all algorithms (including **CUDNN_TENSOR_OP_MATH** and **CUDNN_DEFAULT_MATH** versions of algorithms where **CUDNN_TENSOR_OP_MATH** may be available) sorted by expected (based on internal heuristic) relative performance with the fastest being index 0 of **perfResults**. For an exhaustive search for the fastest algorithm, use **cudnnFindConvolutionBackwardDataAlgorithm**. The total number of resulting algorithms can be queried through the API **cudnnGetConvolutionBackwardMaxCount()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **handle**, **wDesc**, **dyDesc**, **convDesc**, **dxDesc**, **perfResults**, **returnedAlgoCount** is **NULL**.
- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.
- **requestedAlgoCount** is less than or equal to 0.

## 4.92. cudnnGetConvolutionBackwardDataAlgorithmMaxCount

```
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithmMaxCount(
    cudnnHandle_t       handle,
    int                 *count)
```

This function returns the maximum number of algorithms which can be returned from **cudnnFindConvolutionBackwardDataAlgorithm()** and **cudnnGetConvolutionForwardAlgorithm_v7()**. This is the sum of all algorithms plus the sum of all algorithms with Tensor Core operations supported for the current device.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**count**

*Output*. The resulting maximum number of algorithms.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was successful.

**CUDNN_STATUS_BAD_PARAM**

The provided handle is not allocated properly.

## 4.93. cudnnGetConvolutionBackwardDataWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionBackwardDataWorkspaceSize(
    cudnnHandle_t                       handle,
    const cudnnFilterDescriptor_t       wDesc,
```

```
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       dxDesc,
    cudnnConvolutionBwdDataAlgo_t       algo,
    size_t                             *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **cudnnConvolutionBackwardData** with the specified algorithm. The workspace allocated will then be passed to the routine **cudnnConvolutionBackwardData**. The specified algorithm can be the result of the call to **cudnnGetConvolutionBackwardDataAlgorithm** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dxDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**algo**

*Input*. Enumerant that specifies the chosen convolution algorithm.

**sizeInBytes**

*Output*. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **algo**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.

**CUDNN_STATUS_NOT_SUPPORTED**

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

# 4.94. cudnnGetConvolutionBackwardFilterAlgorithm

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithm(
    cudnnHandle_t                          handle,
    const cudnnTensorDescriptor_t          xDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnFilterDescriptor_t          dwDesc,
    cudnnConvolutionBwdFilterPreference_t  preference,
    size_t                                 memoryLimitInBytes,
    cudnnConvolutionBwdFilterAlgo_t       *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardFilter** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, use **cudnnFindConvolutionBackwardFilterAlgorithm**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor.

**preference**

*Input*. Enumerant to express the preference criteria in terms of memory requirement and speed.

**memoryLimitInBytes**

*Input*. It is to specify the maximum amount of GPU memory the user is willing to use as a workspace. This is currently a placeholder and is not used.

**algo**

*Output*. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.

## 4.95. cudnnGetConvolutionBackwardFilterAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithm_v7(
    cudnnHandle_t                              handle,
    const cudnnTensorDescriptor_t              xDesc,
    const cudnnTensorDescriptor_t              dyDesc,
    const cudnnConvolutionDescriptor_t         convDesc,
    const cudnnFilterDescriptor_t              dwDesc,
    const int                                  requestedAlgoCount,
    int                                       *returnedAlgoCount,
    cudnnConvolutionBwdFilterAlgoPerf_t       *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardFilter** for the given layer specifications. This function will return all algorithms (including **CUDNN_TENSOR_OP_MATH** and **CUDNN_DEFAULT_MATH** versions of algorithms where **CUDNN_TENSOR_OP_MATH** may be available) sorted by expected (based on internal heuristic) relative performance with fastest being index 0 of **perfResults**. For an exhaustive search for the fastest algorithm, use **cudnnFindConvolutionBackwardFilterAlgorithm**. The total number of resulting algorithms can be queried through the API **cudnnGetConvolutionBackwardMaxCount()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **handle**, **xDesc**, **dyDesc**, **convDesc**, **dwDesc**, **perfResults**, **returnedAlgoCount** is **NULL**.
- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.
- **requestedAlgoCount** is less than or equal to 0.

## 4.96. cudnnGetConvolutionBackwardFilterAlgorithmMaxCou

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithmMaxCount(
    cudnnHandle_t       handle,
    int                 *count)
```

This function returns the maximum number of algorithms which can be returned from **cudnnFindConvolutionBackwardFilterAlgorithm()** and **cudnnGetConvolutionForwardAlgorithm_v7()**. This is the sum of all algorithms plus the sum of all algorithms with Tensor Core operations supported for the current device.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**count**

*Output*. The resulting maximum count of algorithms.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was successful.

**CUDNN_STATUS_BAD_PARAM**

The provided handle is not allocated properly.

## 4.97. cudnnGetConvolutionBackwardFilterWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterWorkspaceSize(
    cudnnHandle_t                       handle,
    const cudnnTensorDescriptor_t       xDesc,
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnFilterDescriptor_t       dwDesc,
    cudnnConvolutionBwdFilterAlgo_t     algo,
    size_t                             *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **cudnnConvolutionBackwardFilter** with the specified algorithm. The workspace allocated will then be passed to the routine **cudnnConvolutionBackwardFilter**. The specified algorithm can be the result of the call to **cudnnGetConvolutionBackwardFilterAlgorithm** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor.

**algo**

*Input*. Enumerant that specifies the chosen convolution algorithm.

**sizeInBytes**

*Output*. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **algo**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.

**CUDNN_STATUS_NOT_SUPPORTED**

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

## 4.98. cudnnGetConvolutionForwardAlgorithm

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    cudnnConvolutionFwdPreference_t    preference,
    size_t                             memoryLimitInBytes,
    cudnnConvolutionFwdAlgo_t         *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionForward** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, use **cudnnFindConvolutionForwardAlgorithm**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**wDesc**

*Input*. Handle to a previously initialized convolution filter descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**preference**

*Input*. Enumerant to express the preference criteria in terms of memory requirement and speed.

**memoryLimitInBytes**

*Input*. It is used when an enumerant **preference** is set to **CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT** to specify the maximum amount of GPU memory the user is willing to use as a workspace

**algo**

*Output*. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **handle**, **xDesc**, **wDesc**, **convDesc**, **yDesc** is **NULL**.
- Either **yDesc** or **wDesc** have different dimensions from **xDesc**.
- The data types of tensors **xDesc**, **yDesc** or **wDesc** are not all the same.
- The number of feature maps in **xDesc** and **wDesc** differs.
- The tensor **xDesc** has a dimension smaller than 3.

## 4.99. cudnnGetConvolutionForwardAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm_v7(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionForward** for the given layer specifications. This function will return all algorithms (including **CUDNN_TENSOR_OP_MATH** and **CUDNN_DEFAULT_MATH**

versions of algorithms where **CUDNN_TENSOR_OP_MATH** may be available) sorted by expected (based on internal heuristic) relative performance with the fastest being index 0 of **perfResults**. For an exhaustive search for the fastest algorithm, use **cudnnFindConvolutionForwardAlgorithm**. The total number of resulting algorithms can be queried through the API **cudnnGetConvolutionForwardMaxCount()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**wDesc**

*Input*. Handle to a previously initialized convolution filter descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**requestedAlgoCount**

*Input*. The maximum number of elements to be stored in **perfResults**.

**returnedAlgoCount**

*Output*. The number of output elements stored in **perfResults**.

**perfResults**

*Output*. A user-allocated array to store performance metrics sorted ascending by compute time.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **handle**, **xDesc**, **wDesc**, **convDesc**, **yDesc**, **perfResults**, **returnedAlgoCount** is **NULL**.
- Either **yDesc** or **wDesc** have different dimensions from **xDesc**.
- The data types of tensors **xDesc**, **yDesc** or **wDesc** are not all the same.
- The number of feature maps in **xDesc** and **wDesc** differs.
- The tensor **xDesc** has a dimension smaller than 3.

- **requestedAlgoCount** is less than or equal to 0.

# 4.100. cudnnGetConvolutionForwardAlgorithmMaxCount

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithmMaxCount(
    cudnnHandle_t   handle,
    int             *count)
```

This function returns the maximum number of algorithms which can be returned from **cudnnFindConvolutionForwardAlgorithm()** and **cudnnGetConvolutionForwardAlgorithm_v7()**. This is the sum of all algorithms plus the sum of all algorithms with Tensor Core operations supported for the current device.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**count**

*Output*. The resulting maximum number of algorithms.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was successful.

**CUDNN_STATUS_BAD_PARAM**

The provided handle is not allocated properly.

# 4.101. cudnnGetConvolutionForwardWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionForwardWorkspaceSize(
    cudnnHandle_t   handle,
    const   cudnnTensorDescriptor_t         xDesc,
    const   cudnnFilterDescriptor_t         wDesc,
    const   cudnnConvolutionDescriptor_t    convDesc,
    const   cudnnTensorDescriptor_t         yDesc,
    cudnnConvolutionFwdAlgo_t               algo,
    size_t                                 *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **cudnnConvolutionForward** with the specified algorithm. The workspace allocated will then be passed to the routine **cudnnConvolutionForward**. The specified algorithm can be the result of the call to **cudnnGetConvolutionForwardAlgorithm** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**xDesc**

*Input*. Handle to the previously initialized x tensor descriptor.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**convDesc**

*Input*. Previously initialized convolution descriptor.

**yDesc**

*Input*. Handle to the previously initialized **y** tensor descriptor.

**algo**

*Input*. Enumerant that specifies the chosen convolution algorithm.

**sizeInBytes**

*Output*. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **algo**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **handle**, **xDesc**, **wDesc**, **convDesc**, **yDesc** is **NULL**.
- The tensor **yDesc** or **wDesc** are not of the same dimension as **xDesc**.
- The tensor **xDesc**, **yDesc** or **wDesc** are not of the same data type.
- The numbers of feature maps of the tensor **xDesc** and **wDesc** differ.
- The tensor **xDesc** has a dimension smaller than 3.

**CUDNN_STATUS_NOT_SUPPORTED**

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

## 4.102. cudnnGetConvolutionGroupCount

```
cudnnStatus_t cudnnGetConvolutionGroupCount(
    cudnnConvolutionDescriptor_t    convDesc,
    int                            *groupCount)
```

This function returns the group count specified in the given convolution descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The group count was returned successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid convolution descriptor was provided.

## 4.103. cudnnGetConvolutionMathType

```
cudnnStatus_t cudnnGetConvolutionMathType(
    cudnnConvolutionDescriptor_t    convDesc,
    cudnnMathType_t                *mathType)
```

This function returns the math type specified in a given convolution descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The math type was returned successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid convolution descriptor was provided.

## 4.104. cudnnGetConvolutionNdDescriptor

```
cudnnStatus_t cudnnGetConvolutionNdDescriptor(
    const cudnnConvolutionDescriptor_t  convDesc,
    int                                 arrayLengthRequested,
    int                                *arrayLength,
    int                                 padA[],
    int                                 filterStrideA[],
    int                                 dilationA[],
    cudnnConvolutionMode_t             *mode,
    cudnnDataType_t                    *dataType)
```

This function queries a previously initialized convolution descriptor object.

### Parameters

**convDesc**

*Input/Output*. Handle to a previously created convolution descriptor.

**arrayLengthRequested**

*Input*. Dimension of the expected convolution descriptor. It is also the minimum size of the arrays **padA**, **filterStrideA**, and **dilationA** in order to be able to hold the results

**arrayLength**

*Output*. Actual dimension of the convolution descriptor.

**padA**

*Output*. Array of dimension of at least **arrayLengthRequested** that will be filled with the padding parameters from the provided convolution descriptor.

**filterStrideA**

*Output*. Array of dimension of at least **arrayLengthRequested** that will be filled with the filter stride from the provided convolution descriptor.

**dilationA**

*Output*. Array of dimension of at least **arrayLengthRequested** that will be filled with the dilation parameters from the provided convolution descriptor.

**mode**

*Output*. Convolution mode of the provided descriptor.

**datatype**

*Output*. Datatype of the provided descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **convDesc** is nil.
- The **arrayLengthRequest** is negative.

**CUDNN_STATUS_NOT_SUPPORTED**

The **arrayLengthRequested** is greater than **CUDNN_DIM_MAX-2**.

## 4.105. cudnnGetConvolutionNdForwardOutputDim

```
cudnnStatus_t cudnnGetConvolutionNdForwardOutputDim(
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       inputTensorDesc,
    const cudnnFilterDescriptor_t       filterDesc,
    int                                 nbDims,
    int                                 tensorOuputDimA[])
```

This function returns the dimensions of the resulting n-D tensor of a **nbDims-2-D** convolution, given the convolution descriptor, the input tensor descriptor and the filter descriptor This function can help to setup the output tensor and allocate the proper amount of memory prior to launch the actual convolution.

Each dimension of the **(nbDims-2)-D** images of the output tensor is computed as follows:

```
    outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) )/
convolutionStride;
```

The dimensions provided by this routine must be strictly respected when calling **cudnnConvolutionForward()** or **cudnnConvolutionBackwardBias()**. Providing a smaller or larger output tensor is not supported by the convolution routines.

### Parameters

**convDesc**

*Input*. Handle to a previously created convolution descriptor.

**inputTensorDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**filterDesc**

*Input*. Handle to a previously initialized filter descriptor.

**nbDims**

*Input*. Dimension of the output tensor

**tensorOuputDimA**

*Output*. Array of dimensions **nbDims** that contains on exit of this routine the sizes of the output tensor

### Returns

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the parameters **convDesc**, **inputTensorDesc**, and **filterDesc** is nil.
- The dimension of the filter descriptor **filterDesc** is different from the dimension of input tensor descriptor **inputTensorDesc**.
- The dimension of the convolution descriptor is different from the dimension of input tensor descriptor **inputTensorDesc-2**.
- The features map of the filter descriptor **filterDesc** is different from the one of input tensor descriptor **inputTensorDesc**.
- The size of the dilated filter **filterDesc** is larger than the padded sizes of the input tensor.
- The dimension **nbDims** of the output array is negative or greater than the dimension of input tensor descriptor **inputTensorDesc**.

**CUDNN_STATUS_SUCCESS**

The routine exits successfully.

# 4.106. cudnnGetConvolutionReorderType

```
cudnnStatus_t cudnnGetConvolutionReorderType(
 cudnnConvolutionDescriptor_t convDesc,
 cudnnReorderType_t *reorderType);
```

This function retrieves the convolution reorder type from the given convolution descriptor.

### Parameters

**convDesc**

*Input*. The convolution descriptor from which the reorder type should be retrieved.

**reorderType**

*Output*. The retrieved reorder type. For more information, see cudnnReorderType_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

One of the inputs to this function is not valid.

**CUDNN_STATUS_SUCCESS**

The reorder type is retrieved successfully.

# 4.107. cudnnGetCTCLossDescriptor

```
cudnnStatus_t cudnnGetCTCLossDescriptor(
    cudnnCTCLossDescriptor_t         ctcLossDesc,
    cudnnDataType_t*                 compType)
```

This function returns the configuration of the passed CTC loss function descriptor.

### Parameters

**ctcLossDesc**

*Input*. CTC loss function descriptor passed, from which to retrieve the configuration.

**compType**

*Output*. Compute type associated with this CTC loss function descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

Input **OpTensor** descriptor passed is invalid.

# 4.108. cudnnGetCTCLossWorkspaceSize

```
cudnnStatus_t cudnnGetCTCLossWorkspaceSize(
    cudnnHandle_t                          handle,
    const   cudnnTensorDescriptor_t        probsDesc,
    const   cudnnTensorDescriptor_t        gradientsDesc,
    const   int                           *labels,
    const   int                           *labelLengths,
    const   int                           *inputLengths,
    cudnnCTCLossAlgo_t                     algo,
    const   cudnnCTCLossDescriptor_t       ctcLossDesc,
    size_t                                *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **cudnnCTCLoss** with the specified algorithm. The workspace allocated will then be passed to the routine **cudnnCTCLoss**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**probsDesc**

*Input*. Handle to the previously initialized probabilities tensor descriptor.

**gradientsDesc**

*Input*. Handle to a previously initialized gradients tensor descriptor.

**labels**

*Input*. Pointer to a previously initialized labels list.

**labelLengths**

*Input*. Pointer to a previously initialized lengths list, to walk the above labels list.

**inputLengths**

*Input*. Pointer to a previously initialized list of the lengths of the timing steps in each batch.

**algo**

*Input*. Enumerant that specifies the chosen CTC loss algorithm

**ctcLossDesc**

*Input*. Handle to the previously initialized CTC loss descriptor.

**sizeInBytes**

*Output*. Amount of GPU memory needed as workspace to be able to execute the CTC loss computation with the specified **algo**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions of **probsDesc** do not match the dimensions of **gradientsDesc**.
- The **inputLengths** do not agree with the first dimension of **probsDesc**.
- The **workSpaceSizeInBytes** is not sufficient.
- The **labelLengths** is greater than 256.

**CUDNN_STATUS_NOT_SUPPORTED**

A compute or data type other than **FLOAT** was chosen, or an unknown algorithm type was chosen.

## 4.109. cudnnGetCudartVersion

```
size_t cudnnGetCudartVersion()
```

The same version of a given cuDNN library can be compiled against different CUDA Toolkit versions. This routine returns the CUDA Toolkit version that the currently used cuDNN library has been compiled against.

## 4.110. cudnnGetDropoutDescriptor

```
cudnnStatus_t cudnnGetDropoutDescriptor(
    cudnnDropoutDescriptor_t    dropoutDesc,
    cudnnHandle_t               handle,
    float                      *dropout,
    void                      **states,
    unsigned long long         *seed)
```

This function queries the fields of a previously initialized dropout descriptor.

### Parameters

**dropoutDesc**

*Input*. Previously initialized dropout descriptor.

**handle**

*Input*. Handle to a previously created cuDNN context.

**dropout**

*Output*. The probability with which the value from input is set to 0 during the dropout layer.

**states**

*Output*. Pointer to user-allocated GPU memory that holds random number generator states.

**seed**

*Output*. Seed used to initialize random number generator states.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

One or more of the arguments was an invalid pointer.

## 4.111. cudnnGetErrorString

```
const char * cudnnGetErrorString(cudnnStatus_t status)
```

This function converts the cuDNN status code to a **NULL** terminated (ASCIIZ) static string. For example, when the input argument is **CUDNN_STATUS_SUCCESS**, the returned string is **CUDNN_STATUS_SUCCESS**. When an invalid status value is passed to the function, the returned string is **CUDNN_UNKNOWN_STATUS**.

### Parameters

**status**

*Input*. cuDNN enumerant status code.

### Returns

Pointer to a static, **NULL** terminated string with the status name.

## 4.112. cudnnGetFilter4dDescriptor

```
cudnnStatus_t cudnnGetFilter4dDescriptor(
    const cudnnFilterDescriptor_t     filterDesc,
    cudnnDataType_t                  *dataType,
    cudnnTensorFormat_t              *format,
    int                              *k,
    int                              *c,
    int                              *h,
    int                              *w)
```

This function queries the parameters of the previously initialized filter descriptor object.

### Parameters

**filterDesc**

*Input*. Handle to a previously created filter descriptor.

**datatype**

*Output*. Data type.

**format**

*Output*. Type of format.

**k**

*Output*. Number of output feature maps.

**c**

*Output*. Number of input feature maps.

**h**

*Output*. Height of each filter.

**w**

*Output*. Width of each filter.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

## 4.113. cudnnGetFilterNdDescriptor

```
cudnnStatus_t cudnnGetFilterNdDescriptor(
    const cudnnFilterDescriptor_t   wDesc,
    int                              nbDimsRequested,
    cudnnDataType_t                 *dataType,
    cudnnTensorFormat_t             *format,
    int                             *nbDims,
    int                              filterDimA[])
```

This function queries a previously initialized filter descriptor object.

### Parameters

**wDesc**

*Input*. Handle to a previously initialized filter descriptor.

**nbDimsRequested**

*Input*. Dimension of the expected filter descriptor. It is also the minimum size of the arrays **filterDimA** in order to be able to hold the results

**datatype**

*Output*. Data type.

**format**

*Output*. Type of format.

**nbDims**

*Output*. Actual dimension of the filter.

**filterDimA**

*Output*. Array of dimension of at least **nbDimsRequested** that will be filled with the filter parameters from the provided filter descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

The parameter **nbDimsRequested** is negative.

## 4.114. cudnnGetFusedOpsConstParamPackAttribute

```
cudnnStatus_t cudnnGetFusedOpsConstParamPackAttribute(
 const cudnnFusedOpsConstParamPack_t constPack,
 cudnnFusedOpsConstParamLabel_t paramLabel,
 void *param,
 int *isNULL);
```

This function retrieves the values of the descriptor pointed to by the **param** pointer input. The type of the descriptor is indicated by the enum value of **paramLabel** input.

### Parameters

**constPack**

*Input*. The opaque cudnnFusedOpsConstParamPack_t structure that contains the various problem size information, such as the shape, layout and the type of tensors, and the descriptors for convolution and activation, for the selected sequence of **cudnnFusedOps** computations.

**paramLabel**

*Input*. Several types of descriptors can be retrieved by this getter function. The **param** input points to the descriptor itself, and this input indicates the type of the descriptor pointed to by the **param** input. The cudnnFusedOpsConstParamLabel_t enumerant

type enables the selection of the type of the descriptor. Refer to the **param** description below.

**param**

*Input*. Data pointer to the host memory associated with the descriptor that should be retrieved. The type of this descriptor depends on the value of **paramLabel**. For the given **paramLabel**, if the associated value inside the **constPack** is set to **NULL** or by default **NULL**, then cuDNN will copy the value or the opaque structure in the **constPack** to the host memory buffer pointed to by **param**. For more information, see the table in **cudnnFusedOpsConstParamLabel_t**.

**isNULL**

*Input/Output*. User must pass a pointer to an integer in the host memory in this field. If the value in the **constPack** associated with the given **paramLabel** is by default **NULL** or previously set by the user to **NULL**, then cuDNN will write a non-zero value to the location pointed by **isNULL**.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor values are retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**

If either **constPack**, **param** or **isNULL** is **NULL**; or if **paramLabel** is invalid.

## 4.115. cudnnGetFusedOpsVariantParamPackAttribute

```
cudnnStatus_t cudnnGetFusedOpsVariantParamPackAttribute(
 const cudnnFusedOpsVariantParamPack_t varPack,
 cudnnFusedOpsVariantParamLabel_t paramLabel,
 void *ptr);
```

This function retrieves the settings of the variable parameter pack descriptor.

### Parameters

**varPack**

*Input*. Pointer to the **cudnnFusedOps** variant parameter pack (**varPack**) descriptor.

**paramLabel**

*Input*. Type of the buffer pointer parameter (in the **varPack** descriptor). For more information, see cudnnFusedOpsConstParamLabel_t. The retrieved descriptor values vary according to this type.

**ptr**

*Output*. Pointer to the host or device memory where the retrieved value is written by this function. The data type of the pointer, and the host/device memory location, depend on the **paramLabel** input selection. For more information, see **cudnnFusedOpsVariantParamLabel_t**.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor values are retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**

If either **varPack** or **ptr** is **NULL**, or if **paramLabel** is set to invalid value.

# 4.116. cudnnGetLRNDescriptor

```
cudnnStatus_t cudnnGetLRNDescriptor(
    cudnnLRNDescriptor_t    normDesc,
    unsigned               *lrnN,
    double                 *lrnAlpha,
    double                 *lrnBeta,
    double                 *lrnK)
```

This function retrieves values stored in the previously initialized LRN descriptor object.

### Parameters

**normDesc**

*Output*. Handle to a previously created LRN descriptor.

**lrnN, lrnAlpha, lrnBeta, lrnK**

*Output*. Pointers to receive values of parameters stored in the descriptor object. See cudnnSetLRNDescriptor for more details. Any of these pointers can be **NULL** (no value is returned for the corresponding parameter).

### Returns

**CUDNN_STATUS_SUCCESS**

Function completed successfully.

# 4.117. cudnnGetMultiHeadAttnBuffers

```
cudnnStatus_t cudnnGetMultiHeadAttnBuffers(
 cudnnHandle_t handle,
 const cudnnAttnDescriptor_t attnDesc,
 size_t *weightSizeInBytes,
 size_t *workSpaceSizeInBytes,
 size_t *reserveSpaceSizeInBytes);
```

This function computes weight, work, and reserve space buffer sizes used by the following functions:

- **cudnnMultiHeadAttnForward()**
- **cudnnMultiHeadAttnBackwardData()**
- **cudnnMultiHeadAttnBackwardWeights()**

Assigning **`NULL`** to the **`reserveSpaceSizeInBytes`** argument indicates that the user does not plan to invoke multi-head attention gradient functions: **`cudnnMultiHeadAttnBackwardData()`** and **`cudnnMultiHeadAttnBackwardWeights()`**. This situation occurs in the inference mode.

**`NULL`** cannot be assigned to **`weightSizeInBytes`** and **`workSpaceSizeInBytes`** pointers.

The user must allocate weight, work, and reserve space buffer sizes in the GPU memory using **`cudaMalloc()`** with the reported buffer sizes. The buffers can be also carved out from a larger chunk of allocated memory but the buffer addresses must be at least 16B aligned.

The work-space buffer is used for temporary storage. Its content can be discarded or modified after all GPU kernels launched by the corresponding API complete. The reserve-space buffer is used to transfer intermediate results from **`cudnnMultiHeadAttnForward()`** to **`cudnnMultiHeadAttnBackwardData()`**, and from **`cudnnMultiHeadAttnBackwardData()`** to **`cudnnMultiHeadAttnBackwardWeights()`**. The content of the reserve-space buffer cannot be modified until all GPU kernels launched by the above three multi-head attention API functions finish.

All multi-head attention weight and bias tensors are stored in a single weight buffer. For speed optimizations, the cuDNN API may change tensor layouts and their relative locations in the weight buffer based on the provided attention parameters. Use the **`cudnnGetMultiHeadAttnWeights()`** function to obtain the start address and the shape of each weight or bias tensor.

### Parameters

**`handle`**

*Input*. The current cuDNN context handle.

**`attnDesc`**

*Input*. Pointer to a previously initialized attention descriptor.

**`weightSizeInBytes`**

*Output*. Minimum buffer size required to store all multi-head attention trainable parameters.

**`workSpaceSizeInBytes`**

*Output*. Minimum buffer size required to hold all temporary surfaces used by the forward and gradient multi-head attention API calls.

**`reserveSpaceSizeInBytes`**

*Output*. Minimum buffer size required to store all intermediate data exchanged between forward and backward (gradient) multi-head attention functions. Set this

parameter to **NULL** in the inference mode indicating that gradient API calls will not be invoked.

### Returns

**CUDNN_STATUS_SUCCESS**

The requested buffer sizes were computed successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was found.

## 4.118. cudnnGetMultiHeadAttnWeights

```
cudnnStatus_t cudnnGetMultiHeadAttnWeights(
 cudnnHandle_t handle,
 const cudnnAttnDescriptor_t attnDesc,
 cudnnMultiHeadAttnWeightKind_t wKind,
 size_t weightSizeInBytes,
 const void *w,
 cudnnTensorDescriptor_t wDesc,
 void **wAddr);
```

This function obtains the shape of the weight or bias tensor. It also retrieves the start address of tensor data located in the **weight** buffer. Use the **wKind** argument to select a particular tensor. For more information, see **cudnnMultiHeadAttnWeightKind_t** for the description of the enumerant type.

Biases are used in the input and output projections when the **CUDNN_ATTN_ENABLE_PROJ_BIASES** flag is set in the attention descriptor. See **cudnnSetAttnDescriptor()** for the description of flags to control projection biases.

When the corresponding weight or bias tensor does not exist, the function writes **NULL** to the storage location pointed by **wAddr** and returns zeros in the **wDesc** tensor descriptor. The return status of the **cudnnGetMultiHeadAttnWeights()** function is **CUDNN_STATUS_SUCCESS** in this case.

The cuDNN **multiHeadAttention** sample code demonstrates how to access multi-head attention weights. Although the buffer with weights and biases should be allocated in the GPU memory, the user can copy it to the host memory and invoke the **cudnnGetMultiHeadAttnWeights()** function with the host weights address to obtain tensor pointers in the host memory. This scheme allows the user to inspect trainable parameters directly in the CPU memory.

### Parameters

**handle**

*Input*. The current cuDNN context handle.

**attnDesc**

*Input*. A previously configured attention descriptor.

**wKind**

*Input*. Enumerant type to specify which weight or bias tensor should be retrieved.

**weightSizeInBytes**

*Input*. Buffer size that stores all multi-head attention weights and biases.

**weights**

*Input*. Pointer to the **weight** buffer in the host or device memory.

**wDesc**

*Output*. The descriptor specifying weight or bias tensor shape. For weights, the **wDesc.dimA[]** array has three elements: **[nHeads, projected size, original size]**. For biases, the **wDesc.dimA[]** array also has three elements: **[nHeads, projected size, 1]**. The **wDesc.strideA[]** array describes how tensor elements are arranged in memory.

**wAddr**

*Output*. Pointer to a location where the start address of the requested tensor should be written. When the corresponding projection is disabled, the address written to **wAddr** is **NULL**.

### Returns

**CUDNN_STATUS_SUCCESS**

The weight tensor descriptor and the address of data in the device memory were successfully retrieved.

**CUDNN_STATUS_BAD_PARAM**

An invalid or incompatible input argument was encountered. For example, **wKind** did not have a valid value or **weightSizeInBytes** was too small.

## 4.119. cudnnGetOpTensorDescriptor

```
cudnnStatus_t cudnnGetOpTensorDescriptor(
    const cudnnOpTensorDescriptor_t opTensorDesc,
    cudnnOpTensorOp_t               *opTensorOp,
    cudnnDataType_t                 *opTensorCompType,
    cudnnNanPropagation_t           *opTensorNanOpt)
```

This function returns the configuration of the passed Tensor Pointwise math descriptor.

### Parameters

**opTensorDesc**

*Input*. Tensor pointwise math descriptor passed to get the configuration from.

**opTensorOp**

*Output*. Pointer to the tensor pointwise math operation type, associated with this tensor pointwise math descriptor.

**opTensorCompType**

*Output*. Pointer to the cuDNN data-type associated with this tensor pointwise math descriptor.

**opTensorNanOpt**

*Output*. Pointer to the NAN propagation option associated with this tensor pointwise math descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

Input tensor pointwise math descriptor passed is invalid.

## 4.120. cudnnGetPooling2dDescriptor

```
cudnnStatus_t cudnnGetPooling2dDescriptor(
    const cudnnPoolingDescriptor_t      poolingDesc,
    cudnnPoolingMode_t                 *mode,
    cudnnNanPropagation_t              *maxpoolingNanOpt,
    int                                *windowHeight,
    int                                *windowWidth,
    int                                *verticalPadding,
    int                                *horizontalPadding,
    int                                *verticalStride,
    int                                *horizontalStride)
```

This function queries a previously created 2D pooling descriptor object.

### Parameters

**poolingDesc**

*Input*. Handle to a previously created pooling descriptor.

**mode**

*Output*. Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Output*. Enumerant to specify the Nan propagation mode.

**windowHeight**

*Output*. Height of the pooling window.

**windowWidth**

*Output*. Width of the pooling window.

**verticalPadding**

*Output*. Size of vertical padding.

**horizontalPadding**

*Output*. Size of horizontal padding.

**verticalStride**

*Output*. Pooling vertical stride.

**horizontalStride**

*Output*. Pooling horizontal stride.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

## 4.121. cudnnGetPooling2dForwardOutputDim

```
cudnnStatus_t cudnnGetPooling2dForwardOutputDim(
    const cudnnPoolingDescriptor_t      poolingDesc,
    const cudnnTensorDescriptor_t       inputDesc,
    int                                *outN,
    int                                *outC,
    int                                *outH,
    int                                *outW)
```

This function provides the output dimensions of a tensor after 2d pooling has been applied.

Each dimension **h and w** of the output images is computed as follows:

```
    outputDim = 1 + (inputDim + 2*padding - windowDim)/poolingStride;
```

### Parameters

**poolingDesc**

*Input*. Handle to a previously initialized pooling descriptor.

**inputDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**N**

*Output*. Number of images in the output.

**C**

*Output*. Number of channels in the output.

**H**

*Output*. Height of images in the output.

**W**

*Output*. Width of images in the output.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **poolingDesc** has not been initialized.
- **poolingDesc** or **inputDesc** has an invalid number of dimensions (2 and 4 respectively are required).

## 4.122. cudnnGetPoolingNdDescriptor

```
cudnnStatus_t cudnnGetPoolingNdDescriptor(
const cudnnPoolingDescriptor_t      poolingDesc,
int                                 nbDimsRequested,
cudnnPoolingMode_t                 *mode,
cudnnNanPropagation_t              *maxpoolingNanOpt,
int                                *nbDims,
int                                 windowDimA[],
int                                 paddingA[],
int                                 strideA[])
```

This function queries a previously initialized generic pooling descriptor object.

### Parameters

**poolingDesc**

*Input*. Handle to a previously created pooling descriptor.

**nbDimsRequested**

*Input*. Dimension of the expected pooling descriptor. It is also the minimum size of the arrays **windowDimA**, **paddingA**, and **strideA** in order to be able to hold the results.

**mode**

*Output*. Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Input*. Enumerant to specify the Nan propagation mode.

**nbDims**

*Output*. Actual dimension of the pooling descriptor.

**windowDimA**

*Output*. Array of dimension of at least **nbDimsRequested** that will be filled with the window parameters from the provided pooling descriptor.

**paddingA**

*Output*. Array of dimension of at least **nbDimsRequested** that will be filled with the padding parameters from the provided pooling descriptor.

**strideA**

*Output*. Array of dimension at least **nbDimsRequested** that will be filled with the stride parameters from the provided pooling descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The parameter **nbDimsRequested** is greater than CUDNN_DIM_MAX.

## 4.123. cudnnGetPoolingNdForwardOutputDim

```
cudnnStatus_t cudnnGetPoolingNdForwardOutputDim(
    const cudnnPoolingDescriptor_t  poolingDesc,
    const cudnnTensorDescriptor_t   inputDesc,
    int                             nbDims,
    int                             outDimA[])
```

This function provides the output dimensions of a tensor after Nd pooling has been applied.

Each dimension of the **(nbDims-2)-D** images of the output tensor is computed as follows:

```
    outputDim = 1 + (inputDim + 2*padding - windowDim)/poolingStride;
```

### Parameters

**poolingDesc**

*Input*. Handle to a previously initialized pooling descriptor.

**inputDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**nbDims**

*Input*. Number of dimensions in which pooling is to be applied.

**outDimA**

*Output*. Array of **nbDims** output dimensions.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **poolingDesc** has not been initialized.
- The value of **nbDims** is inconsistent with the dimensionality of **poolingDesc** and **inputDesc**.

## 4.124. cudnnGetProperty

```
cudnnStatus_t cudnnGetProperty(
    libraryPropertyType     type,
    int                    *value)
```

This function writes a specific part of the cuDNN library version number into the provided host storage.

### Parameters

**type**

*Input*. Enumerant type that instructs the function to report the numerical value of the cuDNN major version, minor version, or the patch level.

**value**

*Output*. Host pointer where the version information should be written.

### Returns

**CUDNN_STATUS_INVALID_VALUE**

Invalid value of the **type** argument.

**CUDNN_STATUS_SUCCESS**

Version information was stored successfully at the provided address.

# 4.125. cudnnGetReduceTensorDescriptor

```
cudnnStatus_t cudnnGetReduceTensorDescriptor(
    const cudnnReduceTensorDescriptor_t reduceTensorDesc,
    cudnnReduceTensorOp_t               *reduceTensorOp,
    cudnnDataType_t                     *reduceTensorCompType,
    cudnnNanPropagation_t               *reduceTensorNanOpt,
    cudnnReduceTensorIndices_t          *reduceTensorIndices,
    cudnnIndicesType_t                  *reduceTensorIndicesType)
```

This function queries a previously initialized reduce tensor descriptor object.

### Parameters

**reduceTensorDesc**

*Input*. Pointer to a previously initialized reduce tensor descriptor object.

**reduceTensorOp**

*Output*. Enumerant to specify the reduce tensor operation.

**reduceTensorCompType**

*Output*. Enumerant to specify the computation datatype of the reduction.

**reduceTensorNanOpt**

*Input*. Enumerant to specify the Nan propagation mode.

**reduceTensorIndices**

*Output*. Enumerant to specify the reduce tensor indices.

**reduceTensorIndicesType**

*Output*. Enumerant to specify the reduce tensor indices type.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

**CUDNN_STATUS_BAD_PARAM**

**reduceTensorDesc** is **NULL**.

# 4.126. cudnnGetReductionIndicesSize

```
cudnnStatus_t cudnnGetReductionIndicesSize(
    cudnnHandle_t                       handle,
    const cudnnReduceTensorDescriptor_t reduceDesc,
    const cudnnTensorDescriptor_t       aDesc,
    const cudnnTensorDescriptor_t       cDesc,
    size_t                              *sizeInBytes)
```

This is a helper function to return the minimum size of the index space to be passed to the reduction given the input and output tensors.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**reduceDesc**

*Input*. Pointer to a previously initialized reduce tensor descriptor object.

**aDesc**

*Input*. Pointer to the input tensor descriptor.

**cDesc**

*Input*. Pointer to the output tensor descriptor.

**sizeInBytes**

*Output*. Minimum size of the index space to be passed to the reduction.

### Returns

**CUDNN_STATUS_SUCCESS**

The index space size is returned successfully.

## 4.127. cudnnGetReductionWorkspaceSize

```
cudnnStatus_t cudnnGetReductionWorkspaceSize(
    cudnnHandle_t                       handle,
    const cudnnReduceTensorDescriptor_t reduceDesc,
    const cudnnTensorDescriptor_t       aDesc,
    const cudnnTensorDescriptor_t       cDesc,
    size_t                              *sizeInBytes)
```

This is a helper function to return the minimum size of the workspace to be passed to the reduction given the input and output tensors.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**reduceDesc**

*Input*. Pointer to a previously initialized reduce tensor descriptor object.

**aDesc**

*Input*. Pointer to the input tensor descriptor.

**cDesc**

*Input*. Pointer to the output tensor descriptor.

**sizeInBytes**

*Output*. Minimum size of the index space to be passed to the reduction.

### Returns

**CUDNN_STATUS_SUCCESS**

The workspace size is returned successfully.

## 4.128. cudnnGetRNNBiasMode

```
cudnnStatus_t cudnnGetRNNBiasMode(
   cudnnRNNDescriptor_t   rnnDesc,
   cudnnRNNBiasMode_t     *biasMode)
```

This function retrieves the RNN bias mode that was configured by **cudnnSetRNNBiasMode**. The default value of **biasMode** in **rnnDesc** after **cudnnCreateRNNDescriptor**is **CUDNN_RNN_DOUBLE_BIAS**.

### Parameters

**rnnDesc**

*Input*. A previously created RNN descriptor.

***biasMode**

*Input*. Pointer to where RNN bias mode should be saved.

### Returns

**CUDNN_STATUS_BAD_PARAM**

Either the **rnnDesc** or ***biasMode** is **NULL**.

**CUDNN_STATUS_SUCCESS**

The **biasMode** parameter was retrieved set successfully.

## 4.129. cudnnGetRNNDataDescriptor

```
cudnnStatus_t cudnnGetRNNDataDescriptor(
    cudnnRNNDataDescriptor_t       RNNDataDesc,
    cudnnDataType_t                *dataType,
    cudnnRNNDataLayout_t           *layout,
    int                            *maxSeqLength,
    int                            *batchSize,
    int                            *vectorSize,
    int                            arrayLengthRequested,
```

```
    int                                seqLengthArray[],
    void                               *paddingFill);
```

This function retrieves a previously created RNN data descriptor object.

### Parameters

**RNNDataDesc**

*Input*. A previously created and initialized RNN descriptor.

**dataType**

*Output*. Pointer to the host memory location to store the datatype of the RNN data tensor.

**layout**

*Output*. Pointer to the host memory location to store the memory layout of the RNN data tensor.

**maxSeqLength**

*Output*. The maximum sequence length within this RNN data tensor, including the padding vectors.

**batchSize**

*Output*. The number of sequences within the mini-batch.

**vectorSize**

*Output*. The vector length (meaning, embedding size) of the input or output tensor at each time-step.

**arrayLengthRequested**

*Input*. The number of elements that the user requested for **seqLengthArray**.

**seqLengthArray**

*Output*. Pointer to the host memory location to store the integer array describing the length (meaning, number of time-steps) of each sequence. This is allowed to be a **NULL** pointer if **arrayLengthRequested** is 0.

**paddingFill**

*Output*. Pointer to the host memory location to store the user defined symbol. The symbol should be interpreted as the same data type as the RNN data tensor.

### Returns

**CUDNN_STATUS_SUCCESS**

The parameters are fetched successfully.

**CUDNN_STATUS_BAD_PARAM**

Any one of these have occurred:

- Any of **RNNDataDesc**, **dataType**, **layout**, **maxSeqLength**, **batchSize**, **vectorSize**, **paddingFill** is **NULL**.
- **seqLengthArray** is **NULL** while **arrayLengthRequested** is greater than zero.
- **arrayLengthRequested** is less than zero.

# 4.130. cudnnGetRNNDescriptor

```
cudnnStatus_t cudnnGetRNNDescriptor(
    cudnnHandle_t                handle,
    cudnnRNNDescriptor_t         rnnDesc,
    int *                        hiddenSize,
    int *                        numLayers,
    cudnnDropoutDescriptor_t *   dropoutDesc,
    cudnnRNNInputMode_t *        inputMode,
    cudnnDirectionMode_t *       direction,
    cudnnRNNMode_t *             mode,
    cudnnRNNAlgo_t *             algo,
    cudnnDataType_t *            dataType)
```

This function retrieves RNN network parameters that were configured by **cudnnSetRNNDescriptor()**. All pointers passed to the function should be not-**NULL** or **CUDNN_STATUS_BAD_PARAM** is reported. The function does not check the validity of retrieved network parameters. The parameters are verified when they are written to the RNN descriptor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously created and initialized RNN descriptor.

**hiddenSize**

*Output*. Pointer to where the size of the hidden state should be stored (the same value is used in every layer).

**numLayers**

*Output*. Pointer to where the number of RNN layers should be stored.

**dropoutDesc**

*Output*. Pointer to where the handle to a previously configured dropout descriptor should be stored.

**inputMode**

*Output*. Pointer to where the mode of the first RNN layer should be saved.

**direction**

*Output*. Pointer to where RNN uni-directional/bi-directional mode should be saved.

**mode**

*Output*. Pointer to where RNN cell type should be saved.

**algo**

*Output*. Pointer to where RNN algorithm type should be stored.

**dataType**

*Output*. Pointer to where the data type of RNN weights/biases should be stored.

### Returns

**CUDNN_STATUS_SUCCESS**

RNN parameters were successfully retrieved from the RNN descriptor.

**CUDNN_STATUS_BAD_PARAM**

At least one pointer passed to the **cudnnGetRNNDescriptor()** function is **NULL**.

## 4.131. cudnnGetRNNLinLayerBiasParams

```
cudnnStatus_t cudnnGetRNNLinLayerBiasParams(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        pseudoLayer,
    const cudnnTensorDescriptor_t    xDesc,
    const cudnnFilterDescriptor_t    wDesc,
    const void                      *w,
    const int                        linLayerID,
    cudnnFilterDescriptor_t          linLayerBiasDesc,
    void                           **linLayerBias)
```

This function is used to obtain a pointer and a descriptor of every RNN bias column vector in each pseudo-layer within the recurrent network defined by **rnnDesc** and its input width specified in **xDesc**.

The **cudnnGetRNNLinLayerBiasParams()** function was changed in cuDNN version 7.1.1 to match the behavior of **cudnnGetRNNLinLayerMatrixParams()**.

The **cudnnGetRNNLinLayerBiasParams()** function returns the RNN bias vector size in two dimensions: rows and columns.

Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. In previous versions of the cuDNN library, the function returns the total number of vector elements in **linLayerBiasDesc** as follows:

```
filterDimA[0]=total_size,
filterDimA[1]=1,
filterDimA[2]=1
```

For more information, see the description of the cudnnGetFilterNdDescriptor function.

In v7.1.1, the format was changed to:

```
filterDimA[0]=1,
filterDimA[1]=rows,
```

```
filterDimA[2]=1 (number of columns)
```

In both cases, the **format** field of the filter descriptor should be ignored when retrieved by **cudnnGetFilterNdDescriptor()**.

The RNN implementation in cuDNN uses two bias vectors before the cell non-linear function. Note that the RNN implementation in cuDNN depends on the number of bias vectors before the cell non-linear function. Refer to the equations in the cudnnRNNMode_t description, for the enumerant type based on the value of **cudnnRNNBiasMode_t biasMode** in **rnnDesc**. If nonexistent biases are referenced by **linLayerID**, then this function sets **linLayerBiasDesc** to a zeroed filter descriptor where:

```
filterDimA[0]=0,
filterDimA[1]=0, and
filterDimA[2]=2
```

and sets **linLayerBias** to **NULL**. Refer to the details for function parameter **linLayerID** to determine the relevant values of **linLayerID** based on **biasMode**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**pseudoLayer**

*Input*. The pseudo-layer to query. In uni-directional RNNs, a pseudo-layer is the same as a physical layer (**pseudoLayer=0** is the RNN input layer, **pseudoLayer=1** is the first hidden layer). In bi-directional RNNs, there are twice as many pseudo-layers in comparison to physical layers.

- **pseudoLayer=0** refers to the forward part of the physical input layer
- **pseudoLayer=1** refers to the backward part of the physical input layer
- **pseudoLayer=2** is the forward part of the first hidden layer, and so on

**xDesc**

*Input*. A fully packed tensor descriptor describing the input to one recurrent iteration (to retrieve the RNN input width).

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**linLayerID**

*Input*. The linear layer to obtain information about:

- - If **mode** in **rnnDesc** was set to **CUDNN_RNN_RELU** or **CUDNN_RNN_TANH**:
    - Value 0 references the bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).
    - Value 1 references the bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
  - If **mode** in **rnnDesc** was set to **CUDNN_LSTM**:
    - Values of 0, 1, 2 and 3 reference bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).
    - Values of 4, 5, 6 and 7 reference bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
    - Values and their associated gates:
      - Values 0 and 4 reference the input gate.
      - Values 1 and 5 reference the forget gate.
      - Values 2 and 6 reference the new memory gate.
      - Values 3 and 7 reference the output gate.
  - If **mode** in **rnnDesc** was set to **CUDNN_GRU**:
    - Values of 0, 1 and 2 reference bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).
    - Values of 3, 4 and 5 reference bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
    - Values and their associated gates:
      - Values 0 and 3 reference the reset gate.
      - Values 1 and 4 reference the update gate.
      - Values 2 and 5 reference the new memory gate.

For more information on modes and bias modes, see cudnnRNNMode_t.

**linLayerBiasDesc**

*Output*. Handle to a previously created filter descriptor.

**linLayerBias**

*Output*. Data pointer to GPU memory associated with the filter descriptor **linLayerBiasDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the following arguments is **NULL**: **handle**, **rnnDesc**, **xDesc**, **wDesc**, **linLayerBiasDesc**, **linLayerBias**.
- A data type mismatch was detected between **rnnDesc** and other descriptors.
- Minimum requirement for the **w** pointer alignment is not satisfied.
- The value of **pseudoLayer** or **linLayerID** is out of range.

**CUDNN_STATUS_INVALID_VALUE**

Some elements of the **linLayerBias** vector are outside the **w** buffer boundaries as specified by the **wDesc** descriptor.

## 4.132. cudnnGetRNNLinLayerMatrixParams

```
cudnnStatus_t cudnnGetRNNLinLayerMatrixParams(
cudnnHandle_t                   handle,
const cudnnRNNDescriptor_t      rnnDesc,
const int                       pseudoLayer,
const cudnnTensorDescriptor_t   xDesc,
const cudnnFilterDescriptor_t   wDesc,
const void                     *w,
const int                       linLayerID,
cudnnFilterDescriptor_t         linLayerMatDesc,
void                          **linLayerMat)
```

This function is used to obtain a pointer and a descriptor of every RNN weight matrix in each pseudo-layer within the recurrent network defined by **rnnDesc** and its input width specified in **xDesc**.

The **cudnnGetRNNLinLayerMatrixParams()** function was enhanced in cuDNN version 7.1.1 without changing its prototype. Instead of reporting the total number of elements in each weight matrix in the **linLayerMatDesc** filter descriptor, the function returns the matrix size as two dimensions: rows and columns. Moreover, when a weight matrix does not exist, for example, due to **CUDNN_SKIP_INPUT** mode, the function returns **NULL** in **linLayerMat** and all fields of **linLayerMatDesc** are zero.

The **cudnnGetRNNLinLayerMatrixParams()** function returns the RNN matrix size in two dimensions: rows and columns. This allows the user to easily print and initialize RNN weight matrices. Elements in each weight matrix are arranged in the row-major order. Due to historical reasons, the minimum number of

dimensions in the filter descriptor is three. In previous versions of the cuDNN library, the function returned the total number of weights in **linLayerMatDesc** as follows: **filterDimA[0]=total_size**, **filterDimA[1]=1**, **filterDimA[2]=1** (see the description of the **cudnnGetFilterNdDescriptor()** function). In v7.1.1, the format was changed to: **filterDimA[0]=1**, **filterDimA[1]=rows**, **filterDimA[2]=columns**. In both cases, the "format" field of the filter descriptor should be ignored when retrieved by **cudnnGetFilterNdDescriptor()**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**pseudoLayer**

*Input*. The pseudo-layer to query. In uni-directional RNNs, a pseudo-layer is the same as a physical layer (**pseudoLayer=0** is the RNN input layer, **pseudoLayer=1** is the first hidden layer). In bi-directional RNNs, there are twice as many pseudo-layers in comparison to physical layers.

- **pseudoLayer=0** refers to the forward part of the physical input layer
- **pseudoLayer=1** refers to the backward part of the physical input layer
- **pseudoLayer=2** is the forward part of the first hidden layer, and so on

**xDesc**

*Input*. A fully packed tensor descriptor describing the input to one recurrent iteration (to retrieve the RNN input width).

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**linLayerID**

*Input*. The linear layer to obtain information about:

- - If **mode** in **rnnDesc** was set to **CUDNN_RNN_RELU** or **CUDNN_RNN_TANH**:
    - Value 0 references the bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).

    - Value 1 references the bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
  - If **mode** in **rnnDesc** was set to **CUDNN_LSTM**:
    - Values of 0, 1, 2 and 3 reference bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).
    - Values of 4, 5, 6 and 7 reference bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
    - Values and their associated gates:
      - Values 0 and 4 reference the input gate.
      - Values 1 and 5 reference the forget gate.
      - Values 2 and 6 reference the new memory gate.
      - Values 3 and 7 reference the output gate.
  - If **mode** in **rnnDesc** was set to **CUDNN_GRU**:
    - Values of 0, 1 and 2 reference bias applied to the input from the previous layer (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_SINGLE_INP_BIAS** or **CUDNN_RNN_DOUBLE_BIAS**).
    - Values of 3, 4 and 5 reference bias applied to the recurrent input (relevant if **biasMode** in **rnnDesc** is **CUDNN_RNN_DOUBLE_BIAS** or **CUDNN_RNN_SINGLE_REC_BIAS**).
    - Values and their associated gates:
      - Values 0 and 3 reference the reset gate.
      - Values 1 and 4 reference the update gate.
      - Values 2 and 5 reference the new memory gate.

For more information on modes and bias modes, see cudnnRNNMode_t.

**linLayerMatDesc**

*Output*. Handle to a previously created filter descriptor. When the weight matrix does not exist, the returned filer descriptor has all fields set to zero.

**linLayerMat**

*Output*. Data pointer to GPU memory associated with the filter descriptor **linLayerMatDesc**. When the weight matrix does not exist, the returned pointer is **NULL**.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the following arguments is **NULL**: **handle**, **rnnDesc**, **xDesc**, **wDesc**, **linLayerMatDesc**, **linLayerMat**.
- A data type mismatch was detected between **rnnDesc** and other descriptors.
- Minimum requirement for the **w** pointer alignment is not satisfied.
- The value of **pseudoLayer** or **linLayerID** is out of range.

**CUDNN_STATUS_INVALID_VALUE**

Some elements of the **linLayerMat** vector are outside the **w** buffer boundaries as specified by the **wDesc** descriptor.

# 4.133. cudnnGetRNNPaddingMode

```
cudnnStatus_t cudnnGetRNNPaddingMode(
    cudnnRNNDescriptor_t        rnnDesc,
    cudnnRNNPaddingMode_t       *paddingMode)
```

This function retrieves the RNN padding mode from the RNN descriptor.

### Parameters

**rnnDesc**

*Input/Output*. A previously created RNN descriptor.

***paddingMode**

*Input*. Pointer to the host memory where the RNN padding mode is saved.

### Returns

**CUDNN_STATUS_SUCCESS**

The RNN padding mode parameter was retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**

Either the **rnnDesc** or ***paddingMode** is **NULL**.

# 4.134. cudnnGetRNNParamsSize

```
cudnnStatus_t cudnnGetRNNParamsSize(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const cudnnTensorDescriptor_t   xDesc,
```

```
    size_t                          *sizeInBytes,
    cudnnDataType_t                  dataType)
```

This function is used to query the amount of parameter space required to execute the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**xDesc**

*Input*. A fully packed tensor descriptor describing the input to one recurrent iteration.

**sizeInBytes**

*Output*. Minimum amount of GPU memory needed as parameter space to be able to execute an RNN with the specified descriptor and input tensors.

**dataType**

*Input*. The data type of the parameters.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- The descriptor **xDesc** is invalid.
- The descriptor **xDesc** is not fully packed.
- The combination of **dataType** and tensor descriptor data type is invalid.

**CUDNN_STATUS_NOT_SUPPORTED**

The combination of the RNN descriptor and tensor descriptors is not supported.

## 4.135. cudnnGetRNNProjectionLayers

```
cudnnStatus_t cudnnGetRNNProjectionLayers(
    cudnnHandle_t           handle,
    cudnnRNNDescriptor_t    rnnDesc,
    int                     *recProjSize,
    int                     *outProjSize)
```

This function retrieves the current RNN projection parameters. By default, the projection feature is disabled so invoking this function immediately after **cudnnSetRNNDescriptor()** will yield **recProjSize** equal to **hiddenSize** and **outProjSize** set to zero. The **cudnnSetRNNProjectionLayers()** method enables the RNN projection.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously created and initialized RNN descriptor.

**recProjSize**

*Output*. Pointer where the recurrent projection size should be stored.

**outProjSize**

*Output*. Pointer where the output projection size should be stored.

### Returns

**CUDNN_STATUS_SUCCESS**

RNN projection parameters were retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**

A **NULL** pointer was passed to the function.

## 4.136. cudnnGetRNNTrainingReserveSize

```
cudnnStatus_t cudnnGetRNNTrainingReserveSize(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t   *xDesc,
    size_t                          *sizeInBytes)
```

This function is used to query the amount of reserved space required for training the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**. The same reserved space buffer must be passed to **cudnnRNNForwardTraining**, **cudnnRNNBackwardData** and **cudnnRNNBackwardWeights**. Each of these calls overwrites the contents of the reserved space, however it can safely be backed up and restored between calls if reuse of the memory is desired.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**sizeInBytes**

*Output*. Minimum amount of GPU memory needed as reserve space to be able to train an RNN with the specified descriptor and input tensors.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors in **xDesc** is invalid.
- The descriptors in **xDesc** have inconsistent second dimensions, strides or data types.
- The descriptors in **xDesc** have increasing first dimensions.
- The descriptors in **xDesc** is not fully packed.

**CUDNN_STATUS_NOT_SUPPORTED**

The the data types in tensors described by **xDesc** is not supported.

## 4.137. cudnnGetRNNWorkspaceSize

```
cudnnStatus_t cudnnGetRNNWorkspaceSize(
    cudnnHandle_t                     handle,
    const cudnnRNNDescriptor_t        rnnDesc,
    const int                         seqLength,
    const cudnnTensorDescriptor_t    *xDesc,
    size_t                           *sizeInBytes)
```

This function is used to query the amount of work space required to execute the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. Workspace that is allocated, based on the size that this function provides, cannot be used for sequences longer than **seqLength**.

**xDesc**

*Input*. An array of tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. For example, if you have multiple time series in a batch, they can be different lengths. This dimension is the batch size for the particular iteration of the sequence, and so it should decrease when a sequence in the batch has been terminated.

Each tensor descriptor must have the same second dimension (vector length).

**sizeInBytes**

*Output*. Minimum amount of GPU memory needed as workspace to be able to execute an RNN with the specified descriptor and input tensors.

### Returns

**CUDNN_STATUS_SUCCESS**

The query was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors in **xDesc** is invalid.
- The descriptors in **xDesc** have inconsistent second dimensions, strides or data types.
- The descriptors in **xDesc** have increasing first dimensions.
- The descriptors in **xDesc** is not fully packed.

**CUDNN_STATUS_NOT_SUPPORTED**

The data types in tensors described by **xDesc** is not supported.

# 4.138. cudnnGetSeqDataDescriptor

```
cudnnStatus_t cudnnGetSeqDataDescriptor(
 const cudnnSeqDataDescriptor_t seqDataDesc,
 cudnnDataType_t *dataType,
 int *nbDims,
 int nbDimsRequested,
 int dimA[],
 cudnnSeqDataAxis_t axes[],
 size_t *seqLengthArraySize,
 size_t seqLengthSizeRequested,
 int seqLengthArray[],
 void *paddingFill);
```

This function retrieves settings from a previously created sequence data descriptor. The user can assign **NULL** to any pointer except **seqDataDesc** when the retrieved value is not needed. The **nbDimsRequested** argument applies to both **dimA[]** and **axes[]** arrays. A positive value of **nbDimsRequested** or **seqLengthSizeRequested** is ignored when the corresponding array, **dimA[]**, **axes[]**, or **seqLengthArray[]** is **NULL**.

The **cudnnGetSeqDataDescriptor()** function does not report the actual strides in the sequence data buffer. Those strides can be handy in computing the offset to any sequence data element. The user must precompute strides based on the **axes[]** and **dimA[]** arrays reported by the **cudnnGetSeqDataDescriptor()** function. Below is sample code that performs this task:

```
// Array holding sequence data strides.
size_t strA[CUDNN_SEQDATA_DIM_COUNT] = {0};

// Compute strides from dimension and order arrays.
size_t stride = 1;
for (int i = nbDims - 1; i >= 0; i--) {
 int j = int(axes[i]);
 if (unsigned(j) < CUDNN_SEQDATA_DIM_COUNT-1 && strA[j] == 0) {
    strA[j] = stride;
    stride *= dimA[j];
 } else {
     fprintf(stderr, "ERROR: invalid axes[%d]=%d\n\n", i, j);
     abort();
 }
}
```

Now, the **strA[]** array can be used to compute the index to any sequence data element, for example:

```
// Using four indices (batch, beam, time, vect) with ranges already checked.
size_t base = strA[CUDNN_SEQDATA_BATCH_DIM] * batch
         + strA[CUDNN_SEQDATA_BEAM_DIM]  * beam
         + strA[CUDNN_SEQDATA_TIME_DIM]  * time;
val = seqDataPtr[base + vect];
```

The above code assumes that all four indices (**batch, beam, time, vect**) are less than the corresponding value in the **dimA[]** array. The sample code also omits the **strA[CUDNN_SEQDATA_VECT_DIM]** stride because its value is always **1**, meaning, elements of one vector occupy a contiguous block of memory.

### Parameters

**seqDataDesc**
*Input.* Sequence data descriptor.

**dataType**
*Output.* Data type used in the sequence data buffer.

**nbDims**
*Output.* The number of active dimensions in the **dimA[]** and **axes[]** arrays.

**nbDimsRequested**
*Input.* The maximum number of consecutive elements that can be written to **dimA[]** and **axes[]** arrays starting from index zero. The recommended value for this argument is **CUDNN_SEQDATA_DIM_COUNT**.

**dimA[]**
*Output.* Integer array holding sequence data dimensions.

**axes[]**
*Output.* Array of **cudnnSeqDataAxis_t** that defines the layout of sequence data in memory.

**seqLengthArraySize**
*Output.* The number of required elements in **seqLengthArray[]** to save all sequence lengths.

**seqLengthSizeRequested**
*Input.* The maximum number of consecutive elements that can be written to the **seqLengthArray[]** array starting from index zero.

**seqLengthArray[]**
*Output.* Integer array holding sequence lengths.

**paddingFill**
*Output.* Pointer to a storage location of **dataType** with the fill value that should be written to all padding vectors. Use **NULL** when an explicit initialization of output padding vectors was not requested.

### Returns

**CUDNN_STATUS_SUCCESS**
Requested sequence data descriptor fields were retrieved successfully.

**CUDNN_STATUS_BAD_PARAM**
An invalid input argument was found.

**CUDNN_STATUS_INTERNAL_ERROR**
An inconsistent internal state was encountered.

## 4.139. cudnnGetStream

```
cudnnStatus_t cudnnGetStream(
    cudnnHandle_t   handle,
    cudaStream_t   *streamId)
```

This function retrieves the user CUDA stream programmed in the cuDNN handle. When the user's CUDA stream is not set in the cuDNN handle, this function reports the null-stream.

### Parameters

**handle**

*Input*. Pointer to the cuDNN handle.

**streamID**

*Output*. Pointer where the current CUDA stream from the cuDNN handle should be stored.

### Returns

**CUDNN_STATUS_BAD_PARAM**

Invalid (**NULL**) handle.

**CUDNN_STATUS_SUCCESS**

The stream identifier was retrieved successfully.

## 4.140. cudnnGetTensor4dDescriptor

```
cudnnStatus_t cudnnGetTensor4dDescriptor(
    const cudnnTensorDescriptor_t  tensorDesc,
    cudnnDataType_t          *dataType,
    int                      *n,
    int                      *c,
    int                      *h,
    int                      *w,
    int                      *nStride,
    int                      *cStride,
    int                      *hStride,
    int                      *wStride)
```

This function queries the parameters of the previously initialized Tensor4D descriptor object.

### Parameters

**tensorDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**datatype**

*Output*. Data type.

**n**

*Output*. Number of images.

**c**

*Output*. Number of feature maps per image.

**h**

*Output*. Height of each feature map.

**w**

*Output*. Width of each feature map.

**nStride**

*Output*. Stride between two consecutive images.

**cStride**

*Output*. Stride between two consecutive feature maps.

**hStride**

*Output*. Stride between two consecutive rows.

**wStride**

*Output*. Stride between two consecutive columns.

### Returns

**CUDNN_STATUS_SUCCESS**

The operation succeeded.

## 4.141. cudnnGetTensorNdDescriptor

```
cudnnStatus_t cudnnGetTensorNdDescriptor(
    const cudnnTensorDescriptor_t   tensorDesc,
    int                             nbDimsRequested,
    cudnnDataType_t                *dataType,
    int                            *nbDims,
    int                             dimA[],
    int                             strideA[])
```

This function retrieves values stored in a previously initialized tensor descriptor object.

### Parameters

**tensorDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**nbDimsRequested**

*Input*. Number of dimensions to extract from a given tensor descriptor. It is also the minimum size of the arrays **dimA** and **strideA**. If this number is greater than the resulting **nbDims[0]**, only **nbDims[0]** dimensions will be returned.

**datatype**

*Output*. Data type.

**nbDims**

*Output*. Actual number of dimensions of the tensor will be returned in **nbDims[0]**.

**dimA**

*Output*. Array of dimension of at least **nbDimsRequested** that will be filled with the dimensions from the provided tensor descriptor.

**strideA**

*Input*. Array of dimension of at least **nbDimsRequested** that will be filled with the strides from the provided tensor descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The results were returned successfully.

**CUDNN_STATUS_BAD_PARAM**

Either **tensorDesc** or **nbDims** pointer is **NULL**.

## 4.142. cudnnGetTensorSizeInBytes

```
cudnnStatus_t cudnnGetTensorSizeInBytes(
    const cudnnTensorDescriptor_t   tensorDesc,
    size_t                          *size)
```

This function returns the size of the tensor in memory in respect to the given descriptor. This function can be used to know the amount of GPU memory to be allocated to hold that tensor.

### Parameters

**tensorDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**size**

*Output*. Size in bytes needed to hold the tensor in GPU memory.

### Returns

**CUDNN_STATUS_SUCCESS**

The results were returned successfully.

# 4.143. cudnnGetTensorTransformDescriptor

```
cudnnStatus_t cudnnGetTensorTransformDescriptor(
 cudnnTensorTransformDescriptor_t transformDesc,
 uint32_t nbDimsRequested,
 cudnnTensorFormat_t *destFormat,
 int32_t padBeforeA[],
 int32_t padAfterA[],
 uint32_t foldA[],
 cudnnFoldingDirection_t *direction);
```

This function returns the values stored in a previously initialized tensor transform descriptor.

### Parameters

**transformDesc**
: *Input*. A previously initialized tensor transform descriptor.

**nbDimsRequested**
: Input. The number of dimensions to consider. For more information, see the Tensor Descriptor section in the *cuDNN Developer Guide*.

**destFormat**
: *Output*. The transform format that will be returned.

**padBeforeA[]**
: *Output*. An array filled with the amount of padding to add before each dimension. The dimension of this **padBeforeA[]** parameter equal to **nbDimsRequested**.

**padAfterA[]**
: *Output*. An array filled with the amount of padding to add after each dimension. The dimension of this **padBeforeA[]** parameter is equal to **nbDimsRequested**.

**foldA[]**
: *Output*. An array that was filled with the folding parameters for each spatial dimension. The dimension of this **foldA[]** array is **nbDimsRequested-2**.

**direction**
: *Output*. The setting that selects folding or unfolding. For more information, see cudnnFoldingDirection_t.

### Returns

**CUDNN_STATUS_SUCCESS**
: The results were obtained successfully.

**CUDNN_STATUS_BAD_PARAM**
: If **transformDesc** is **NULL** or if **nbDimsRequested** is less than 3 or greater than **CUDNN_DIM_MAX**.

# 4.144. cudnnGetVersion

```
size_t cudnnGetVersion()
```

This function returns the version number of the cuDNN library. It returns the **CUDNN_VERSION** define present in the **cudnn.h** header file. Starting with release R2, the routine can be used to identify dynamically the current cuDNN library used by the application. The define **CUDNN_VERSION** can be used to have the same application linked against different cuDNN versions using conditional compilation statements.

# 4.145. cudnnIm2Col

```
cudnnStatus_t cudnnIm2Col(
    cudnnHandle_t                   handle,
    cudnnTensorDescriptor_t         srcDesc,
    const void                      *srcData,
    cudnnFilterDescriptor_t         filterDesc,
    cudnnConvolutionDescriptor_t    convDesc,
    void                            *colBuffer)
```

This function constructs the **A** matrix necessary to perform a forward pass of GEMM convolution. This **A** matrix has a height of **batch_size*y_height*y_width** and width of **input_channels*filter_height*filter_width**, where:

- **batch_size** is **xDesc** first dimension
- **y_height/y_width** are computed from **cudnnGetConvolutionNdForwardOutputDim()**
- **input_channels** is **xDesc** second dimension
- **filter_height/filter_width** are **wDesc** third and fourth dimension

The **A** matrix is stored in format HW fully-packed in GPU memory.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**srcDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**srcData**

*Input*. Data pointer to GPU memory associated with the input tensor descriptor.

**filterDesc**

*Input*. Handle to a previously initialized filter descriptor.

**convDesc**

*Input*. Handle to a previously initialized convolution descriptor.

**colBuffer**

*Output*. Data pointer to GPU memory storing the output matrix.

### Returns

**CUDNN_STATUS_BAD_PARAM**

**srcData** or **colBuffer** is **NULL**.

**CUDNN_STATUS_NOT_SUPPORTED**

Any of **srcDesc**, **filterDesc**, **convDesc** has **dataType** of **CUDNN_DATA_INT8**, **CUDNN_DATA_INT8x4**, **CUDNN_DATA_INT8** or **CUDNN_DATA_INT8x4** **convDesc** has **groupCount** larger than 1.

**CUDNN_STATUS_EXECUTION_FAILED**

The CUDA kernel execution was unsuccessful.

**CUDNN_STATUS_SUCCESS**

The output data array is successfully generated.

## 4.146. cudnnInitTransformDest

```
cudnnStatus_t cudnnInitTransformDest(
 const cudnnTensorTransformDescriptor_t transformDesc,
 const cudnnTensorDescriptor_t srcDesc,
 cudnnTensorDescriptor_t destDesc,
 size_t *destSizeInBytes);
```

This function initializes and returns a destination tensor descriptor **destDesc** for tensor transform operations. The initialization is done with the desired parameters described in the transform descriptor cudnnTensorDescriptor_t.

The returned tensor descriptor will be packed.

### Parameters

**transformDesc**

*Input*. Handle to a previously initialized tensor transform descriptor.

**srcDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**destDesc**

*Output*. Handle of the tensor descriptor that will be initialized and returned.

**destSizeInBytes**

*Output*. A pointer to hold the size, in bytes, of the new tensor.

### Returns

**CUDNN_STATUS_SUCCESS**
The tensor descriptor was initialized successfully.

**CUDNN_STATUS_BAD_PARAM**
If either **srcDesc** or **destDesc** is **NULL**, or if the tensor descriptor's **nbDims** is incorrect. For more information, see the Tensor Descriptor section in the *cuDNN Developer Guide*.

**CUDNN_STATUS_NOT_SUPPORTED**
If the provided configuration is not 4D.

**CUDNN_STATUS_EXECUTION_FAILED**
Function failed to launch on the GPU.

## 4.147. cudnnLRNCrossChannelBackward

```
cudnnStatus_t cudnnLRNCrossChannelBackward(
    cudnnHandle_t                      handle,
    cudnnLRNDescriptor_t               normDesc,
    cudnnLRNMode_t                     lrnMode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      yDesc,
    const void                        *y,
    const cudnnTensorDescriptor_t      dyDesc,
    const void                        *dy,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const void                        *beta,
    const cudnnTensorDescriptor_t      dxDesc,
    void                              *dx)
```

This function performs the backward LRN layer computation.

Supported formats are: **positive-strided**, NCHW for 4D **x** and **y**, and only NCDHW DHW-packed for 5D (for both **x** and **y**). Only non-overlapping 4D and 5D tensors are supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**normDesc**

*Input*. Handle to a previously initialized LRN parameter descriptor.

**lrnMode**

*Input*. LRN layer mode of operation. Currently only **CUDNN_LRN_CROSS_CHANNEL_DIM1** is implemented. Normalization is performed along the tensor's **dimA[1]**.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**yDesc, y**

*Input*. Tensor descriptor and pointer in device memory for the layer's **y** data.

**dyDesc, dy**

*Input*. Tensor descriptor and pointer in device memory for the layer's input cumulative loss differential data **dy** (including error backpropagation).

**xDesc, x**

*Input*. Tensor descriptor and pointer in device memory for the layer's **x** data. Note that these values are not modified during backpropagation.

**dxDesc, dx**

*Output*. Tensor descriptor and pointer in device memory for the layer's resulting cumulative loss differential data **dx** (including error backpropagation).

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the tensor pointers **x**, **y** is **NULL**.
- Number of input tensor dimensions is 2 or less.
- LRN descriptor parameters are outside of their valid ranges.
- One of tensor parameters is 5D but is not in NCDHW DHW-packed format.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- Any of the input tensor datatypes is not the same as any of the output tensor datatype.
- Any pairwise tensor dimensions mismatch for **x**, **y**, **dx**, **dy**.
- Any tensor parameters strides are negative.

# 4.148. cudnnLRNCrossChannelForward

```
cudnnStatus_t cudnnLRNCrossChannelForward(
    cudnnHandle_t                    handle,
    cudnnLRNDescriptor_t             normDesc,
    cudnnLRNMode_t                   lrnMode,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This function performs the forward LRN layer computation.

Supported formats are: **positive-strided**, NCHW for 4D **x** and **y**, and only NCDHW DHW-packed for 5D (for both **x** and **y**). Only non-overlapping 4D and 5D tensors are supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**normDesc**

*Input*. Handle to a previously initialized LRN parameter descriptor.

**lrnMode**

*Input*. LRN layer mode of operation. Currently only **CUDNN_LRN_CROSS_CHANNEL_DIM1** is implemented. Normalization is performed along the tensor's **dimA[1]**.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc, yDesc**

*Input*. Tensor descriptor objects for the input and output tensors.

**x**

*Input*. Input tensor data pointer in device memory.

**y**

*Output*. Output tensor data pointer in device memory.

### Returns

**CUDNN_STATUS_SUCCESS**

The computation was performed successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- One of the tensor pointers **x**, **y** is **NULL**.
- Number of input tensor dimensions is 2 or less.
- LRN descriptor parameters are outside of their valid ranges.
- One of tensor parameters is 5D but is not in NCDHW DHW-packed format.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- Any of the input tensor datatypes is not the same as any of the output tensor datatype.
- **x** and **y** tensor dimensions mismatch.
- Any tensor parameters strides are negative.

## 4.149. cudnnMakeFusedOpsPlan

```
cudnnStatus_t cudnnMakeFusedOpsPlan(
 cudnnHandle_t handle,
 cudnnFusedOpsPlan_t plan,
 const cudnnFusedOpsConstParamPack_t constPack,
 size_t *workspaceSizeInBytes);
```

This function determines the optimum kernel to execute, and the workspace size the user should allocate, prior to the actual execution of the fused operations by cudnnFusedOpsExecute.

### Parameters

**handle**

*Input*. Pointer to the cuDNN library context.

**plan**

*Input*. Pointer to a previously-created and initialized plan descriptor.

**constPack**

*Input*. Pointer to the descriptor to the const parameters pack.

**workspaceSizeInBytes**

*Output*. The amount of workspace size the user should allocate for the execution of this plan.

### Returns

**CUDNN_STATUS_BAD_PARAM**
If any of the inputs is **NULL**, or if the type of **cudnnFusedOps_t** in the **constPack** descriptor is unsupported.

**CUDNN_STATUS_SUCCESS**
The function executed successfully.

## 4.150. cudnnMultiHeadAttnBackwardData

```
cudnnStatus_t cudnnMultiHeadAttnBackwardData(
 cudnnHandle_t handle,
 const cudnnAttnDescriptor_t attnDesc,
 const int loWinIdx[],
 const int hiWinIdx[],
 const int devSeqLengthsDQDO[],
 const int devSeqLengthsDKDV[],
 const cudnnSeqDataDescriptor_t doDesc,
 const void *dout,
 const cudnnSeqDataDescriptor_t dqDesc,
 void *dqueries,
 const void *queries,
 const cudnnSeqDataDescriptor_t dkDesc,
 void *dkeys,
 const void *keys,
 const cudnnSeqDataDescriptor_t dvDesc,
 void *dvalues,
 const void *values,
 size_t weightSizeInBytes,
 const void *weights,
 size_t workSpaceSizeInBytes,
 void *workSpace,
 size_t reserveSpaceSizeInBytes,
 void *reserveSpace);
```

This function computes exact, first-order derivatives of the multi-head attention block with respect to its inputs: **Q, K, V**. If **y**=*F*(**w**) is a vector-valued function that represents the multi-head attention layer and it takes some vector $x \in \mathbb{R}^n$ as an input (with all other parameters and inputs constant), and outputs vector $y \in \mathbb{R}^m$, then **cudnnMultiHeadAttnBackwardData()** computes the result of $\left(\partial y_i / \partial x_j\right)^T \delta_{\text{out}}$ where $\delta_{\text{out}}$ is the $m \times 1$ gradient of the loss function with respect to multi-head attention outputs. The $\delta_{\text{out}}$ gradient is back propagated through prior layers of the deep learning model. $\partial y_i / x_j$ is the $m \times n$ Jacobian matrix of $F(\mathbf{x})$. The input is supplied via the **dout** argument and gradient results for **Q, K, V** are written to the **dqueries**, **dkeys**, and **dvalues** buffers.

The **cudnnMultiHeadAttnBackwardData()** function does not output partial derivatives for residual connections because this result is equal to $\delta_{\text{out}}$. If the multi-head attention model enables residual connections sourced directly from **Q**, then the **dout** tensor needs to be added to **dqueries** to obtain the correct result of the latter. This operation is demonstrated in the cuDNN **multiHeadAttention** sample code.

The **cudnnMultiHeadAttnBackwardData()** function must be invoked after **cudnnMultiHeadAttnForward()**. The **loWinIdx[]**, **hiWinIdx[]**, **queries**,

**keys**, **values**, **weights**, and **reserveSpace** arguments should be the same as in the **cudnnMultiHeadAttnForward()** call. **devSeqLengthsDQDO[]** and **devSeqLengthsDKDV[]** device arrays should contain the same start and end attention window indices as **devSeqLengthsQO[]** and **devSeqLengthsKV[]** arrays in the forward function invocation.

**cudnnMultiHeadAttnBackwardData()** does not verify that sequence lengths stored in **devSeqLengthsDQDO[]** and **devSeqLengthsDKDV[]** contain the same settings as **seqLengthArray[]** in the corresponding sequence data descriptor.

### Parameters

**handle**

*Input*. The current cuDNN context handle.

**attnDesc**

*Input*. A previously initialized attention descriptor.

**loWinIdx[], hiWinIdx[]**

*Input*. Two host integer arrays specifying the start and end indices of the attention window for each **Q** time-step. The start index in **K, V** sets is inclusive, and the end index is exclusive.

**devSeqLengthsDQDO[]**

*Input*. Device array containing a copy of the sequence length array from the **dqDesc** or **doDesc** sequence data descriptor.

**devSeqLengthsDKDV[]**

*Input*. Device array containing a copy of the sequence length array from the **dkDesc** or **dvDesc** sequence data descriptor.

**doDesc**

*Input*. Descriptor for the $\delta_{out}$ gradients (vectors of partial derivatives of the loss function with respect to the multi-head attention outputs).

**dout**

Pointer to $\delta_{out}$ gradient data in the device memory.

**dqDesc**

*Input*. Descriptor for **queries** and **dqueries** sequence data.

**dqueries**

*Output*. Device pointer to gradients of the loss function computed with respect to **queries** vectors.

**queries**

*Input*. Pointer to **queries** data in the device memory. This is the same input as in **cudnnMultiHeadAttnForward()**.

**dkDesc**

*Input*. Descriptor for **keys** and **dkeys** sequence data.

**dkeys**

*Output*. Device pointer to gradients of the loss function computed with respect to **keys** vectors.

**keys**

*Input*. Pointer to **keys** data in the device memory. This is the same input as in **cudnnMultiHeadAttnForward()**.

**dvDesc**

*Input*. Descriptor for **values** and **dvalues** sequence data.

**dvalues**

*Output*. Device pointer to gradients of the loss function computed with respect to **values** vectors.

**values**

*Input*. Pointer to **values** data in the device memory. This is the same input as in **cudnnMultiHeadAttnForward()**.

**weightSizeInBytes**

*Input*. Size of the **weight** buffer in bytes where all multi-head attention trainable parameters are stored.

**weights**

*Input*. Address of the **weight** buffer in the device memory.

**workSpaceSizeInBytes**

*Input*. Size of the work-space buffer in bytes used for temporary API storage.

**workSpace**

*Input/Output*. Address of the work-space buffer in the device memory.

**reserveSpaceSizeInBytes**

*Input*. Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls.

**reserveSpace**

*Input/Output*. Address to the reserve-space buffer in the device memory.

**Returns**

**CUDNN_STATUS_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN_STATUS_BAD_PARAM**

An invalid or incompatible input argument was encountered.

**CUDNN_STATUS_EXECUTION_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN_STATUS_INTERNAL_ERROR**

An inconsistent internal state was encountered.

**CUDNN_STATUS_NOT_SUPPORTED**

A requested option or a combination of input arguments is not supported.

**CUDNN_STATUS_ALLOC_FAILED**

Insufficient amount of shared memory to launch a GPU kernel.

# 4.151. cudnnMultiHeadAttnBackwardWeights

```
cudnnStatus_t cudnnMultiHeadAttnBackwardWeights(
 cudnnHandle_t handle,
 const cudnnAttnDescriptor_t attnDesc,
 cudnnWgradMode_t addGrad,
 const cudnnSeqDataDescriptor_t qDesc,
 const void *queries,
 const cudnnSeqDataDescriptor_t kDesc,
 const void *keys,
 const cudnnSeqDataDescriptor_t vDesc,
 const void *values,
 const cudnnSeqDataDescriptor_t doDesc,
 const void *dout,
 size_t weightSizeInBytes,
 const void *weights,
 void *dweights,
 size_t workSpaceSizeInBytes,
 void *workSpace,
 size_t reserveSpaceSizeInBytes,
 void *reserveSpace);
```

This function computes exact, first-order derivatives of the multi-head attention block with respect to its trainable parameters: projection weights and projection biases. If $\mathbf{y}=F(\mathbf{w})$ is a vector-valued function that represents the multi-head attention layer and it takes some vector $x \in \mathbb{R}^n$ of flatten weights or biases as an input (with all other parameters and inputs fixed), and outputs vector $y \in \mathbb{R}^m$, then **cudnnMultiHeadAttnBackwardWeights()** computes the result of $\left(\partial y_i / \partial x_j\right)^T \delta_{\text{out}}$ where $\delta_{\text{out}}$ is the $\mathrm{m} \times 1$ gradient of the loss function with respect to multi-head attention outputs. The $\delta_{\text{out}}$ gradient is back propagated through prior layers of the deep learning

model. $\partial y_i / x_j$ is the $m \times n$ Jacobian matrix of $F(\mathbf{w})$. The $\delta_{out}$ input is supplied via the **dout** argument.

All gradient results with respect to weights and biases are written to the **dweights** buffer. The size and the organization of the **dweights** buffer is the same as the **weights** buffer that holds multi-head attention weights and biases. The cuDNN **multiHeadAttention** sample code demonstrates how to access those weights.

Gradient of the loss function with respect to weights or biases is typically computed over multiple batches. In such a case, partial results computed for each batch should be summed together. The **addGrad** argument specifies if the gradients from the current batch should be added to previously computed results or the **dweights** buffer should be overwritten with the new results.

The **cudnnMultiHeadAttnBackwardWeights()** function should be invoked after **cudnnMultiHeadAttnBackwardData()**. The **queries**, **keys**, **values**, **weights**, and **reserveSpace** arguments should be the same as in **cudnnMultiHeadAttnForward()** and **cudnnMultiHeadAttnBackwardData()** calls. The **dout** argument should be the same as in **cudnnMultiHeadAttnBackwardData()**.

### Parameters

**handle**

*Input*. The current cuDNN context handle.

**attnDesc**

*Input*. A previously initialized attention descriptor.

**addGrad**

*Input*. Weight gradient output mode.

**qDesc**

*Input*. Descriptor for the **query** sequence data.

**queries**

*Input*. Pointer to **queries** sequence data in the device memory.

**kDesc**

*Input*. Descriptor for the **keys** sequence data.

**keys**

*Input*. Pointer to **keys** sequence data in the device memory.

**vDesc**

*Input*. Descriptor for the **values** sequence data.

**values**

*Input*. Pointer to **values** sequence data in the device memory.

**doDesc**

*Input*. Descriptor for the $\delta_{out}$ gradients (vectors of partial derivatives of the loss function with respect to the multi-head attention outputs).

**dout**

*Input*. Pointer to $\delta_{out}$ gradient data in the device memory.

**weightSizeInBytes**

*Input*. Size of the **weights** and **dweights** buffers in bytes.

**weights**

*Input*. Address of the **weight** buffer in the device memory.

**dweights**

*Output*. Address of the weight gradient buffer in the device memory.

**workSpaceSizeInBytes**

*Input*. Size of the work-space buffer in bytes used for temporary API storage.

**workSpace**

*Input/Output*. Address of the work-space buffer in the device memory.

**reserveSpaceSizeInBytes**

*Input*. Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls.

**reserveSpace**

*Input/Output*. Address to the reserve-space buffer in the device memory.

### Returns

**CUDNN_STATUS_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN_STATUS_BAD_PARAM**

An invalid or incompatible input argument was encountered.

**CUDNN_STATUS_EXECUTION_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN_STATUS_INTERNAL_ERROR**

An inconsistent internal state was encountered.

**CUDNN_STATUS_NOT_SUPPORTED**

A requested option or a combination of input arguments is not supported.

# 4.152. cudnnMultiHeadAttnForward

```
cudnnStatus_t cudnnMultiHeadAttnForward(
 cudnnHandle_t handle,
 const cudnnAttnDescriptor_t attnDesc,
 int currIdx,
 const int loWinIdx[],
 const int hiWinIdx[],
 const int devSeqLengthsQO[],
 const int devSeqLengthsKV[],
 const cudnnSeqDataDescriptor_t qDesc,
 const void *queries,
 const void *residuals,
 const cudnnSeqDataDescriptor_t kDesc,
 const void *keys,
 const cudnnSeqDataDescriptor_t vDesc,
 const void *values,
 const cudnnSeqDataDescriptor_t oDesc,
       void *out,
 size_t weightSizeInBytes,
 const void *weights,
 size_t workSpaceSizeInBytes,
 void *workSpace,
 size_t reserveSpaceSizeInBytes,
 void *reserveSpace);
```

The **cudnnMultiHeadAttnForward()** function computes the forward responses of the multi-head attention layer. When **reserveSpaceSizeInBytes=0** and **reserveSpace=NULL**, the function operates in the inference mode in which backward (gradient) functions are not invoked, otherwise, the training mode is assumed. In the training mode, the reserve space is used to pass intermediate results from **cudnnMultiHeadAttnForward()** to **cudnnMultiHeadAttnBackwardData()** and from **cudnnMultiHeadAttnBackwardData()** to **cudnnMultiHeadAttnBackwardWeights()**.

In the inference mode, the **currIdx** specifies the time-step or sequence index of the embedding vectors to be processed. In this mode, the user can perform one iteration for time-step zero (**currIdx=0**), then update **Q, K, V** vectors and the attention window, and execute the next step (**currIdx=1**). The iterative process can be repeated for all time-steps.

When all **Q** time-steps are available (for example, in the training mode or in the inference mode on the encoder side in self-attention),the user can assign a negative value to **currIdx** and the **cudnnMultiHeadAttnForward()** API will automatically sweep through all **Q** time-steps.

The **loWinIdx[]** and **hiWinIdx[]** host arrays specify the attention window size for each **Q** time-step. In a typical self-attention case, the user must include all previously visited embedding vectors but not the current or future vectors. In this situation, the user should set:

```
currIdx=0: loWinIdx[0]=0; hiWinIdx[0]=0;  // initial time-step, no attention
 window
currIdx=1: loWinIdx[1]=0; hiWinIdx[1]=1;  // attention window spans one vector
currIdx=2: loWinIdx[2]=0; hiWinIdx[2]=2;  // attention window spans two vectors
(...)
```

When **currIdx** is negative in **cudnnMultiHeadAttnForward()**, the **loWinIdx[]** and **hiWinIdx[]** arrays must be fully initialized for all time-steps. When **cudnnMultiHeadAttnForward()** is invoked with **currIdx=0**, **currIdx=1**, **currIdx=2**, etc., then the user can update **loWinIdx[currIdx]** and **hiWinIdx[currIdx]** elements only before invoking the forward response function. All other elements in the **loWinIdx[]** and **hiWinIdx[]** arrays will not be accessed. Any adaptive attention window scheme can be implemented that way.

Use the following settings when the attention window should be the maximum size, for example, in cross-attention:

```
currIdx=0: loWinIdx[0]=0; hiWinIdx[0]=maxSeqLenK;
currIdx=1: loWinIdx[1]=0; hiWinIdx[1]=maxSeqLenK;
currIdx=2: loWinIdx[2]=0; hiWinIdx[2]=maxSeqLenK;
(...)
```

The **maxSeqLenK** value above should be equal to or larger than **dimA[CUDNN_SEQDATA_TIME_DIM]** in the **kDesc** descriptor. A good choice is to use **maxSeqLenK=INT_MAX** from **limits.h**.

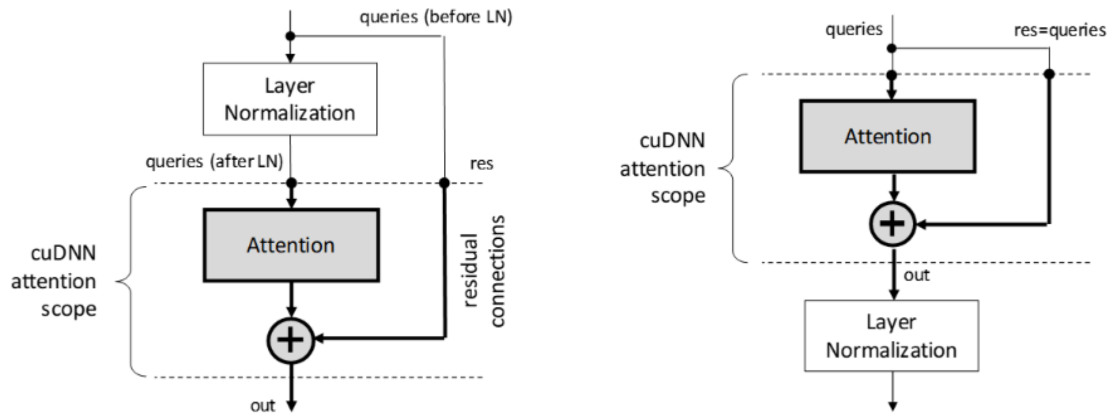
The actual length of any **K** sequence defined in **seqLengthArray[]** in **cudnnSetSeqDataDescriptor()** can be shorter than **maxSeqLenK**. The effective attention window span is computed based on **seqLengthArray[]** stored in the **K** sequence descriptor and indices held in **loWinIdx[]** and **hiWinIdx[]** arrays.

**devSeqLengthsQO[]** and **devSeqLengthsKV[]** are pointers to device (not host) arrays with **Q, O**, and **K, V** sequence lengths. Note that the same information is also passed in the corresponding descriptors of type **cudnnSeqDataDescriptor_t** on the host side. The need for extra device arrays comes from the asynchronous nature of cuDNN calls and limited size of the constant memory dedicated to GPU kernel arguments. When the **cudnnMultiHeadAttnForward()** API returns, the sequence length arrays stored in the descriptors can be immediately modified for the next iteration. However, the GPU kernels launched by the forward call may not have started at this point. For this reason, copies of sequence arrays are needed on the device side to be accessed directly by GPU kernels. Those copies cannot be created inside the **cudnnMultiHeadAttnForward()** function for very large **K, V** inputs without the device memory allocation and CUDA stream synchronization.

To reduce the **cudnnMultiHeadAttnForward()** API overhead, **devSeqLengthsQO[]** and **devSeqLengthsKV[]** device arrays are not validated to contain the same settings as **seqLengthArray[]** in the sequence data descriptors.

Sequence lengths in the **kDesc** and **vDesc** descriptors should be the same. Similarly, sequence lengths in the **qDesc** and **oDesc** descriptors should match. The user can define six different data layouts in the **qDesc**, **kDesc**, **vDesc** and **oDesc** descriptors. See the **cudnnSetSeqDataDescriptor()** function for the discussion of those layouts. All multi-head attention API calls require that the same layout is used in all sequence data descriptors.

In the transformer model, the multi-head attention block is tightly coupled with the layer normalization and residual connections. **cudnnMultiHeadAttnForward()** does not encompass the layer normalization but it can be used to handle residual connections as depicted in the following figure.




Queries and residuals share the same **`qDesc`** descriptor in **`cudnnMultiHeadAttnForward()`**. When residual connections are disabled, the residuals pointer should be **`NULL`**. When residual connections are enabled, the vector length in **`qDesc`** should match the vector length specified in the **`oDesc`** descriptor, so that a vector addition is feasible.

The **`queries`**, **`keys`**, and **`values`** pointers are not allowed to be **`NULL`**, even when **K** and **V** are the same inputs or **Q, K, V** are the same inputs.

## Parameters

**`handle`**

*Input*. The current cuDNN context handle.

**`attnDesc`**

*Input*. A previously initialized attention descriptor.

**`currIdx`**

*Input*. Time-step in queries to process. When the **`currIdx`** argument is negative, all **Q** time-steps are processed. When **`currIdx`** is zero or positive, the forward response is computed for the selected time-step only. The latter input can be used in inference mode only, to process one time-step while updating the next attention window and **Q, R, K, V** inputs in-between calls.

**`loWinIdx[], hiWinIdx[]`**

*Input*. Two host integer arrays specifying the start and end indices of the attention window for each **Q** time-step. The start index in **K, V** sets is inclusive, and the end index is exclusive.

**`devSeqLengthsQO[]`**

*Input*. Device array specifying sequence lengths of query, residual, and output sequence data.

**`devSeqLengthsKV[]`**

*Input*. Device array specifying sequence lengths of key and value input data.

**`qDesc`**

*Input*. Descriptor for the query and residual sequence data.

**`queries`**

*Input*. Pointer to queries data in the device memory.

**`residuals`**

*Input*. Pointer to residual data in device memory. Set this argument to **`NULL`** if no residual connections are required.

**`kDesc`**

*Input*. Descriptor for the **`keys`** sequence data.

**`keys`**

*Input*. Pointer to **`keys`** data in device memory.

**`vDesc`**

*Input*. Descriptor for the **`values`** sequence data.

**`values`**

*Input*. Pointer to **`values`** data in device memory.

**`oDesc`**

*Input*. Descriptor for the multi-head attention output sequence data.

**`out`**

*Output*. Pointer to device memory where the output response should be written.

**`weightSizeInBytes`**

*Input*. Size of the weight buffer in bytes where all multi-head attention trainable parameters are stored.

**`weights`**

*Input*. Pointer to the weight buffer in device memory.

**`workSpaceSizeInBytes`**

*Input*. Size of the work-space buffer in bytes used for temporary API storage.

**`workSpace`**

*Input/Output*. Pointer to the work-space buffer in device memory.

**`reserveSpaceSizeInBytes`**

*Input*. Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls. This parameter should be zero in the inference mode and non-zero in the training mode.

**reserveSpace**

*Input/Output*. Pointer to the reserve-space buffer in device memory. This argument should be **NULL** in inference mode and **non-NULL** in the training mode.

### Returns

**CUDNN_STATUS_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN_STATUS_BAD_PARAM**

An invalid or incompatible input argument was encountered. Some examples include:

- a required input pointer was **NULL**
- **currIdx** was out of bound
- the descriptor value for **attention**, **query**, **key**, **value**, and **output** were incompatible with one another

**CUDNN_STATUS_EXECUTION_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN_STATUS_INTERNAL_ERROR**

An inconsistent internal state was encountered.

**CUDNN_STATUS_NOT_SUPPORTED**

A requested option or a combination of input arguments is not supported.

**CUDNN_STATUS_ALLOC_FAILED**

Insufficient amount of shared memory to launch a GPU kernel.

# 4.153. cudnnOpTensor

```
cudnnStatus_t cudnnOpTensor(
    cudnnHandle_t                     handle,
    const cudnnOpTensorDescriptor_t   opTensorDesc,
    const void                       *alpha1,
    const cudnnTensorDescriptor_t     aDesc,
    const void                       *A,
    const void                       *alpha2,
    const cudnnTensorDescriptor_t     bDesc,
    const void                       *B,
    const void                       *beta,
    const cudnnTensorDescriptor_t     cDesc,
    void                             *C)
```

This function implements the equation **C = op(alpha1[0] * A, alpha2[0] * B) + beta[0] * C**, given the tensors **A**, **B**, and **C** and the scaling factors **alpha1**, **alpha2**,

and **beta**. The **op** to use is indicated by the descriptor cudnnOpTensorDescriptor_t, meaning, the type of **opTensorDesc**. Currently-supported ops are listed by the cudnnOpTensorOp_t enum.

The following restrictions on the input and destination tensors apply:

- Each dimension of the input tensor **A** must match the corresponding dimension of the destination tensor **C**, and each dimension of the input tensor **B** must match the corresponding dimension of the destination tensor **C** or must be equal to 1. In the latter case, the same value from the input tensor **B** for those dimensions will be used to blend into the **C** tensor.
- The data types of the input Tensors **A** and **B**, and the destination Tensor C, must satisfy Table 25.

## Table 25 Supported Datatypes

| **opTensorCompType** in **opTensorDesc** | **A** | **B** | **C** (destination) |
|---|---|---|---|
| FLOAT | FLOAT | FLOAT | FLOAT |
| FLOAT | INT8 | INT8 | FLOAT |
| FLOAT | HALF | HALF | FLOAT |
| DOUBLE | DOUBLE | DOUBLE | DOUBLE |
| FLOAT | FLOAT | FLOAT | HALF |
| FLOAT | HALF | HALF | HALF |
| FLOAT | INT8 | INT8 | INT8 |
| FLOAT | FLOAT | FLOAT | INT8 |

All tensor formats up to dimension five (5) are supported. This routine does not support tensor formats beyond these dimensions.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**opTensorDesc**

*Input*. Handle to a previously initialized op tensor descriptor.

**alpha1, alpha2, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**aDesc, bDesc, cDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**A, B**

*Input*. Pointer to data of the tensors described by the **aDesc** and **bDesc** descriptors, respectively.

**C**

*Input/Output*. Pointer to data of the tensor described by the **cDesc** descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function executed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimensions of the bias tensor and the output tensor dimensions are above 5.
- **opTensorCompType** is not set as stated above.

**CUDNN_STATUS_BAD_PARAM**

The data type of the destination tensor **C** is unrecognized, or the restrictions on the input and destination tensors, stated above, are not met.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.154. cudnnPoolingBackward

```
cudnnStatus_t cudnnPoolingBackward(
    cudnnHandle_t                       handle,
    const cudnnPoolingDescriptor_t      poolingDesc,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       yDesc,
    const void                         *y,
    const cudnnTensorDescriptor_t       dyDesc,
    const void                         *dy,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *xData,
    const void                         *beta,
    const cudnnTensorDescriptor_t       dxDesc,
    void                               *dx)
```

This function computes the gradient of a pooling operation.

As of cuDNN version 6.0, a deterministic algorithm is implemented for max backwards pooling. This algorithm can be chosen via the pooling mode enum of **poolingDesc**. The

deterministic algorithm has been measured to be up to 50% slower than the legacy max backwards pooling algorithm, or up to 20% faster, depending upon the use case.

All tensor formats are supported, best performance is expected when using **`HW-packed`** tensors. Only 2 and 3 spatial dimensions are allowed

### Parameters

**`handle`**

*Input*. Handle to a previously created cuDNN context.

**`poolingDesc`**

*Input*. Handle to the previously initialized pooling descriptor.

**`alpha, beta`**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**`yDesc`**

*Input*. Handle to the previously initialized input tensor descriptor.

**`y`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`yDesc`**.

**`dyDesc`**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**`dy`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`dyData`**.

**`xDesc`**

*Input*. Handle to the previously initialized output tensor descriptor.

**`x`**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **`xDesc`**.

**`dxDesc`**

*Input*. Handle to the previously initialized output differential tensor descriptor.

**`dx`**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **`dxDesc`**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions **n**, **c**, **h**, **w** of the **yDesc** and **dyDesc** tensors differ.
- The strides **nStride**, **cStride**, **hStride**, **wStride** of the **yDesc** and **dyDesc** tensors differ.
- The dimensions **n**, **c**, **h**, **w** of the **dxDesc** and **dxDesc** tensors differ.
- The strides **nStride**, **cStride**, **hStride**, **wStride** of the **xDesc** and **dxDesc** tensors differ.
- The datatype of the four tensors differ.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The **wStride** of input tensor or output tensor is not 1.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.155. cudnnPoolingForward

```
cudnnStatus_t cudnnPoolingForward(
    cudnnHandle_t                    handle,
    const cudnnPoolingDescriptor_t   poolingDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This function computes pooling of input values (meaning, the maximum or average of several adjacent values) to produce an output with smaller height and/or width.

- All tensor formats are supported, best performance is expected when using **HW-packed** tensors. Only 2 and 3 spatial dimensions are allowed.
- The dimensions of the output tensor **yDesc** can be smaller or bigger than the dimensions advised by the routine **cudnnGetPooling2dForwardOutputDim** or **cudnnGetPoolingNdForwardOutputDim**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**poolingDesc**

*Input*. Handle to a previously initialized pooling descriptor.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor. Must be of type **FLOAT**, **DOUBLE**, **HALF** or **INT8**. For more information, see cudnnDataType_t.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor. Must be of type **FLOAT**, **DOUBLE**, **HALF** or **INT8**. For more information, see cudnnDataType_t.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions **n**, **c** of the input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differs.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.156. cudnnQueryRuntimeError

```
cudnnStatus_t cudnnQueryRuntimeError(
    cudnnHandle_t            handle,
    cudnnStatus_t           *rstatus,
    cudnnErrQueryMode_t      mode,
    cudnnRuntimeTag_t       *tag)
```

cuDNN library functions perform extensive input argument checking before launching GPU kernels. The last step is to verify that the GPU kernel actually started. When a kernel fails to start, **CUDNN_STATUS_EXECUTION_FAILED** is returned by the corresponding API call. Typically, after a GPU kernel starts, no runtime checks are performed by the kernel itself - numerical results are simply written to output buffers.

When the **CUDNN_BATCHNORM_SPATIAL_PERSISTENT** mode is selected in **cudnnBatchNormalizationForwardTraining** or **cudnnBatchNormalizationBackward**, the algorithm may encounter numerical overflows where **CUDNN_BATCHNORM_SPATIAL** performs just fine albeit at a slower speed. The user can invoke **cudnnQueryRuntimeError** to make sure numerical overflows did not occur during the kernel execution. Those issues are reported by the kernel that performs computations.

**cudnnQueryRuntimeError** can be used in polling and blocking software control flows. There are two polling modes (**CUDNN_ERRQUERY_RAWCODE** and **CUDNN_ERRQUERY_NONBLOCKING**) and one blocking mode **CUDNN_ERRQUERY_BLOCKING**.

**CUDNN_ERRQUERY_RAWCODE** reads the error storage location regardless of the kernel completion status. The kernel might not even started and the error storage (allocated per cuDNN handle) might be used by an earlier call.

**CUDNN_ERRQUERY_NONBLOCKING** checks if all tasks in the user stream completed. The **cudnnQueryRuntimeError** function will return immediately and report **CUDNN_STATUS_RUNTIME_IN_PROGRESS** in **rstatus** if some tasks in the user stream are pending. Otherwise, the function will copy the remote kernel error code to **rstatus**.

In the blocking mode (**CUDNN_ERRQUERY_BLOCKING**), the function waits for all tasks to drain in the user stream before reporting the remote kernel error code. The blocking flavor can be further adjusted by calling **cudaSetDeviceFlags** with the **cudaDeviceScheduleSpin**, **cudaDeviceScheduleYield**, or **cudaDeviceScheduleBlockingSync** flag.

**CUDNN_ERRQUERY_NONBLOCKING** and **CUDNN_ERRQUERY_BLOCKING** modes should not be used when the user stream is changed in the cuDNN handle, meaning, **cudnnSetStream** is invoked between functions that report runtime kernel errors and the **cudnnQueryRuntimeError** function.

The remote error status reported in rstatus can be set to: **CUDNN_STATUS_SUCCESS**, **CUDNN_STATUS_RUNTIME_IN_PROGRESS**, or **CUDNN_STATUS_RUNTIME_FP_OVERFLOW**. The remote kernel error is automatically cleared by **cudnnQueryRuntimeError**.

The **cudnnQueryRuntimeError** function should be used in conjunction with **cudnnBatchNormalizationForwardTraining** and **cudnnBatchNormalizationBackward** when the **cudnnBatchNormMode_t** argument is **CUDNN_BATCHNORM_SPATIAL_PERSISTENT**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rstatus**

*Output*. Pointer to the user's error code storage.

**mode**

*Input*. Remote error query mode.

**tag**

*Input/Output*. Currently, this argument should be **NULL**.

### Returns

**CUDNN_STATUS_SUCCESS**

No errors detected (**rstatus** holds a valid value).

**CUDNN_STATUS_BAD_PARAM**

Invalid input argument.

**CUDNN_STATUS_INTERNAL_ERROR**

A stream blocking synchronization or a non-blocking stream query failed.

**CUDNN_STATUS_MAPPING_ERROR**

Device cannot access zero-copy memory to report kernel errors.

## 4.157. cudnnReduceTensor

```
cudnnStatus_t cudnnReduceTensor(
    cudnnHandle_t                         handle,
    const cudnnReduceTensorDescriptor_t   reduceTensorDesc,
    void                                 *indices,
    size_t                                indicesSizeInBytes,
    void                                 *workspace,
    size_t                                workspaceSizeInBytes,
    const void                           *alpha,
    const cudnnTensorDescriptor_t         aDesc,
    const void                           *A,
```

```
    const void                          *beta,
    const cudnnTensorDescriptor_t        cDesc,
    void                                *C)
```

This function reduces tensor **A** by implementing the equation **C = alpha * reduce op ( A ) + beta * C**, given tensors **A** and **C** and scaling factors **alpha** and **beta**. The reduction op to use is indicated by the descriptor **reduceTensorDesc**. Currently-supported ops are listed by the **cudnnReduceTensorOp_t** enum.

Each dimension of the output tensor **C** must match the corresponding dimension of the input tensor **A** or must be equal to 1. The dimensions equal to 1 indicate the dimensions of **A** to be reduced.

The implementation will generate indices for the min and max ops only, as indicated by the **cudnnReduceTensorIndices_t** enum of the **reduceTensorDesc**. Requesting indices for the other reduction ops results in an error. The data type of the indices is indicated by the **cudnnIndicesType_t** enum; currently only the 32-bit (unsigned int) type is supported.

The indices returned by the implementation are not absolute indices but relative to the dimensions being reduced. The indices are also flattened, meaning, not coordinate tuples.

The data types of the tensors **A** and **C** must match if of type double. In this case, **alpha** and **beta** and the computation enum of **reduceTensorDesc** are all assumed to be of type double.

The **HALF** and **INT8** data types may be mixed with the **FLOAT** data types. In these cases, the computation enum of **reduceTensorDesc** is required to be of type **FLOAT**.

Up to dimension 8, all tensor formats are supported. Beyond those dimensions, this routine is not supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**reduceTensorDesc**

*Input*. Handle to a previously initialized reduce tensor descriptor.

**indices**

*Output*. Handle to a previously allocated space for writing indices.

**indicesSizeInBytes**

*Input*. Size of the above previously allocated space.

**workspace**

*Input*. Handle to a previously allocated space for the reduction implementation.

**workspaceSizeInBytes**

*Input*. Size of the above previously allocated space.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*resultValue + beta[0]*priorDstValue
```

For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

**aDesc, cDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**A**

*Input*. Pointer to data of the tensor described by the **aDesc** descriptor.

**C**

*Input/Output*. Pointer to data of the tensor described by the **cDesc** descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function executed successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimensions of the input tensor and the output tensor are above 8.
- **reduceTensorCompType** is not set as stated above.

**CUDNN_STATUS_BAD_PARAM**

The corresponding dimensions of the input and output tensors all match, or the conditions in the above paragraphs are unmet.

**CUDNN_INVALID_VALUE**

The allocations for the indices or workspace are insufficient.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.158. cudnnReorderFilterAndBias

```
cudnnStatus_t cudnnReorderFilterAndBias(
 cudnnHandle_t handle,
 const cudnnFilterDescriptor_t filterDesc,
 cudnnReorderType_t reorderType,
 const void *filterData,
```

```
void *reorderedFilterData,
int reorderBias,
const void *biasData,
void *reorderedBiasData);
```

This function **cudnnReorderFilterAndBias()** reorders the filter and bias values. It can be used to enhance the inference time by separating the reordering operation from convolution.

For example, convolutions in a neural network of multiple layers can require reordering of kernels at every layer, which can take up a significant fraction of the total inference time. Using this function, the reordering can be done one time on the filter and bias data followed by the convolution operations at the multiple layers, thereby enhancing the inference time.

### Parameters

**filterDesc**
*Input*. Descriptor for the kernel dataset.

**reorderType**
*Input*. Setting to either perform reordering or not. For more information, see cudnnReorderType_t.

**filterData**
*Input*. Pointer to the filter (kernel) data location in the device memory.

**reorderedFilterData**
*Input*. Pointer to the location in the device memory where the reordered filter data will be written to, by this function.

**reorderBias**
*Input*. If > 0, then reorders the bias data also. If <= 0 then does not perform reordering operation on the bias data.

**biasData**
*Input*. Pointer to the bias data location in the device memory.

**reorderedBiasData**
*Input*. Pointer to the location in the device memory where the reordered bias data will be written to, by this function.

### Returns

**CUDNN_STATUS_SUCCESS**
Reordering was successful.

**CUDNN_STATUS_EXECUTION_FAILED**
Either the reordering of the filter data or of the bias data failed.

## 4.159. cudnnRestoreAlgorithm

```
cudnnStatus_t cudnnRestoreAlgorithm(
    cudnnHandle_t                handle,
    void*                        algoSpace,
```

```
    size_t                          algoSpaceSizeInBytes,
    cudnnAlgorithmDescriptor_t  algoDesc)
```

This function reads algorithm metadata from the host memory space provided by the user in **algoSpace**, allowing the user to use the results of RNN finds from previous cuDNN sessions.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**algoDesc**

*Input*. A previously created algorithm descriptor.

**algoSpace**

*Input*. Pointer to the host memory to be read.

**algoSpaceSizeInBytes**

*Input*. Amount of host memory needed as workspace to be able to hold the metadata from the specified **algoDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The metadata is from a different cuDNN version.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions is met:

- One of the arguments is **NULL**.
- The metadata is corrupted.

## 4.160. cudnnRestoreDropoutDescriptor

```
cudnnStatus_t cudnnRestoreDropoutDescriptor(
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnHandle_t            handle,
    float                    dropout,
    void                    *states,
    size_t                   stateSizeInBytes,
    unsigned long long       seed)
```

This function restores a dropout descriptor to a previously saved-off state.

### Parameters

**dropoutDesc**

*Input/Output*. Previously created dropout descriptor.

**handle**

*Input*. Handle to a previously created cuDNN context.

**dropout**

*Input*. Probability with which the value from an input tensor is set to 0 when performing dropout.

**states**

*Input*. Pointer to GPU memory that holds random number generator states initialized by a prior call to **cudnnSetDropoutDescriptor**.

**stateSizeInBytes**

*Input*. Size in bytes of buffer holding random number generator **states**.

**seed**

*Input*. Seed used in prior call to **cudnnSetDropoutDescriptor** that initialized **states** buffer. Using a different seed from this has no effect. A change of seed, and subsequent update to random number generator states can be achieved by calling **cudnnSetDropoutDescriptor**.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_INVALID_VALUE**

States buffer size (as indicated in **stateSizeInBytes**) is too small.

# 4.161. cudnnRNNBackwardData

```
cudnnStatus_t cudnnRNNBackwardData(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       seqLength,
    const cudnnTensorDescriptor_t  *yDesc,
    const void                     *y,
    const cudnnTensorDescriptor_t  *dyDesc,
    const void                     *dy,
    const cudnnTensorDescriptor_t   dhyDesc,
    const void                     *dhy,
    const cudnnTensorDescriptor_t   dcyDesc,
    const void                     *dcy,
    const cudnnFilterDescriptor_t   wDesc,
    const void                     *w,
    const cudnnTensorDescriptor_t   hxDesc,
    const void                     *hx,
```

```
    const cudnnTensorDescriptor_t   cxDesc,
    const void                     *cx,
    const cudnnTensorDescriptor_t  *dxDesc,
    void                           *dx,
    const cudnnTensorDescriptor_t   dhxDesc,
    void                           *dhx,
    const cudnnTensorDescriptor_t   dcxDesc,
    void                           *dcx,
    void                           *workspace,
    size_t                          workSpaceSizeInBytes,
    const void                     *reserveSpace,
    size_t                          reserveSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **rnnDesc** with output gradients **dy**, **dhy**, and **dhc**, weights **w** and input gradients **dx**, **dhx**, and **dcx**. **workspace** is required for intermediate storage. The data in **reserveSpace** must have previously been generated by cudnnRNNForwardTraining. The same **reserveSpace** data must be used for future calls to cudnnRNNBackwardWeights if they execute on the same input data.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information , see cudnnHandle_t.

**rnnDesc**

*Input*. A previously initialized RNN descriptor. For more information, see cudnnRNNDescriptor_t.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in the **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). For more information, see cudnnTensorDescriptor_t. The second dimension of the tensor depends on the **direction** argument passed to the cudnnSetRNNDescriptor call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

**y**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**dyDesc**

*Input*. An array of fully packed tensor descriptors describing the gradient at the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dxDesc**.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **dyDesc**.

**dhyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dhy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dhyDesc**. If a **NULL** pointer is passed, the gradients at the final hidden state of the network will be initialized to zero.

**dcyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dcy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dcyDesc**. If a **NULL** pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN. For more information, see cudnnFilterDescriptor_t.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the second dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the second dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**dxDesc**

*Input*. An array of fully packed tensor descriptors describing the gradient at the input of each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**dx**

*Output*. Data pointer to GPU memory associated with the tensor descriptors in the array **dxDesc**.

**dhxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dhx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dhxDesc**. If a **NULL** pointer is passed, the gradient at the hidden input of the network will not be set.

**dcxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dcx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dcxDesc**. If a **NULL** pointer is passed, the gradient at the cell input of the network will not be set.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **dhxDesc**, **wDesc**, **hxDesc**, **cxDesc**, **dcxDesc**, **dhyDesc**, **dcyDesc** or one of the descriptors in **yDesc**, **dxdesc**, **dydesc** is invalid.

- The descriptors in one of **yDesc**, **dxDesc**, **dyDesc**, **dhxDesc**, **wDesc**, **hxDesc**, **cxDesc**, **dcxDesc**, **dhyDesc**, **dcyDesc** has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.162. cudnnRNNBackwardDataEx

```
cudnnStatus_t cudnnRNNBackwardDataEx(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const cudnnRNNDataDescriptor_t     yDesc,
    const void                         *y,
    const cudnnRNNDataDescriptor_t     dyDesc,
    const void                         *dy,
    const cudnnRNNDataDescriptor_t     dcDesc,
    const void                         *dcAttn,
    const cudnnTensorDescriptor_t      dhyDesc,
    const void                         *dhy,
    const cudnnTensorDescriptor_t      dcyDesc,
    const void                         *dcy,
    const cudnnFilterDescriptor_t      wDesc,
    const void                         *w,
    const cudnnTensorDescriptor_t      hxDesc,
    const void                         *hx,
    const cudnnTensorDescriptor_t      cxDesc,
    const void                         *cx,
    const cudnnRNNDataDescriptor_t     dxDesc,
    void                               *dx,
    const cudnnTensorDescriptor_t      dhxDesc,
    void                               *dhx,
    const cudnnTensorDescriptor_t      dcxDesc,
    void                               *dcx,
    const cudnnRNNDataDescriptor_t     dkDesc,
    void                               *dkeys,
    void                               *workSpace,
    size_t                             workSpaceSizeInBytes,
    void                               *reserveSpace,
    size_t                             reserveSpaceSizeInBytes)
```

This routine is the extended version of the function **cudnnRNNBackwardData**. This function **cudnnRNNBackwardDataEx** allows the user to use unpacked (padded) layout for input **y** and output **dx**.

In the unpacked layout, each sequence in the mini-batch is considered to be of fixed length, specified by **maxSeqLength** in its corresponding **RNNDataDescriptor**. Each fixed-length sequence, for example, the **nth** sequence in the mini-batch, is composed of a valid segment specified by the **seqLengthArray[n]** in its corresponding **RNNDataDescriptor**; and a padding segment to make the combined sequence length equal to **maxSeqLength**.

With the unpacked layout, both sequence major (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function **cudnnRNNBackwardData**, the sequences in the mini-batch need to be sorted in descending order according to length.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**yDesc**

*Input*. A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**y**

*Input*. Data pointer to the GPU memory associated with the RNN data descriptor **yDesc**. The vectors are expected to be laid out in memory according to the layout specified by **yDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported. Must contain the exact same data previously produced by **cudnnRNNForwardTrainingEx**.

**dyDesc**

*Input*. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, **vectorSize**, and **seqLengthArray** need to match the **yDesc** previously passed to **cudnnRNNForwardTrainingEx**.

**dy**

*Input*. Data pointer to the GPU memory associated with the RNN data descriptor **dyDesc**. The vectors are expected to be laid out in memory according to the layout specified by **dyDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**dhyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. Additionally:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the **batchSize** parameter in **xDesc**. The third dimension depends on whether the RNN mode is **CUDNN_LSTM** and whether LSTM projection is enabled. Additionally:

- If the RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the third dimension must match the **recProjSize** argument passed to **cudnnSetRNNProjectionLayers** call used to set **rnnDesc**.
- Otherwise, the third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**.

**dhy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dhyDesc**. If a **NULL** pointer is passed, the gradients at the final hidden state of the network will be initialized to zero.

**dcyDesc**

*Input*. A fully packed tensor descriptor describing the gradients at the final cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. Additionally:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**dcy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dcyDesc**. If a **NULL** pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero. Must contain the exact same data previously passed into **cudnnRNNForwardTrainingEx**, or be **NULL** if **NULL** was previously passed to **cudnnRNNForwardTrainingEx**.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero. Must contain the exact same data previously passed into **cudnnRNNForwardTrainingEx**, or be **NULL** if **NULL** was previously passed to **cudnnRNNForwardTrainingEx**.

**dxDesc**

*Input*. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, **vectorSize** and **seqLengthArray** need to match that of **xDesc** previously passed to **cudnnRNNForwardtrainingEx**.

**dx**

*Output*. Data pointer to the GPU memory associated with the RNN data descriptor **dxDesc**. The vectors are expected to be laid out in memory according to the layout specified by **dxDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**dhxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial hidden state of the RNN. The descriptor must be set exactly the same way as **dhyDesc**.

**dhx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dhxDesc**. If a **NULL** pointer is passed, the gradient at the hidden input of the network will not be set.

**dcxDesc**

*Input*. A fully packed tensor descriptor describing the gradient at the initial cell state of the RNN. The descriptor must be set exactly the same way as **dcyDesc**.

**dcx**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **dcxDesc**. If a **NULL** pointer is passed, the gradient at the cell input of the network will not be set.

**dkDesc**

Reserved. User may pass in **NULL**.

**dkeys**

Reserved. User may pass in **NULL**.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- Variable sequence length input is passed in while **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used.
- **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used on pre-Pascal devices.
- Double input/output is used for **CUDNN_RNN_ALGO_PERSIST_STATIC**.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **yDesc**, **dxdesc**, **dydesc**, **dhxDesc**, **wDesc**, **hxDesc**, **cxDesc**, **dcxDesc**, **dhyDesc**, **dcyDesc** is invalid or has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.

- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.163. cudnnRNNBackwardWeights

```
cudnnStatus_t cudnnRNNBackwardWeights(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       seqLength,
    const cudnnTensorDescriptor_t  *xDesc,
    const void                     *x,
    const cudnnTensorDescriptor_t   hxDesc,
    const void                     *hx,
    const cudnnTensorDescriptor_t  *yDesc,
    const void                     *y,
    const void                     *workspace,
    size_t                          workSpaceSizeInBytes,
    const cudnnFilterDescriptor_t   dwDesc,
    void                           *dw,
    const void                     *reserveSpace,
    size_t                          reserveSpaceSizeInBytes)
```

This routine accumulates weight gradients **dw** from the recurrent neural network described by **rnnDesc** with inputs **x**, **hx** and outputs **y**. The mode of operation in this case is additive, the weight gradients calculated will be added to those already existing in **dw**. **workspace** is required for intermediate storage. The data in **reserveSpace** must have previously been generated by **cudnnRNNBackwardData**.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

**y**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**dw**

*Input/Output*. Data pointer to GPU memory associated with the filter descriptor **dwDesc**.

**reserveSpace**

*Input*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **hxDesc**, **dwDesc** or one of the descriptors in **xDesc**, **yDesc** is invalid.
- The descriptors in one of **xDesc**, **hxDesc**, **yDesc**, **dwDesc** has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

## 4.164. cudnnRNNBackwardWeightsEx

```
cudnnStatus_t cudnnRNNBackwardWeightsEx(
    cudnnHandle_t                     handle,
    const cudnnRNNDescriptor_t        rnnDesc,
    const cudnnRNNDataDescriptor_t    xDesc,
    const void                        *x,
```

```
    const cudnnTensorDescriptor_t    hxDesc,
    const void                       *hx,
    const cudnnRNNDataDescriptor_t   yDesc,
    const void                       *y,
    void                             *workSpace,
    size_t                           workSpaceSizeInBytes,
    const cudnnFilterDescriptor_t    dwDesc,
    void                             *dw,
    void                             *reserveSpace,
    size_t                           reserveSpaceSizeInBytes)
```

This routine is the extended version of the function **cudnnRNNBackwardWeights**. This function **cudnnRNNBackwardWeightsEx** allows the user to use unpacked (padded) layout for input **x** and output **dw**.

In the unpacked layout, each sequence in the mini-batch is considered to be of fixed length, specified by **maxSeqLength** in its corresponding **RNNDataDescriptor**. Each fixed-length sequence, for example, the **nth** sequence in the mini-batch, is composed of a valid segment specified by the **seqLengthArray[n]** in its corresponding **RNNDataDescriptor**; and a padding segment to make the combined sequence length equal to **maxSeqLength**.

With the unpacked layout, both sequence major (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function **cudnnRNNBackwardWeights**, the sequences in the mini-batch need to be sorted in descending order according to length.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**xDesc**

*Input*. A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**. Must contain the exact same data previously passed into **cudnnRNNForwardTrainingEx**.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero. Must contain the exact same data previously passed into **cudnnRNNForwardTrainingEx**, or be **NULL** if **NULL** was previously passed to **cudnnRNNForwardTrainingEx**.

**yDesc**

*Input*. A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into **cudnnRNNForwardTrainingEx**.

**y**

*Input*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**. Must contain the exact same data previously produced by **cudnnRNNForwardTrainingEx**.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**dwDesc**

*Input*. Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**dw**

*Input/Output*. Data pointer to GPU memory associated with the filter descriptor **dwDesc**.

**reserveSpace**

*Input*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **xDesc**, **yDesc**, **hxDesc**, **dwDesc** is invalid, or has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

## 4.165. cudnnRNNForwardInference

```
cudnnStatus_t cudnnRNNForwardInference(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const int                          seqLength,
    const cudnnTensorDescriptor_t     *xDesc,
    const void                        *x,
    const cudnnTensorDescriptor_t      hxDesc,
    const void                        *hx,
    const cudnnTensorDescriptor_t      cxDesc,
    const void                        *cx,
    const cudnnFilterDescriptor_t      wDesc,
    const void                        *w,
    const cudnnTensorDescriptor_t      *yDesc,
    void                              *y,
    const cudnnTensorDescriptor_t      hyDesc,
    void                              *hy,
    const cudnnTensorDescriptor_t      cyDesc,
    void                              *cy,
    void                              *workspace,
    size_t                             workSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **rnnDesc** with inputs **x**, **hx**, and **cx**, weights **w** and outputs **y**, **hy**, and **cy**. **workspace** is required for intermediate storage. This function does not store intermediate data required for training; **cudnnRNNForwardTraining** should be used for that purpose.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of **seqLength** fully packed tensor descriptors. Each descriptor in the array should have three dimensions that describe the input data format to one recurrent iteration (one descriptor per RNN time-step). The first dimension (batch size) of the tensors may decrease from iteration **n** to iteration **n+1** but may not increase. Each tensor descriptor must have the same second dimension (RNN input vector length, **inputSize**). The third dimension of each tensor should be 1. Input data are expected to be arranged in the column-major order so strides in **xDesc** should be set as follows:

```
strideA[0]=inputSize, strideA[1]=1, strideA[2]=1
```

**x**

*Input*. Data pointer to GPU memory associated with the array of tensor descriptors **xDesc**. The input vectors are expected to be packed contiguously with the first vector of iteration (time-step) **n+1** following directly from the last vector of iteration **n**. In other words, input vectors for all RNN time-steps should be packed in the contiguous block of GPU memory with no gaps between the vectors.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **xDesc**.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**. The data are expected to be packed contiguously with the first element of iteration **n+1** following directly from the last element of iteration **n**.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.

- At least one of the descriptors **hxDesc**, **cxDesc**, **wDesc**, **hyDesc**, **cyDesc** or one of the descriptors in **xDesc**, **yDesc** is invalid.
- The descriptors in one of **xDesc**, **hxDesc**, **cxDesc**, **wDesc**, **yDesc**, **hyDesc**, **cyDesc** have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

## 4.166. cudnnRNNForwardInferenceEx

```
cudnnStatus_t cudnnRNNForwardInferenceEx(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const cudnnRNNDataDescriptor_t  xDesc,
    const void                      *x,
    const cudnnTensorDescriptor_t   hxDesc,
    const void                      *hx,
    const cudnnTensorDescriptor_t   cxDesc,
    const void                      *cx,
    const cudnnFilterDescriptor_t   wDesc,
    const void                      *w,
    const cudnnRNNDataDescriptor_t  yDesc,
    void                            *y,
    const cudnnTensorDescriptor_t   hyDesc,
    void                            *hy,
    const cudnnTensorDescriptor_t   cyDesc,
    void                            *cy,
    const cudnnRNNDataDescriptor_t  kDesc,
    const void                      *keys,
    const cudnnRNNDataDescriptor_t  cDesc,
    void                            *cAttn,
    const cudnnRNNDataDescriptor_t  iDesc,
    void                            *iAttn,
    const cudnnRNNDataDescriptor_t  qDesc,
    void                            *queries,
    void                            *workSpace,
    size_t                          workSpaceSizeInBytes)
```

This routine is the extended version of the **cudnnRNNForwardInference** function. The **cudnnRNNForwardTrainingEx** allows the user to use unpacked (padded) layout for input **x** and output **y**. In the unpacked layout, each sequence in the mini-batch is considered to be of fixed length, specified by **maxSeqLength** in its corresponding **RNNDataDescriptor**. Each fixed-length sequence, for example, the **nth** sequence in the mini-batch, is composed of a valid segment, specified by the **seqLengthArray[n]** in its corresponding **RNNDataDescriptor**, and a padding segment to make the combined sequence length equal to **maxSeqLength**.

With unpacked layout, both sequence major (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported.

However, similar to the non-extended function **cudnnRNNForwardInference**, the sequences in the mini-batch need to be sorted in descending order according to length.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**xDesc**

*Input*. A previously initialized RNN Data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, and **seqLengthArray** need to match that of **yDesc**.

**x**

*Input*. Data pointer to the GPU memory associated with the RNN data descriptor **xDesc**. The vectors are expected to be laid out in memory according to the layout specified by **xDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the **batchSize** parameter described in **xDesc**. The third dimension depends on whether RNN mode is **CUDNN_LSTM** and whether LSTM projection is enabled. Specifically:

- If RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the third dimension must match the **recProjSize** argument passed to **cudnnSetRNNProjectionLayers** call used to set **rnnDesc**.
- Otherwise, the third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the **batchSize** parameter in **xDesc**.
The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength** , **batchSize**, and **seqLengthArray** must match that of **dyDesc** and **dxDesc**. The parameter **vectorSize** depends on whether RNN mode is **CUDNN_LSTM** and whether LSTM projection is enabled and whether the network is bidirectional. Specifically:

- For unidirectional network, if the RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the parameter **vectorSize** must match the **recProjSize** argument passed to **cudnnSetRNNProjectionLayers** call used to set **rnnDesc**. If the network is bidirectional, then multiply the value by 2.
- Otherwise, for unidirectional network, the parameter **vectorSize** must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. If the network is bidirectional, then multiply the value by 2.

**y**

*Output*. Data pointer to the GPU memory associated with the RNN data descriptor **yDesc**. The vectors are expected to be laid out in memory according to the layout specified by **yDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The descriptor must be set exactly the same way as **hxDesc**.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The descriptor must be set exactly the same way as **cxDesc**.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**kDesc**

Reserved. User may pass in **NULL**.

**keys**

Reserved. User may pass in **NULL**.

**cDesc**

Reserved. User may pass in **NULL**.

**cAttn**

Reserved. User may pass in **NULL**.

**iDesc**

Reserved. User may pass in **NULL**.

**iAttn**

Reserved. User may pass in **NULL**.

**qDesc**

Reserved. User may pass in **NULL**.

**queries**

Reserved. User may pass in **NULL**.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- Variable sequence length input is passed in while **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used.
- **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used on pre-Pascal devices.
- Double input/output is used for **CUDNN_RNN_ALGO_PERSIST_STATIC**.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors in **xDesc**, **yDesc**, **hxDesc**, **cxDesc**, **wDesc**, **hyDesc**, **cyDesc** is invalid, or have incorrect strides or dimensions.
- **reserveSpaceSizeInBytes** is too small.
- **workSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

## 4.167. cudnnRNNForwardTraining

```
cudnnStatus_t cudnnRNNForwardTraining(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       seqLength,
    const cudnnTensorDescriptor_t  *xDesc,
    const void                     *x,
    const cudnnTensorDescriptor_t   hxDesc,
    const void                     *hx,
    const cudnnTensorDescriptor_t   cxDesc,
    const void                     *cx,
    const cudnnFilterDescriptor_t   wDesc,
    const void                     *w,
    const cudnnTensorDescriptor_t  *yDesc,
    void                           *y,
    const cudnnTensorDescriptor_t   hyDesc,
    void                           *hy,
    const cudnnTensorDescriptor_t   cyDesc,
    void                           *cy,
    void                           *workspace,
```

```
    size_t                              workSpaceSizeInBytes,
    void                               *reserveSpace,
    size_t                              reserveSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **rnnDesc** with inputs **x**, **hx**, and **cx**, weights **w** and outputs **y**, **hy**, and **cy**. **workspace** is required for intermediate storage. **reserveSpace** stores data required for training. The same **reserveSpace** data must be used for future calls to **cudnnRNNBackwardData** and **cudnnRNNBackwardWeights** if these execute on the same input data.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**seqLength**

*Input*. Number of iterations to unroll over. The value of this **seqLength** must not exceed the value that was used in **cudnnGetRNNWorkspaceSize()** function for querying the workspace size required to execute the RNN.

**xDesc**

*Input*. An array of **seqLength** fully packed tensor descriptors. Each descriptor in the array should have three dimensions that describe the input data format to one recurrent iteration (one descriptor per RNN time-step). The first dimension (batch size) of the tensors may decrease from iteration element n to iteration element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (RNN input vector length, **inputSize**). The third dimension of each tensor should be 1. Input vectors are expected to be arranged in the column-major order so strides in **xDesc** should be set as follows:

```
strideA[0]=inputSize, strideA[1]=1, strideA[2]=1
```

**x**

*Input*. Data pointer to GPU memory associated with the array of tensor descriptors **xDesc**. The input vectors are expected to be packed contiguously with the first vector of iterations (time-step) **n+1** following directly the last vector of iteration **n**. In other words, input vectors for all RNN time-steps should be packed in the contiguous block of GPU memory with no gaps between the vectors.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

- If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudnnSetRNNDescriptor**.

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **xDesc**.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **hxDesc, cxDesc, wDesc, hyDesc, cyDesc** or one of the descriptors in **xDesc, yDesc** is invalid.
- The descriptors in one of **xDesc, hxDesc, cxDesc, wDesc, yDesc, hyDesc, cyDesc** have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.168. cudnnRNNForwardTrainingEx

```
cudnnStatus_t cudnnRNNForwardTrainingEx(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const cudnnRNNDataDescriptor_t     xDesc,
    const void                         *x,
    const cudnnTensorDescriptor_t      hxDesc,
    const void                         *hx,
    const cudnnTensorDescriptor_t      cxDesc,
    const void                         *cx,
    const cudnnFilterDescriptor_t      wDesc,
    const void                         *w,
    const cudnnRNNDataDescriptor_t     yDesc,
```

```
    void                              *y,
    const cudnnTensorDescriptor_t     hyDesc,
    void                              *hy,
    const cudnnTensorDescriptor_t     cyDesc,
    void                              *cy,
    const cudnnRNNDataDescriptor_t    kDesc,
    const void                        *keys,
    const cudnnRNNDataDescriptor_t    cDesc,
    void                              *cAttn,
    const cudnnRNNDataDescriptor_t    iDesc,
    void                              *iAttn,
    const cudnnRNNDataDescriptor_t    qDesc,
    void                              *queries,
    void                              *workSpace,
    size_t                            workSpaceSizeInBytes,
    void                              *reserveSpace,
    size_t                            reserveSpaceSizeInBytes);
```

This routine is the extended version of the **cudnnRNNForwardTraining** function. The **cudnnRNNForwardTrainingEx** allows the user to use unpacked (padded) layout for input **x** and output **y**.

In the unpacked layout, each sequence in the mini-batch is considered to be of fixed length, specified by **maxSeqLength** in its corresponding **RNNDataDescriptor**. Each fixed-length sequence, for example, the **nth** sequence in the mini-batch, is composed of a valid segment specified by the **seqLengthArray[n]** in its corresponding **RNNDataDescriptor**; and a padding segment to make the combined sequence length equal to **maxSeqLength**.

With the unpacked layout, both sequence major (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function **cudnnRNNForwardTraining**, the sequences in the mini-batch need to be sorted in descending order according to length.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**rnnDesc**

*Input*. A previously initialized RNN descriptor.

**xDesc**

*Input*. A previously initialized RNN Data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, and **seqLengthArray** need to match that of **yDesc**.

**x**

*Input*. Data pointer to the GPU memory associated with the RNN data descriptor **xDesc**. The input vectors are expected to be laid out in memory according to the layout specified by **xDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hxDesc**

*Input*. A fully packed tensor descriptor describing the initial hidden state of the RNN.

The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. Moreover:

- If **direction** is **CUDNN_UNIDIRECTIONAL** then the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** then the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the **batchSize** parameter in **xDesc**. The third dimension depends on whether RNN mode is **CUDNN_LSTM** and whether **LSTM** projection is enabled. Additionally:

- If RNN mode is **CUDNN_LSTM** and **LSTM** projection is enabled, the third dimension must match the **recProjSize** argument passed to **cudnnSetRNNProjectionLayers** call used to set **rnnDesc**.
- Otherwise, the third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**.

**hx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a **NULL** pointer is passed, the initial hidden state of the network will be initialized to zero.

**cxDesc**

*Input*. A fully packed tensor descriptor describing the initial cell state for LSTM networks.

The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. Additionally:

- If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudnnSetRNNDescriptor**.
- If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudnnSetRNNDescriptor**.

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

**cx**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a **NULL** pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input*. Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

*Input*. Data pointer to GPU memory associated with the filter descriptor **wDesc**.

**yDesc**

*Input*. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, and **seqLengthArray** need to match that of **dyDesc** and **dxDesc**. The parameter **vectorSize** depends on whether the RNN mode is **CUDNN_LSTM** and whether LSTM projection is enabled and whether the network is bidirectional. Specifically:

- For unidirectional network, if the RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the parameter **vectorSize** must match the **recProjSize** argument passed to **cudnnSetRNNProjectionLayers** call used to set **rnnDesc**. If the network is bidirectional, then multiply the value by 2.
- Otherwise, for unidirectional network, the parameter **vectorSize** must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. If the network is bidirectional, then multiply the value by 2.

**y**

*Output*. Data pointer to GPU memory associated with the RNN data descriptor **yDesc**. The input vectors are expected to be laid out in memory according to the layout specified by **yDesc**. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hyDesc**

*Input*. A fully packed tensor descriptor describing the final hidden state of the RNN. The descriptor must be set exactly the same as **hxDesc**.

**hy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a **NULL** pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input*. A fully packed tensor descriptor describing the final cell state for LSTM networks. The descriptor must be set exactly the same as **cxDesc**.

**cy**

*Output*. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a **NULL** pointer is passed, the final cell state of the network will not be saved.

**kDesc**

Reserved. User may pass in **NULL**.

**keys**

Reserved. User may pass in **NULL**.

**cDesc**

Reserved. User may pass in **NULL**.

**cAttn**

Reserved. User may pass in **NULL**.

**iDesc**

Reserved. User may pass in **NULL**.

**iAttn**

Reserved. User may pass in **NULL**.

**qDesc**

Reserved. User may pass in **NULL**.

**queries**

Reserved. User may pass in **NULL**.

**workspace**

*Input*. Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **workspace**.

**reserveSpace**

*Input/Output*. Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input*. Specifies the size in bytes of the provided **reserveSpace**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- Variable sequence length input is passed in while **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used.
- **CUDNN_RNN_ALGO_PERSIST_STATIC** or **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** is used on pre-Pascal devices.
- Double input/output is used for **CUDNN_RNN_ALGO_PERSIST_STATIC**.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **xDesc**, **yDesc**, **hxDesc**, **cxDesc**, **wDesc**, **hyDesc**, and **cyDesc** is invalid, or have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

**CUDNN_STATUS_ALLOC_FAILED**

The function was unable to allocate memory.

# 4.169. cudnnRNNGetClip

```
cudnnStatus_t cudnnRNNGetClip(
    cudnnHandle_t               handle,
    cudnnRNNDescriptor_t        rnnDesc,
    cudnnRNNClipMode_t          *clipMode,
    cudnnNanPropagation_t       *clipNanOpt,
    double                      *lclip,
    double                      *rclip);
```

Retrieves the current LSTM cell clipping parameters, and stores them in the arguments provided.

### Parameters

***clipMode**

*Output*. Pointer to the location where the retrieved **clipMode** is stored. The **clipMode** can be **CUDNN_RNN_CLIP_NONE** in which case no LSTM cell state clipping is being performed; or **CUDNN_RNN_CLIP_MINMAX**, in which case the cell state activation to other units are being clipped.

***lclip**, ***rclip**

*Output*. Pointers to the location where the retrieved LSTM cell clipping range **[lclip, rclip]** is stored.

***clipNanOpt**

*Output*. Pointer to the location where the retrieved **clipNanOpt** is stored.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

If any of the pointer arguments provided are **NULL**.

## 4.170. cudnnRNNSetClip

```
cudnnStatus_t cudnnRNNSetClip(
    cudnnHandle_t                handle,
    cudnnRNNDescriptor_t         rnnDesc,
    cudnnRNNClipMode_t           clipMode,
    cudnnNanPropagation_t        clipNanOpt,
    double                       lclip,
    double                       rclip);
```

Sets the LSTM cell clipping mode. The LSTM clipping is disabled by default. When enabled, clipping is applied to all layers. This **cudnnRNNSetClip()** function may be called multiple times.

### Parameters

**clipMode**

*Input*. Enables or disables the LSTM cell clipping. When **clipMode** is set to **CUDNN_RNN_CLIP_NONE** no LSTM cell state clipping is performed. When **clipMode** is **CUDNN_RNN_CLIP_MINMAX** the cell state activation to other units are clipped.

**lclip, rclip**

*Input*. The range **[lclip, rclip]** to which the LSTM cell clipping should be set.

**clipNanOpt**

*Input*. When set to **CUDNN_PROPAGATE_NAN** (see the description for cudnnNanPropagation_t), NaN is propagated from the LSTM cell, or it can be set to one of the clipping range boundary values, instead of propagating.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

Returns this value if **lclip** > **rclip**; or if either **lclip** or **rclip** is **NaN**.

# 4.171. cudnnSaveAlgorithm

```
cudnnStatus_t cudnnSaveAlgorithm(
    cudnnHandle_t            handle,
    cudnnAlgorithmDescriptor_t  algoDesc,
    void*                    algoSpace
    size_t                   algoSpaceSizeInBytes)
```

This function writes algorithm metadata into the host memory space provided by the user in **algoSpace**, allowing the user to preserve the results of RNN finds after cuDNN exits.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**algoDesc**

*Input*. A previously created algorithm descriptor.

**algoSpace**

*Input*. Pointer to the host memory to be written.

**algoSpaceSizeInBytes**

*Input*. Amount of host memory needed as workspace to be able to save the metadata from the specified **algoDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions is met:

- One of the arguments is **NULL**.
- **algoSpaceSizeInBytes** is too small.

# 4.172. cudnnScaleTensor

```
cudnnStatus_t cudnnScaleTensor(
    cudnnHandle_t                    handle,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y,
    const void                      *alpha)
```

This function scale all the elements of a tensor by a given factor.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**y**

*Input/Output*. Pointer to data of the tensor described by the **yDesc** descriptor.

**alpha**

*Input*. Pointer in the host memory to a single value that all elements of the tensor will be scaled with. For more information, see Scaling Parameters in the *cuDNN Developer Guide*.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

One of the provided pointers is nil.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.173. cudnnSetActivationDescriptor

```
cudnnStatus_t cudnnSetActivationDescriptor(
    cudnnActivationDescriptor_t         activationDesc,
    cudnnActivationMode_t               mode,
    cudnnNanPropagation_t               reluNanOpt,
    double                              coef)
```

This function initializes a previously created generic activation descriptor object.

### Parameters

**activationDesc**

*Input/Output*. Handle to a previously created pooling descriptor.

**mode**

*Input*. Enumerant to specify the activation mode.

**reluNanOpt**

*Input*. Enumerant to specify the **Nan** propagation mode.

**coef**

*Input*. Floating point number. When the activation mode (see cudnnActivationMode_t) is set to **CUDNN_ACTIVATION_CLIPPED_RELU**, this input specifies the clipping threshold; and when the activation mode is set to **CUDNN_ACTIVATION_RELU**, this input specifies the upper bound.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

**mode** or **reluNanOpt** has an invalid enumerant value.

## 4.174. cudnnSetAlgorithmDescriptor

```
cudnnStatus_t cudnnSetAlgorithmDescriptor(
    cudnnAlgorithmDescriptor_t      algorithmDesc,
    cudnnAlgorithm_t                algorithm)
```

This function initializes a previously created generic algorithm descriptor object.

### Parameters

**algorithmDesc**

*Input/Output*. Handle to a previously created algorithm descriptor.

**algorithm**

*Input*. Struct to specify the algorithm.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

## 4.175. cudnnSetAlgorithmPerformance

```
cudnnStatus_t cudnnSetAlgorithmPerformance(
    cudnnAlgorithmPerformance_t     algoPerf,
    cudnnAlgorithmDescriptor_t      algoDesc,
    cudnnStatus_t                   status,
    float                           time,
    size_t                          memory)
```

This function initializes a previously created generic algorithm performance object.

### Parameters

**`algoPerf`**

*Input/Output*. Handle to a previously created algorithm performance object.

**`algoDesc`**

*Input*. The algorithm descriptor which the performance results describe.

**`status`**

*Input*. The cuDNN status returned from running the **`algoDesc`** algorithm.

**`time`**

*Input*. The GPU time spent running the **`algoDesc`** algorithm.

**`memory`**

*Input*. The GPU memory needed to run the **`algoDesc`** algorithm.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The object was set successfully.

**`CUDNN_STATUS_BAD_PARAM`**

**`mode`** or **`reluNanOpt`** has an invalid enumerate value.

## 4.176. cudnnSetAttnDescriptor

```
cudnnStatus_t cudnnSetAttnDescriptor(
 cudnnAttnDescriptor_t attnDesc,
 unsigned attnMode,
 int nHeads,
 double smScaler,
 cudnnDataType_t dataType,
 cudnnDataType_t computePrec,
 cudnnMathType_t mathType,
 cudnnDropoutDescriptor_t attnDropoutDesc,
 cudnnDropoutDescriptor_t postDropoutDesc,
 int qSize,
 int kSize,
 int vSize,
 int qProjSize,
 int kProjSize,
 int vProjSize,
 int oProjSize,
 int qoMaxSeqLength,
 int kvMaxSeqLength,
 int maxBatchSize,
 int maxBeamSize);
```

This function configures a multi-head attention descriptor that was previously created using the **`cudnnCreateAttnDescriptor()`** function. The function sets attention

parameters that are necessary to compute internal buffer sizes, dimensions of weight and bias tensors, or to select optimized code paths.

Input sequence data descriptors in **`cudnnMultiHeadAttnForward()`**, **`cudnnMultiHeadAttnBackwardData()`** and **`cudnnMultiHeadAttnBackwardWeights()`** functions are checked against the configuration parameters stored in the attention descriptor. Some parameters must match exactly while **`max`** arguments such as **`maxBatchSize`** or **`qoMaxSeqLength`** establish upper limits for the corresponding dimensions.

The multi-head attention model can be described by the following equations:$\mathbf{h}_i = (\mathbf{W}_{V,i}\mathbf{V})\,\mathrm{softmax}\left(\mathrm{smScaler}\left(\mathbf{K}^T\mathbf{W}_{K,i}^T\right)(\mathbf{W}_{Q,i}\mathbf{q})\right)$, for $i = 0 \ldots \mathrm{nHeads}-1$

$$\mathrm{MultiHeadAttn}(\mathbf{q}, \mathbf{K}, \mathbf{V}, \mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_O) = \sum_{i=0}^{\mathrm{nHeads}-1} \mathbf{W}_{O,i}\mathbf{h}_i$$

Where:

- nHeads is the number of independent attention heads that evaluate $\mathbf{h}_i$ vectors.
- $\mathbf{q}$ is a primary input, a single **`query`** column vector.
- $\mathbf{K}$, $\mathbf{V}$ are two matrices of **`key`** and **`value`** column vectors.

For simplicity, the above equations are presented using a single embedding vector $\mathbf{q}$ but the cuDNN API can handle multiple $\mathbf{q}$ candidates in the beam search scheme, process $\mathbf{q}$ vectors from multiple sequences bundled into a batch, or automatically iterate through all embedding vectors (time-steps) of a sequence. Thus, in general, $\mathbf{q}$, $\mathbf{K}$, $\mathbf{V}$ inputs are tensors with additional pieces of information such as active length of each sequence or how unused padding vectors should be saved.

In some publications, $\mathbf{W}_{O,i}$ matrices are combined into one output projection matrix and $\mathbf{h}_i$ vectors are merged explicitly into a single vector. This is an equivalent notation. In the cuDNN library, $\mathbf{W}_{O,i}$ matrices are conceptually treated the same way as $\mathbf{W}_{Q,i}$, $\mathbf{W}_{K,i}$ or $\mathbf{W}_{V,i}$ input projection weights. See the description of the **`cudnnGetMultiHeadAttnWeights()`** function for more details.

Weight matrices $\mathbf{W}_{Q,i}$, $\mathbf{W}_{K,i}$, $\mathbf{W}_{V,i}$ and $\mathbf{W}_{O,i}$ play similar roles, adjusting vector lengths in $\mathbf{q}$, $\mathbf{K}$, $\mathbf{V}$ inputs and in the multi-head attention final output. The user can disable any or all projections by setting qProjSize, kProjSize, vProjSize or oProjSize arguments to zero.

Embedding vector sizes in $\mathbf{q}$, $\mathbf{K}$, $\mathbf{V}$ and the vector lengths after projections need to be selected in such a way that matrix multiplications described above are feasible. Otherwise, **`CUDNN_STATUS_BAD_PARAM`** is returned by the **`cudnnSetAttnDescriptor()`** function. All four weight matrices are used when it is desirable to maintain rank deficiency of $\mathbf{W}_{KQ,i} = \mathbf{W}_{K,i}^T\mathbf{W}_{Q,i}$ or $\mathbf{W}_{OV,i} = \mathbf{W}_{O,i}\mathbf{W}_{V,i}$ matrices to eliminate one or more dimensions during linear transformations in each head. This is a form of feature extraction. In such cases, the projected sizes are smaller than the original vector lengths.

For each attention head, weight matrix sizes are defined as follows:

- $\mathbf{W}_{Q,i}$ - size $[\mathrm{qProjSize}\ x\ \mathrm{qSize}]$, $i = 0..\mathrm{nHeads}-1$
- $\mathbf{W}_{K,i}$ - size $[\mathrm{kProjSize}\ x\ \mathrm{kSize}]$, $i = 0..\mathrm{nHeads}-1$, $\mathrm{kProjSize} = \mathrm{qProjSize}$
- $\mathbf{W}_{V,i}$ - size $[\mathrm{vProjSize}\ x\ \mathrm{vSize}]$, $i = 0..\mathrm{nHeads}-1$

- $\mathbf{W}_{O,i}$ - size [oProjSize x (vProjSize > 0 ? vProjSize : vSize)], $i$ = 0..nHeads-1

When the output projection is disabled (oProjSize = 0), the output vector length is nHeads* (vProjSize > 0 ? vProjSize : vSize), meaning, the output is a concatenation of all $\mathbf{h}_i$ vectors. In the alternative interpretation, a concatenated matrix $\mathbf{W}_O = [\mathbf{W}_{O,0}, \mathbf{W}_{O,1}, \mathbf{W}_{O,2}, ...]$ forms the identity matrix.

Softmax is a normalized, exponential vector function that takes and outputs vectors of the same size. The multi-head attention API utilizes softmax of the **`CUDNN_SOFTMAX_ACCURATE`** type to reduce the likelihood of the floating-point overflow.

The **`smScaler`** parameter is the softmax sharpening/smoothing coefficient. When **`smScaler=1.0`**, softmax uses the natural exponential function **`exp(x)`** or **`2.7183*`**. When **`smScaler<1.0`**, for example **`smScaler=0.2`**, the function used by the softmax block will not grow as fast because **`exp(0.2*x) ≈ 1.2214*`**.

The **`smScaler`** parameter can be adjusted to process larger ranges of values fed to softmax. When the range is too large (or **`smScaler`** is not sufficiently small for the given range), the output vector of the softmax block becomes categorical, meaning, one vector element is close to 1.0 and other outputs are zero or very close to zero. When this occurs, the Jacobian matrix of the softmax block is also close to zero so deltas are not back-propagated during training from output to input except through residual connections, if these connections are enabled. The user can set **`smScaler`** to any positive floating-point value or even zero. The **`smScaler`** parameter is not trainable.

The **`qoMaxSeqLength`**, **`kvMaxSeqLength`**, **`maxBatchSize`**, and **`maxBeamSize`** arguments declare the maximum sequence lengths, maximum batch size, and maximum beam size respectively, in the **`cudnnSeqDataDescriptor_t`** containers. The actual dimensions supplied to forward and backward (gradient) API functions should not exceed the **`max`** limits. The **`max`** arguments should be set carefully because too large values will result in excessive memory usage due to oversized work and reserve space buffers.

The **`attnMode`** argument is treated as a binary mask where various on/off options are set. These options can affect the internal buffer sizes, enforce certain argument checks, select optimized code execution paths, or enable attention variants that do not require additional numerical arguments. An example of such options is the inclusion of biases in input and output projections.

The **`attnDropoutDesc`** and **`postDropoutDesc`** arguments are descriptors that define two dropout layers active in the training mode. The first dropout operation defined by **`attnDropoutDesc`**, is applied directly to the softmax output. The second dropout operation, specified by **`postDropoutDesc`**, alters the multi-head attention output, just before the point where residual connections are added.

The **`cudnnSetAttnDescriptor()`** function performs a shallow copy of **`attnDropoutDesc`** and **`postDropoutDesc`**, meaning, the addresses of both dropout descriptors are stored in the attention descriptor and not the entire structures. Therefore, the user should keep dropout descriptors during the entire life of the attention descriptor.

### Parameters

**attnDesc**

*Output*. Attention descriptor to be configured.

**attnMode**

*Input*. Enables various attention options that do not require additional numerical values. See the table below for the list of supported flags. The user should assign a preferred set of bitwise **OR-ed** flags to this argument.

**nHeads**

*Input*. Number of attention heads.

**smScaler**

*Input*. Softmax smoothing (**1.0 >= smScaler >= 0.0**) or sharpening (**smScaler > 1.0**) coefficient. Negative values are not accepted.

**dataType**

*Input*. Data type used to represent attention inputs, attention weights and attention outputs.

**computePrec**

*Input*. Compute precision.

**mathType**

*Input*. NVIDIA Tensor Core settings.

**attnDropoutDesc**

*Input*. Descriptor of the dropout operation applied to the softmax output. See the table below for a list of unsupported features.

**postDropoutDesc**

*Input*. Descriptor of the dropout operation applied to the multi-head attention output, just before the point where residual connections are added. See the table below for a list of unsupported features.

**qSize, kSize, vSize**

*Input*. **Q**, **K**, **V** embedding vector lengths.

**qProjSize, kProjSize, vProjSize**

*Input*. **Q**, **K**, **V** embedding vector lengths after input projections. Use zero to disable the corresponding projection.

**oProjSize**

*Input*. The $\mathbf{h}_i$ vector length after the output projection. Use zero to disable this projection.

**qoMaxSeqLength**

*Input*. Largest sequence length expected in sequence data descriptors related to **Q**, **O**, **dQ** and **dO** inputs and outputs.

**kvMaxSeqLength**

*Input*. Largest sequence length expected in sequence data descriptors related to **K**, **V**, **dK** and **dV** inputs and outputs.

**maxBatchSize**

*Input*. Largest batch size expected in any **cudnnSeqDataDescriptor_t** container.

**maxBeamSize**

*Input*. Largest beam size expected in any **cudnnSeqDataDescriptor_t** container.

### Supported attnMode flags

**CUDNN_ATTN_QUERYMAP_ALL_TO_ONE**

Forward declaration of mapping between **Q** and **K**, **V** vectors when the beam size is greater than one in the **Q** input. Multiple **Q** vectors from the same beam bundle map to the same **K**, **V** vectors. This means that beam sizes in the **K**, **V** sets are equal to one.

**CUDNN_ATTN_QUERYMAP_ONE_TO_ONE**

Forward declaration of mapping between **Q** and **K**, **V** vectors when the beam size is greater than one in the **Q** input. Multiple **Q** vectors from the same beam bundle map to different **K**, **V** vectors. This requires beam sizes in **K**, **V** sets to be the same as in the **Q** input.

**CUDNN_ATTN_DISABLE_PROJ_BIASES**

Use no biases in the attention input and output projections.

**CUDNN_ATTN_ENABLE_PROJ_BIASES**

Use extra biases in the attention input and output projections. In this case the projected $\overline{\mathbf{K}}$ vectors are computed as $\overline{\mathbf{K}}_i = \mathbf{W}_{K,i}\mathbf{K} + \mathbf{b} * [1, 1, ..., 1]_{1\times n}$, where $n$ is the number of columns in the **K** matrix. In other words, the same column vector **b** is added to all columns of **K** after the weight matrix multiplication.

### Supported combinations of dataType, computePrec, and mathType

## Table 26 Supported combinations

| dataType | computePrec | mathType |
|---|---|---|
| CUDNN_DATA_DOUBLE | CUDNN_DATA_DOUBLE | CUDNN_DEFAULT_MATH |
| CUDNN_DATA_FLOAT | CUDNN_DATA_FLOAT | CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION |

| dataType | computePrec | mathType |
|---|---|---|
| CUDNN_DATA_HALF | CUDNN_DATA_HALF | CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION |

### Unsupported features

1. The dropout option is currently not supported by the multi-head attention API. Assign **NULL** to **attnDropoutDesc** and **postDropoutDesc** arguments when configuring the attention descriptor.
2. The **CUDNN_ATTN_ENABLE_PROJ_BIASES** option is not supported in the multi-head attention gradient functions.
3. The **paddingFill** argument in **cudnnSeqDataDescriptor_t** is currently ignored by all multi-head attention functions.

### Returns

**CUDNN_STATUS_SUCCESS**

The attention descriptor was configured successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was encountered. Some examples include:

- post projection **Q** and **K** sizes were not equal
- **dataType**, **computePrec**, or **mathType** were invalid
- one or more of the following arguments were either negative or zero: **nHeads**, **qSize**, **kSize**, **vSize**, **qoMaxSeqLength**, **kvMaxSeqLength**, **maxBatchSize**, **maxBeamSize**
- one or more of the following arguments were negative: **qProjSize**, **kProjSize**, **vProjSize**, **smScaler**

**CUDNN_STATUS_NOT_SUPPORTED**

A requested option or a combination of input arguments is not supported.

# 4.177. cudnnSetCallback

```
cudnnStatus_t cudnnSetCallback(
        unsigned             mask,
        void                 *udata,
        cudnnCallback_t      fptr)
```

This function sets the internal states of cuDNN error reporting functionality.

### Parameters

**mask**

*Input*. An unsigned integer. The four least significant bits (LSBs) of this unsigned integer are used for switching on and off the different levels of error reporting messages. This applies for both the default callbacks, and for the customized callbacks. The bit position is in correspondence with the enum of **cudnnSeverity_t**. The user may utilize the predefined macros **CUDNN_SEV_ERROR_EN**, **CUDNN_SEV_WARNING_EN**, and **CUDNN_SEV_INFO_EN** to form the bit mask. When a bit is set to 1, the corresponding message channel is enabled.

For example, when bit 3 is set to 1, the API logging is enabled. Currently only the log output of level **CUDNN_SEV_INFO** is functional; the others are not yet implemented. When used for turning on and off the logging with the default callback, the user may pass **NULL** to **udata** and **fptr**. In addition, the environment variable **CUDNN_LOGDEST_DBG** must be set. For more information, see the Backward compatibility and deprecation policy section in the *cuDNN Developer Guide*.

- **CUDNN_SEV_INFO_EN**= 0b1000 (functional).
- **CUDNN_SEV_ERROR_EN**= 0b0010 (not yet functional).
- **CUDNN_SEV_WARNING_EN**= 0b0100 (not yet functional).

The output of C**UDNN_SEV_FATAL** is always enabled and cannot be disabled.

**udata**

*Input*. A pointer provided by the user. This pointer will be passed to the user's custom logging callback function. The data it points to will not be read, nor be changed by cuDNN. This pointer may be used in many ways, such as in a mutex or in a communication socket for the user's callback function for logging. If the user is utilizing the default callback function, or doesn't want to use this input in the customized callback function, they may pass in **NULL**.

**fptr**

*Input*. A pointer to a user-supplied callback function. When **NULL** is passed to this pointer, then cuDNN switches back to the built-in default callback function. The user-supplied callback function prototype must be similar to the following (also defined in the header file):

```
void customizedLoggingCallback (cudnnSeverity_t sev, void *udata, const
 cudnnDebug_t *dbg, const char *msg);
```

- The structure **cudnnDebug_t** is defined in the header file. It provides the metadata, such as time, time since start, stream ID, process and thread ID, that the user may choose to print or store in their customized callback.
- The variable **msg** is the logging message generated by cuDNN. Each line of this message is terminated by **\0**, and the end of message is terminated by **\0\0**. User may select what is necessary to show in the log, and may reformat the string.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

## 4.178. cudnnSetConvolution2dDescriptor

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             pad_h,
    int                             pad_w,
    int                             u,
    int                             v,
    int                             dilation_h,
    int                             dilation_w,
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 computeType)
```

This function initializes a previously created convolution descriptor object into a 2D correlation. This function assumes that the tensor and filter descriptors corresponds to the forward convolution path and checks if their settings are valid. That same convolution descriptor can be reused in the backward path provided it corresponds to the same layer.

### Parameters

**convDesc**

*Input/Output*. Handle to a previously created convolution descriptor.

**pad_h**

*Input*. Zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

**pad_w**

*Input*. Zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

**u**

*Input*. Vertical filter stride.

**v**

*Input*. Horizontal filter stride.

**dilation_h**

*Input*. Filter height dilation.

**dilation_w**

*Input*. Filter width dilation.

**mode**

*Input*. Selects between **CUDNN_CONVOLUTION** and **CUDNN_CROSS_CORRELATION**.

**computeType**

*Input*. compute precision.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **convDesc** is nil.
- One of the parameters **pad_h**, **pad_w** is strictly negative.
- One of the parameters **u**, **v** is negative or zero.
- One of the parameters **dilation_h**, **dilation_w** is negative or zero.
- The parameter **mode** has an invalid enumerant value.

## 4.179. cudnnSetConvolutionGroupCount

```
cudnnStatus_t cudnnSetConvolutionGroupCount(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             groupCount)
```

This function allows the user to specify the number of groups to be used in the associated convolution.

### Returns

**CUDNN_STATUS_SUCCESS**

The group count was set successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid convolution descriptor was provided

## 4.180. cudnnSetConvolutionMathType

```
cudnnStatus_t cudnnSetConvolutionMathType(
    cudnnConvolutionDescriptor_t    convDesc,
    cudnnMathType_t                 mathType)
```

This function allows the user to specify whether or not the use of tensor op is permitted in the library routines associated with a given convolution descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The math type was set successfully.

**CUDNN_STATUS_BAD_PARAM**

Either an invalid convolution descriptor was provided or an invalid math type was specified.

## 4.181. cudnnSetConvolutionNdDescriptor

```
cudnnStatus_t cudnnSetConvolutionNdDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             arrayLength,
    const int                       padA[],
    const int                       filterStrideA[],
    const int                       dilationA[],
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 dataType)
```

This function initializes a previously created generic convolution descriptor object into a n-D correlation. That same convolution descriptor can be reused in the backward path provided it corresponds to the same layer. The convolution computation will be done in the specified **dataType**, which can be potentially different from the input/output tensors.

### Parameters

**convDesc**

*Input/Output*. Handle to a previously created convolution descriptor.

**arrayLength**

*Input*. Dimension of the convolution.

**padA**

*Input*. Array of dimension **arrayLength** containing the zero-padding size for each dimension. For every dimension, the padding represents the number of extra zeros implicitly concatenated at the start and at the end of every element of that dimension.

**filterStrideA**

*Input*. Array of dimension **arrayLength** containing the filter stride for each dimension. For every dimension, the filter stride represents the number of elements to slide to reach the next start of the filtering window of the next point.

**dilationA**

*Input*. Array of dimension **arrayLength** containing the dilation factor for each dimension.

**mode**

*Input*. Selects between **CUDNN_CONVOLUTION** and **CUDNN_CROSS_CORRELATION**.

**datatype**

*Input*. Selects the data type in which the computation will be done.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The descriptor **convDesc** is nil.
- The **arrayLengthRequest** is negative.
- The enumerant **mode** has an invalid value.
- The enumerant **datatype** has an invalid value.
- One of the elements of **padA** is strictly negative.
- One of the elements of **strideA** is negative or zero.
- One of the elements of **dilationA** is negative or zero.

**CUDNN_STATUS_NOT_SUPPORTED**

At least one of the following conditions are met:

- The **arrayLengthRequest** is greater than **CUDNN_DIM_MAX**.

## 4.182. cudnnSetConvolutionReorderType

```
cudnnStatus_t cudnnSetConvolutionReorderType(
 cudnnConvolutionDescriptor_t convDesc,
 cudnnReorderType_t reorderType);
```

This function sets the convolution reorder type for the given convolution descriptor.

### Parameters

**convDesc**

*Input*. The convolution descriptor for which the reorder type should be set.

**reorderType**

*Input*. Set the reorder type to this value. For more information, see cudnnReorderType_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

The reorder type supplied is not supported.

**CUDNN_STATUS_SUCCESS**

Reorder type is set successfully.

# 4.183. cudnnSetCTCLossDescriptor

```
cudnnStatus_t cudnnSetCTCLossDescriptor(
    cudnnCTCLossDescriptor_t        ctcLossDesc,
    cudnnDataType_t                 compType)
```

This function sets a CTC loss function descriptor. See also the extended version cudnnSetCTCLossDescriptorEx to set the input normalization mode.

When the extended version **cudnnSetCTCLossDescriptorEx** is used with **normMode** set to **CUDNN_LOSS_NORMALIZATION_NONE** and the **gradMode** set to **CUDNN_NOT_PROPAGATE_NAN**, then it is the same as the current function **cudnnSetCTCLossDescriptor**, meaning:

```
cudnnSetCtcLossDescriptor(*) = cudnnSetCtcLossDescriptorEx(*,
 normMode=CUDNN_LOSS_NORMALIZATION_NONE, gradMode=CUDNN_NOT_PROPAGATE_NAN)
```

### Parameters

**ctcLossDesc**

*Output*. CTC loss descriptor to be set.

**compType**

*Input*. Compute type for this CTC loss function.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of input parameters passed is invalid.

# 4.184. cudnnSetCTCLossDescriptorEx

```
cudnnStatus_t cudnnSetCTCLossDescriptorEx(
    cudnnCTCLossDescriptor_t        ctcLossDesc,
    cudnnDataType_t                 compType,
    cudnnLossNormalizationMode_t    normMode,
    cudnnNanPropagation_t           gradMode)
```

This function is an extension of cudnnSetCTCLossDescriptor. This function provides an additional interface **normMode** to set the input normalization mode for the CTC loss function, and **gradMode** to control the **NaN** propagation type.

When this function **cudnnSetCTCLossDescriptorEx** is used with **normMode** set to **CUDNN_LOSS_NORMALIZATION_NONE** and the **gradMode** set to **CUDNN_NOT_PROPAGATE_NAN**, then it is the same as **cudnnSetCTCLossDescriptor**, meaning:

```
cudnnSetCtcLossDescriptor(*) = cudnnSetCtcLossDescriptorEx(*,
 normMode=CUDNN_LOSS_NORMALIZATION_NONE, gradMode=CUDNN_NOT_PROPAGATE_NAN)
```

### Parameters

**ctcLossDesc**

*Output*. CTC loss descriptor to be set.

**compType**

*Input*. Compute type for this CTC loss function.

**normMode**

*Input*. Input normalization type for this CTC loss function. For more information, see cudnnLossNormalizationMode_t.

**gradMode**

*Input*. NaN propagation type for this CTC loss function. For **L** the sequence length, **R** the number of repeated letters in the sequence, and **T** the length of sequential data, the following applies: when a sample with **L+R > T** is encountered during the gradient calculation, if **gradMode** is set to **CUDNN_PROPAGATE_NAN** (see cudnnNanPropagation_t), then the CTC loss function does not write to the gradient buffer for that sample. Instead, the current values, even not finite, are retained. If **gradMode** is set to **CUDNN_NOT_PROPAGATE_NAN**, then the gradient for that sample is set to zero. This guarantees finite gradient.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of input parameters passed is invalid.

## 4.185. cudnnSetDropoutDescriptor

```
cudnnStatus_t cudnnSetDropoutDescriptor(
    cudnnDropoutDescriptor_t    dropoutDesc,
    cudnnHandle_t               handle,
    float                       dropout,
    void                       *states,
```

```
    size_t                          stateSizeInBytes,
    unsigned long long              seed)
```

This function initializes a previously created dropout descriptor object. If the **states** argument is equal to **NULL**, then the random number generator states won't be initialized, and only the **dropout** value will be set. No other function should be writing to the memory pointed at by the **states** argument while this function is running. The user is expected not to change the memory pointed at by **states** for the duration of the computation.

### Parameters

**dropoutDesc**

*Input/Output*. Previously created dropout descriptor object.

**handle**

*Input*. Handle to a previously created cuDNN context.

**dropout**

*Input*. The probability with which the value from input is set to zero during the dropout layer.

**states**

*Output*. Pointer to user-allocated GPU memory that will hold random number generator states.

**stateSizeInBytes**

*Input*. Specifies the size in bytes of the provided memory for the states.

**seed**

*Input*. Seed used to initialize random number generator states.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_INVALID_VALUE**

**sizeInBytes** is less than the value returned by **cudnnDropoutGetStatesSize**.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.186. cudnnSetFilter4dDescriptor

```
cudnnStatus_t cudnnSetFilter4dDescriptor(
    cudnnFilterDescriptor_t    filterDesc,
    cudnnDataType_t            dataType,
    cudnnTensorFormat_t        format,
```

```
    int                         k,
    int                         c,
    int                         h,
    int                         w)
```

This function initializes a previously created filter descriptor object into a 4D filter. The layout of the filters must be contiguous in memory.

Tensor format **CUDNN_TENSOR_NHWC** has limited support in **cudnnConvolutionForward**, **cudnnConvolutionBackwardData**, and **cudnnConvolutionBackwardFilter**.

### Parameters

**filterDesc**

*Input/Output*. Handle to a previously created filter descriptor.

**datatype**

*Input*. Data type.

**format**

*Input*.Type of the filter layout format. If this input is set to **CUDNN_TENSOR_NCHW**, which is one of the enumerant values allowed by **cudnnTensorFormat_t** descriptor, then the layout of the filter is in the form of **KCRS**, where:

- **K** represents the number of output feature maps
- **C** is the number of input feature maps
- **R** is the number of rows per filter
- **S** is the number of columns per filter

If this input is set to **CUDNN_TENSOR_NHWC**, then the layout of the filter is in the form of **KRSC**. For more information, see cudnnTensorFormat_t.

**k**

*Input*. Number of output feature maps.

**c**

*Input*. Number of input feature maps.

**h**

*Input*. Height of each filter.

**w**

*Input*. Width of each filter.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the parameters **k**, **c**, **h**, **w** is negative or **dataType** or **format** has an invalid enumerant value.

# 4.187. cudnnSetFilterNdDescriptor

```
cudnnStatus_t cudnnSetFilterNdDescriptor(
    cudnnFilterDescriptor_t filterDesc,
    cudnnDataType_t         dataType,
    cudnnTensorFormat_t     format,
    int                     nbDims,
    const int               filterDimA[])
```

This function initializes a previously created filter descriptor object. The layout of the filters must be contiguous in memory.

The tensor format **CUDNN_TENSOR_NHWC** has limited support in **cudnnConvolutionForward**, **cudnnConvolutionBackwardData**, and **cudnnConvolutionBackwardFilter**.

### Parameters

**filterDesc**

*Input/Output*. Handle to a previously created filter descriptor.

**datatype**

*Input*. Data type.

**format**

*Input*.Type of the filter layout format. If this input is set to **CUDNN_TENSOR_NCHW**, which is one of the enumerant values allowed by **cudnnTensorFormat_t** descriptor, then the layout of the filter is as follows:

- For **N=4**, a 4D filter descriptor, the filter layout is in the form of **KCRS**:
  - **K** represents the number of output feature maps
  - **C** is the number of input feature maps
  - **R** is the number of rows per filter
  - **S** is the number of columns per filter
- For **N=3**, a 3D filter descriptor, the number **S** (number of columns per filter) is omitted.
- For **N=5** and greater, the layout of the higher dimensions immediately follow **RS**.

On the other hand, if this input is set to **CUDNN_TENSOR_NHWC**, then the layout of the filter is as follows:

- For **N=4**, a 4D filter descriptor, the filter layout is in the form of **KRSC**.

- For **N=3**, a 3D filter descriptor, the number **S** (number of columns per filter) is omitted and the layout of **C** immediately follows **R**.
- For **N=5** and greater, the layout of the higher dimensions are inserted between **S** and **C**. For more information, see cudnnTensorFormat_t.

**nbDims**

*Input*. Dimension of the filter.

**filterDimA**

*Input*. Array of dimension **nbDims** containing the size of the filter for each dimension.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the elements of the array **filterDimA** is negative or **dataType** or **format** has an invalid enumerant value.

**CUDNN_STATUS_NOT_SUPPORTED**

The parameter **nbDims** exceeds **CUDNN_DIM_MAX**.

## 4.188. cudnnSetFusedOpsConstParamPackAttribute

```
cudnnStatus_t cudnnSetFusedOpsConstParamPackAttribute(
 cudnnFusedOpsConstParamPack_t constPack,
 cudnnFusedOpsConstParamLabel_t paramLabel,
 const void *param);
```

This function sets the descriptor pointed to by the **param** pointer input. The type of the descriptor to be set is indicated by the enum value of the `paramLabel` input.

### Parameters

**constPack**

*Input*. The opaque cudnnFusedOpsConstParamPack_t structure that contains the various problem size information, such as the shape, layout and the type of tensors, the descriptors for convolution and activation, and settings for operations such as convolution and activation.

**paramLabel**

*Input*. Several types of descriptors can be set by this setter function. The **param** input points to the descriptor itself, and this input indicates the type of the descriptor pointed to by the **param** input. The cudnnFusedOpsConstParamLabel_t enumerant type enables the selection of the type of the descriptor.

**param**

*Input*. Data pointer to the host memory, associated with the specific descriptor. The type of the descriptor depends on the value of **paramLabel**. For more information, see the table in cudnnFusedOpsConstParamLabel_t.

If this pointer is set to **NULL**, then the cuDNN library will record as such. If not, then the values pointed to by this pointer (meaning, the value or the opaque structure underneath) will be copied into the **constPack** during **cudnnSetFusedOpsConstParamPackAttribute()** operation.

### Returns

**CUDNN_STATUS_SUCCESS**

The descriptor is set successfully.

**CUDNN_STATUS_BAD_PARAM**

If **constPack** is **NULL**, or if **paramLabel** or the ops setting for **constPack** is invalid.

## 4.189. cudnnSetFusedOpsVariantParamPackAttribute

```
cudnnStatus_t cudnnSetFusedOpsVariantParamPackAttribute(
 cudnnFusedOpsVariantParamPack_t varPack,
 cudnnFusedOpsVariantParamLabel_t paramLabel,
 void *ptr);
```

This function sets the variable parameter pack descriptor.

### Parameters

**varPack**

*Input*. Pointer to the **cudnnFusedOps** variant parameter pack (**varPack**) descriptor.

**paramLabel**

*Input*. Type to which the buffer pointer parameter (in the **varPack** descriptor) is set by this function. For more information, see cudnnFusedOpsConstParamLabel_t.

**ptr**

*Input*. Pointer, to the host or device memory, to the value to which the descriptor parameter is set. The data type of the pointer, and the host/device memory location, depend on the **paramLabel** input selection. For more information, see cudnnFusedOpsVariantParamLabel_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

If **varPack** is **NULL** or if **paramLabel** is set to an unsupported value.

**CUDNN_STATUS_SUCCESS**

The descriptor was set successfully.

# 4.190. cudnnSetLRNDescriptor

```
cudnnStatus_t cudnnSetLRNDescriptor(
    cudnnLRNDescriptor_t   normDesc,
    unsigned               lrnN,
    double                 lrnAlpha,
    double                 lrnBeta,
    double                 lrnK)
```

This function initializes a previously created LRN descriptor object.

- Macros **CUDNN_LRN_MIN_N**, **CUDNN_LRN_MAX_N**, **CUDNN_LRN_MIN_K**, **CUDNN_LRN_MIN_BETA** defined in **cudnn.h** specify valid ranges for parameters.
- Values of double parameters will be cast down to the tensor **datatype** during computation.

### Parameters

**normDesc**

*Output*. Handle to a previously created LRN descriptor.

**lrnN**

*Input*. Normalization window width in elements. LRN layer uses a window **[center-lookBehind, center+lookAhead]**, where **lookBehind = floor( (lrnN-1)/2 )**, **lookAhead = lrnN-lookBehind-1**. So for **n=10**, the window is **[k-4...k...k+5]** with a total of 10 samples. For **DivisiveNormalization** layer, the window has the same extents as above in all spatial dimensions (**dimA[2]**, **dimA[3]**, **dimA[4]**). By default, **lrnN** is set to **5** in **cudnnCreateLRNDescriptor**.

**lrnAlpha**

*Input*. Value of the alpha variance scaling parameter in the normalization formula. Inside the library code, this value is divided by the window width for LRN and by **(window width)^#spatialDimensions** for **DivisiveNormalization**. By default, this value is set to **1e-4** in **cudnnCreateLRNDescriptor**.

**lrnBeta**

*Input*. Value of the beta power parameter in the normalization formula. By default, this value is set to **0.75** in **cudnnCreateLRNDescriptor**.

**lrnK**

*Input*. Value of the **k** parameter in the normalization formula. By default, this value is set to **2.0**.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

One of the input parameters was out of valid range as described above.

## 4.191. cudnnSetOpTensorDescriptor

```
cudnnStatus_t cudnnSetOpTensorDescriptor(
    cudnnOpTensorDescriptor_t   opTensorDesc,
    cudnnOpTensorOp_t           opTensorOp,
    cudnnDataType_t             opTensorCompType,
    cudnnNanPropagation_t       opTensorNanOpt)
```

This function initializes a tensor pointwise math descriptor.

### Parameters

**opTensorDesc**

*Output*. Pointer to the structure holding the description of the tensor pointwise math descriptor.

**opTensorOp**

*Input*. Tensor pointwise math operation for this tensor pointwise math descriptor.

**opTensorCompType**

*Input*. Computation datatype for this tensor pointwise math descriptor.

**opTensorNanOpt**

*Input*. NAN propagation policy.

### Returns

**CUDNN_STATUS_SUCCESS**

The function returned successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of input parameters passed is invalid.

## 4.192. cudnnSetPersistentRNNPlan

```
cudnnStatus_t cudnnSetPersistentRNNPlan(
    cudnnRNNDescriptor_t        rnnDesc,
    cudnnPersistentRNNPlan_t    plan)
```

This function sets the persistent RNN plan to be executed when using **rnnDesc** and **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** algo.

### Returns

**CUDNN_STATUS_SUCCESS**

The plan was set successfully.

**CUDNN_STATUS_BAD_PARAM**

The algo selected in **rnnDesc** is not **CUDNN_RNN_ALGO_PERSIST_DYNAMIC**.

## 4.193. cudnnSetPooling2dDescriptor

```
cudnnStatus_t cudnnSetPooling2dDescriptor(
    cudnnPoolingDescriptor_t    poolingDesc,
    cudnnPoolingMode_t          mode,
    cudnnNanPropagation_t       maxpoolingNanOpt,
    int                         windowHeight,
    int                         windowWidth,
    int                         verticalPadding,
    int                         horizontalPadding,
    int                         verticalStride,
    int                         horizontalStride)
```

This function initializes a previously created generic pooling descriptor object into a 2D description.

### Parameters

**poolingDesc**

*Input/Output*. Handle to a previously created pooling descriptor.

**mode**

*Input*. Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Input*. Enumerant to specify the Nan propagation mode.

**windowHeight**

*Input*. Height of the pooling window.

**windowWidth**

*Input*. Width of the pooling window.

**verticalPadding**

*Input*. Size of vertical padding.

**horizontalPadding**

*Input*. Size of horizontal padding

**verticalStride**

*Input*. Pooling vertical stride.

**horizontalStride**

*Input*. Pooling horizontal stride.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the parameters **windowHeight**, **windowWidth**, **verticalStride**, **horizontalStride** is negative or **mode** or **maxpoolingNanOpt** has an invalid enumerate value.

## 4.194. cudnnSetPoolingNdDescriptor

```
cudnnStatus_t cudnnSetPoolingNdDescriptor(
    cudnnPoolingDescriptor_t     poolingDesc,
    const cudnnPoolingMode_t     mode,
    const cudnnNanPropagation_t  maxpoolingNanOpt,
    int                          nbDims,
    const int                    windowDimA[],
    const int                    paddingA[],
    const int                    strideA[])
```

This function initializes a previously created generic pooling descriptor object.

### Parameters

**poolingDesc**

*Input/Output*. Handle to a previously created pooling descriptor.

**mode**

*Input*. Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Input*. Enumerant to specify the Nan propagation mode.

**nbDims**

*Input*. Dimension of the pooling operation. Must be greater than zero.

**windowDimA**

*Input*. Array of dimension **nbDims** containing the window size for each dimension. The value of array elements must be greater than zero.

**paddingA**

*Input*. Array of dimension **nbDims** containing the padding size for each dimension. Negative padding is allowed.

**strideA**

*Input*. Array of dimension **nbDims** containing the striding size for each dimension. The value of array elements must be greater than zero (meaning, negative striding size is not allowed).

### Returns

**CUDNN_STATUS_SUCCESS**

The object was initialized successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

If (**nbDims** > **CUDNN_DIM_MAX-2**).

**CUDNN_STATUS_BAD_PARAM**

Either **nbDims**, or at least one of the elements of the arrays **windowDimA** or **strideA** is negative, or **mode** or **maxpoolingNanOpt** has an invalid enumerate value.

## 4.195. cudnnSetReduceTensorDescriptor

```
cudnnStatus_t cudnnSetReduceTensorDescriptor(
    cudnnReduceTensorDescriptor_t   reduceTensorDesc,
    cudnnReduceTensorOp_t           reduceTensorOp,
    cudnnDataType_t                 reduceTensorCompType,
    cudnnNanPropagation_t           reduceTensorNanOpt,
    cudnnReduceTensorIndices_t      reduceTensorIndices,
    cudnnIndicesType_t              reduceTensorIndicesType)
```

This function initializes a previously created reduce tensor descriptor object.

### Parameters

**reduceTensorDesc**

*Input/Output*. Handle to a previously created reduce tensor descriptor.

**reduceTensorOp**

*Input*. Enumerant to specify the reduce tensor operation.

**reduceTensorCompType**

*Input*. Enumerant to specify the computation datatype of the reduction.

**reduceTensorNanOpt**

*Input*. Enumerant to specify the Nan propagation mode.

**reduceTensorIndices**

*Input*. Enumerant to specify the reduce tensor indices.

**reduceTensorIndicesType**

*Input*. Enumerant to specify the reduce tensor indices type.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

**reduceTensorDesc** is **NULL** (**reduceTensorOp**, **reduceTensorCompType**, **reduceTensorNanOpt**, **reduceTensorIndices** or **reduceTensorIndicesType** has an invalid enumerant value).

## 4.196. cudnnSetRNNBiasMode

```
cudnnStatus_t cudnnSetRNNBiasMode(
   cudnnRNNDescriptor_t   rnnDesc,
   cudnnRNNBiasMode_t     biasMode)
```

The **cudnnSetRNNBiasMode()** function sets the number of bias vectors for a previously created and initialized RNN descriptor. This function should be called after cudnnSetRNNDescriptor to enable the specified bias mode in an RNN. The default value of **biasMode** in **rnnDesc** after **cudnnCreateRNNDescriptor()** is **CUDNN_RNN_DOUBLE_BIAS**.

### Parameters

**rnnDesc**

*Input/Output.* A previously created RNN descriptor.

**biasMode**

*Input*. Sets the number of bias vectors. For more information, see cudnnRNNBiasMode_t.

### Returns

**CUDNN_STATUS_BAD_PARAM**

Either the **rnnDesc** is **NULL** or **biasMode** has an invalid enumerant value.

**CUDNN_STATUS_SUCCESS**

The **biasMode** was set successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

Non-default bias mode (an enumerated type besides **CUDNN_RNN_DOUBLE_BIAS**) applied to an RNN algo other than **CUDNN_RNN_ALGO_STANDARD**.

# 4.197. cudnnSetRNNDataDescriptor

```
cudnnStatus_t cudnnSetRNNDataDescriptor(
    cudnnRNNDataDescriptor_t       RNNDataDesc,
    cudnnDataType_t                dataType,
    cudnnRNNDataLayout_t           layout,
    int                            maxSeqLength,
    int                            batchSize,
    int                            vectorSize,
    const int                      seqLengthArray[],
    void                           *paddingFill);
```

This function initializes a previously created RNN data descriptor object. This data structure is intended to support the unpacked (padded) layout for input and output of extended RNN inference and training functions. A packed (unpadded) layout is also supported for backward compatibility.

### Parameters

**RNNDataDesc**

*Input/Output*. A previously created RNN descriptor. For more information, see cudnnRNNDataDescriptor_t.

**dataType**

*Input*. The datatype of the RNN data tensor. For more information, see cudnnDataType_t.

**layout**

*Input*. The memory layout of the RNN data tensor.

**maxSeqLength**

*Input*. The maximum sequence length within this RNN data tensor. In the unpacked (padded) layout, this should include the padding vectors in each sequence. In the packed (unpadded) layout, this should be equal to the greatest element in **seqLengthArray**.

**batchSize**

*Input*. The number of sequences within the mini-batch.

**vectorSize**

*Input*. The vector length (embedding size) of the input or output tensor at each time-step.

**seqLengthArray**

*Input*. An integer array with **batchSize** number of elements. Describes the length (number of time-steps) of each sequence. Each element in **seqLengthArray** must be greater than 0 but less than or equal to **maxSeqLength**. In the packed layout, the elements should be sorted in descending order, similar to the layout required by the non-extended RNN compute functions.

**paddingFill**

*Input*. A user-defined symbol for filling the padding position in RNN output. This is only effective when the descriptor is describing the RNN output, and the unpacked layout is specified. The symbol should be in the host memory, and is interpreted as the same data type as that of the RNN data tensor. If a **NULL** pointer is passed in, then the padding position in the output will be undefined.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

**dataType** is not one of **CUDNN_DATA_HALF**, **CUDNN_DATA_FLOAT** or **CUDNN_DATA_DOUBLE**.

**CUDNN_STATUS_BAD_PARAM**

Any one of these have occurred:

- **RNNDataDesc** is **NULL**.
- Any one of **maxSeqLength**, **batchSize** or **vectorSize** is less than or equal to zero.
- An element of **seqLengthArray** is less than or equal to zero or greater than **maxSeqLength**.
- Layout is not one of **CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED**, **CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_PACKED** or **CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED**.

**CUDNN_STATUS_ALLOC_FAILED**

The allocation of internal array storage has failed.

## 4.198. cudnnSetRNNDescriptor

```
cudnnStatus_t cudnnSetRNNDescriptor(
    cudnnHandle_t                handle,
    cudnnRNNDescriptor_t         rnnDesc,
    int                          hiddenSize,
    int                          numLayers,
    cudnnDropoutDescriptor_t     dropoutDesc,
    cudnnRNNInputMode_t          inputMode,
```

```
    cudnnDirectionMode_t          direction,
    cudnnRNNMode_t                mode,
    cudnnRNNAlgo_t                algo,
    cudnnDataType_t               mathPrec)
```

This function initializes a previously created RNN descriptor object.

Larger networks, for example, longer sequences or more layers, are expected to be more efficient than smaller networks.

### Parameters

**rnnDesc**

*Input/Output*. A previously created RNN descriptor.

**hiddenSize**

*Input*. Size of the internal hidden state for each layer.

**numLayers**

*Input*. Number of stacked layers.

**dropoutDesc**

*Input*. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers; a single layer network will have no dropout applied.

**inputMode**

*Input*. Specifies the behavior at the input to the first layer.

**direction**

*Input*. Specifies the recurrence pattern, for example, bidirectional.

**mode**

*Input*. Specifies the type of RNN to compute.

**mathPrec**

*Input*. Math precision. This parameter is used for controlling the math precision in RNN. The following applies:

- For the input/output in FP16, the parameter **mathPrec** can be **CUDNN_DATA_HALF or CUDNN_DATA_FLOAT**.
- For the input/output in FP32, the parameter **mathPrec** can only be **CUDNN_DATA_FLOAT**.
- For the input/output in FP64, double type, the parameter **mathPrec** can only be **CUDNN_DATA_DOUBLE**.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

Either at least one of the parameters **hiddenSize** or **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, or **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

## 4.199. cudnnSetRNNDescriptor_v5

```
cudnnStatus_t cudnnSetRNNDescriptor_v5(
    cudnnRNNDescriptor_t     rnnDesc,
    int                      hiddenSize,
    int                      numLayers,
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnRNNInputMode_t      inputMode,
    cudnnDirectionMode_t     direction,
    cudnnRNNMode_t           mode,
    cudnnDataType_t          mathPrec)
```

This function initializes a previously created RNN descriptor object.

Larger networks, for example, longer sequences or more layers, are expected to be more efficient than smaller networks.

### Parameters

**rnnDesc**

*Input/Output*. A previously created RNN descriptor.

**hiddenSize**

*Input*. Size of the internal hidden state for each layer.

**numLayers**

*Input*. Number of stacked layers.

**dropoutDesc**

*Input*. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers, for example, a single layer network will have no dropout applied.

**inputMode**

*Input*. Specifies the behavior at the input to the first layer

**direction**

*Input*. Specifies the recurrence pattern, for example, bidirectional.

**mode**

*Input*. Specifies the type of RNN to compute.

**mathPrec**

*Input*. Math precision. This parameter is used for controlling the math precision in RNN. The following applies:

- For the input/output in FP16, the parameter **mathPrec** can be **CUDNN_DATA_HALF** or **CUDNN_DATA_FLOAT**.
- For the input/output in FP32, the parameter **mathPrec** can only be **CUDNN_DATA_FLOAT**.
- For the input/output in FP64, double type, the parameter **mathPrec** can only be **CUDNN_DATA_DOUBLE**.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

Either at least one of the parameters **hiddenSize** or **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, **algo** or **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

## 4.200. cudnnSetRNNDescriptor_v6

```
cudnnStatus_t cudnnSetRNNDescriptor_v6(
 cudnnHandle_t                      handle,
 cudnnRNNDescriptor_t               rnnDesc,
 const int                          hiddenSize,
 const int                          numLayers,
 cudnnDropoutDescriptor_t           dropoutDesc,
 cudnnRNNInputMode_t                inputMode,
 cudnnDirectionMode_t               direction,
 cudnnRNNMode_t                     mode,
 cudnnRNNAlgo_t                     algo,
 cudnnDataType_t                    mathPrec)
```

This function initializes a previously created RNN descriptor object.

Larger networks, for example, longer sequences or more layers, are expected to be more efficient than smaller networks.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input/Output*. A previously created RNN descriptor.

**hiddenSize**

*Input*. Size of the internal hidden state for each layer.

**numLayers**

*Input*. Number of stacked layers.

**dropoutDesc**

*Input*. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers, for example, a single layer network will have no dropout applied.

**inputMode**

*Input*. Specifies the behavior at the input to the first layer

**direction**

*Input*. Specifies the recurrence pattern, for example, bidirectional.

**mode**

*Input*. Specifies the type of RNN to compute.

**algo**

*Input*. Specifies which RNN algorithm should be used to compute the results.

**mathPrec**

*Input*. Math precision. This parameter is used for controlling the math precision in RNN. The following applies:

- For the input/output in FP16, the parameter **mathPrec** can be **CUDNN_DATA_HALF** or **CUDNN_DATA_FLOAT**.
- For the input/output in FP32, the parameter **mathPrec** can only be **CUDNN_DATA_FLOAT**.
- For the input/output in FP64, double type, the parameter **mathPrec** can only be **CUDNN_DATA_DOUBLE**.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

Either at least one of the parameters **hiddenSize** or **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, **algo** or **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

# 4.201. cudnnSetRNNMatrixMathType

```
cudnnStatus_t cudnnSetRNNMatrixMathType(
    cudnnRNNDescriptor_t    rnnDesc,
    cudnnMathType_t         mType)
```

This function sets the preferred option to use NVIDIA Tensor Cores accelerators on Volta GPUs (SM 7.0 or higher). When the **mType** parameter is **CUDNN_TENSOR_OP_MATH**, inference and training RNN APIs will attempt use Tensor Cores when weights/biases are of type **CUDNN_DATA_HALF** or **CUDNN_DATA_FLOAT**. When RNN weights/biases are stored in the **CUDNN_DATA_FLOAT** format, the original weights and intermediate results will be down-converted to **CUDNN_DATA_HALF** before they are used in another recursive iteration.

### Parameters

**rnnDesc**

*Input*. A previously created and initialized RNN descriptor.

**mType**

*Input*. A preferred compute option when performing RNN GEMMs (general matrix-matrix multiplications). This option has an advisory status meaning that Tensor Cores may not be utilized, for example, due to specific GEMM dimensions.

### Returns

**CUDNN_STATUS_SUCCESS**

The preferred compute option for the RNN network was set successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input parameter was detected.

# 4.202. cudnnSetRNNPaddingMode

```
cudnnStatus_t cudnnSetRNNPaddingMode(
    cudnnRNNDescriptor_t        rnnDesc,
    cudnnRNNPaddingMode_t       paddingMode)
```

This function enables or disables the padded RNN input/output for a previously created and initialized RNN descriptor. This information is required before calling the **cudnnGetRNNWorkspaceSize** and **cudnnGetRNNTrainingReserveSize** functions,

to determine whether additional workspace and training reserve space is needed. By default, the padded RNN input/output is not enabled.

### Parameters

**`rnnDesc`**

*Input/Output*. A previously created RNN descriptor.

**`paddingMode`**

*Input*. Enables or disables the padded input/output. For more information, see cudnnRNNPaddingMode_t.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The **`paddingMode`** was set successfully.

**`CUDNN_STATUS_BAD_PARAM`**

Either the **`rnnDesc`** is **`NULL`** or **`paddingMode`** has an invalid enumerant value.

# 4.203. cudnnSetRNNProjectionLayers

```
cudnnStatus_t cudnnSetRNNProjectionLayers(
    cudnnHandle_t           handle,
    cudnnRNNDescriptor_t    rnnDesc,
    int                     recProjSize,
    int                     outProjSize)
```

The **`cudnnSetRNNProjectionLayers()`** function should be called after **`cudnnSetRNNDescriptor()`** to enable the recurrent and/or output projection in a recursive neural network. The recurrent projection is an additional matrix multiplication in the LSTM cell to project hidden state vectors $h_t$ into smaller vectors $r_t = W_r h_t$, where $W_r$ is a rectangular matrix with **`recProjSize`** rows and **`hiddenSize`** columns. When the recurrent projection is enabled, the output of the LSTM cell (both to the next layer and unrolled in-time) is $r_t$ instead of $h_t$. The dimensionality of $i_t$, $f_t$, $o_t$, and $c_t$ vectors used in conjunction with non-linear functions remains the same as in the canonical LSTM cell. To make this possible, the shapes of matrices in the LSTM formulas (see cudnnRNNMode_t type), such as $W_i$ in hidden RNN layers or $R_i$ in the entire network, become rectangular versus square in the canonical LSTM mode. Obviously, the result of $R_i$* $W_r$ is a square matrix but it is rank deficient, reflecting the compression of LSTM output. The recurrent projection is typically employed when the number of independent (adjustable) weights in the RNN network with projection is smaller in comparison to canonical LSTM for the same **`hiddenSize`** value.

The recurrent projection can be enabled for LSTM cells and **`CUDNN_RNN_ALGO_STANDARD`** only. The **`recProjSize`** parameter should be smaller than the **`hiddenSize`** value programmed in the **`cudnnSetRNNDescriptor()`** call. It is legal to set **`recProjSize`** equal to **`hiddenSize`** but in that case the recurrent projection feature is disabled.

The output projection is currently not implemented.

For more information on the recurrent and output RNN projections, see the paper by Hasim Sak, *et al.*: Long Short-Term Memory Based Recurrent Neural Network Architectures For Large Vocabulary Speech Recognition.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input*. A previously created and initialized RNN descriptor.

**recProjSize**

*Input*. The size of the LSTM cell output after the recurrent projection. This value should not be larger than **hiddenSize** programmed via **cudnnSetRNNDescriptor()**.

**outProjSize**

*Input*. This parameter should be zero.

### Returns

**CUDNN_STATUS_SUCCESS**

RNN projection parameters were set successfully.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was detected (for example, **NULL** handles, negative values for projection parameters).

**CUDNN_STATUS_NOT_SUPPORTED**

Projection applied to RNN algo other than **CUDNN_RNN_ALGO_STANDARD**, cell type other than **CUDNN_LSTM**, **recProjSize** larger than **hiddenSize**.

# 4.204. cudnnSetSeqDataDescriptor

```
cudnnStatus_t cudnnSetSeqDataDescriptor(
 cudnnSeqDataDescriptor_t seqDataDesc,
    cudnnDataType_t dataType,
 int nbDims,
 const int dimA[],
 const cudnnSeqDataAxis_t axes[],
 size_t seqLengthArraySize,
 const int seqLengthArray[],
 void *paddingFill);
```

This function initializes a previously created sequence data descriptor object. In the most simplified view, this descriptor defines dimensions (**dimA**) and the data layout (**axes**)

of a four-dimensional tensor. All four dimensions of the sequence data descriptor have unique identifiers that can be used to index the **dimA[]** array:

```
CUDNN_SEQDATA_TIME_DIM
CUDNN_SEQDATA_BATCH_DIM
CUDNN_SEQDATA_BEAM_DIM
CUDNN_SEQDATA_VECT_DIM
```

For example, to express information that vectors in our sequence data buffer are five elements long, we need to assign **dimA[CUDNN_SEQDATA_VECT_DIM]=5** in the **dimA[]** array.

The number of active dimensions in the **dimA[]** and **axes[]** arrays is defined by the **nbDims** argument. Currently, the value of this argument should be four. The actual size of the **dimA[]** and **axes[]** arrays should be declared using the **CUDNN_SEQDATA_DIM_COUNT** macro.

The **cudnnSeqDataDescriptor_t** container is treated as a collection of fixed length vectors that form sequences, similarly to words (vectors of characters) constructing sentences. The **TIME** dimension spans the sequence length. Different sequences are bundled together in a batch. A **BATCH** may be a group of individual sequences or beams. A **BEAM** is a cluster of alternative sequences or candidates. When thinking about the beam, consider a translation task from one language to another. You may want to keep around and experiment with several translated versions of the original sentence before selecting the best one. The number of candidates kept around is the **BEAM** size.

Every sequence can have a different length, even within the same beam, so vectors toward the end of the sequence can be just padding. The **paddingFill** argument specifies how the padding vectors should be written in output sequence data buffers. The **paddingFill** argument points to one value of type **dataType** that should be copied to all elements in padding vectors. Currently, the only supported value for **paddingFill** is **NULL** which means this option should be ignored. In this case, elements of the padding vectors in output buffers will have undefined values.

It is assumed that a non-empty sequence always starts from the time index zero. The **seqLengthArray[]** must specify all sequence lengths in the container so the total size of this array should be **dimA[CUDNN_SEQDATA_BATCH_DIM] * dimA[CUDNN_SEQDATA_BEAM_DIM]**. Each element of the **seqLengthArray[]** array should have a non-negative value, less than or equal to **dimA[CUDNN_SEQDATA_TIME_DIM**; the maximum sequence length. Elements in **seqLengthArray[]** are always arranged in the same batch-major order, meaning, when considering **BEAM** and **BATCH** dimensions, **BATCH** is the outer or the slower changing index when we traverse the array in ascending order of the addresses. Using a simple example, the **seqLengthArray[]** array should hold sequence lengths in the following order:

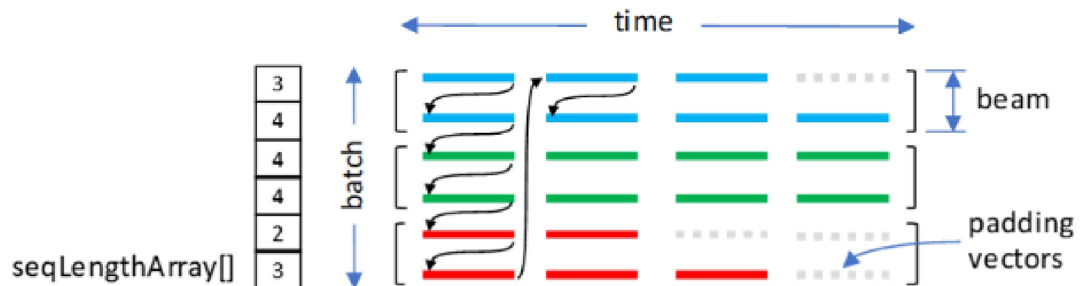
```
{batch_idx=0, beam_idx=0}
{batch_idx=0, beam_idx=1}
{batch_idx=1, beam_idx=0}
{batch_idx=1, beam_idx=1}
{batch_idx=2, beam_idx=0}
{batch_idx=2, beam_idx=1}
```

when **dimA[CUDNN_SEQDATA_BATCH_DIM]=3** and **dimA[CUDNN_SEQDATA_BEAM_DIM]=2**.

Data stored in the **cudnnSeqDataDescriptor_t** container must comply with the following constraints:

- All data are fully packed. There are no unused spaces or gaps between individual vector elements or consecutive vectors.
- The most inner dimension of the container is vector. In other words, the first contiguous group of **dimA[CUDNN_SEQDATA_VECT_DIM]** elements belongs to the first vector, followed by elements of the second vector, and so on.

The **axes** argument in the **cudnnSetSeqDataDescriptor()** function is a bit more complicated. This array should have the same capacity as **dimA[]**. The **axes[]** array specifies the actual data layout in the GPU memory. In this function, the layout is described in the following way: as we move from one element of a vector to another in memory by incrementing the element pointer, what is the order of **VECT**, **TIME**, **BATCH**, and **BEAM** dimensions that we encounter. Let us assume that we want to define the following data layout:




that corresponds to tensor dimensions:

```
int dimA[CUDNN_SEQDATA_DIM_COUNT];
dimA[CUDNN_SEQDATA_TIME_DIM]  = 4;
dimA[CUDNN_SEQDATA_BATCH_DIM] = 3;
dimA[CUDNN_SEQDATA_BEAM_DIM]  = 2;
dimA[CUDNN_SEQDATA_VECT_DIM]  = 5;
```

Now, let's initialize the **axes[]** array. Note that the most inner dimension is described by the last active element of **axes[]**. There is only one valid configuration here as we always traverse a full vector first. Thus, we need to write **CUDNN_SEQDATA_VECT_DIM** in the last active element of **axes[]**.

```
cudnnSeqDataAxis_t axes[CUDNN_SEQDATA_DIM_COUNT];
axes[3] = CUDNN_SEQDATA_VECT_DIM;   // 3 = nbDims-1
```

Now, let's work on the remaining three elements of **axes[]**. When we reach the end of the first vector, we jump to the next beam, therefore:

```
axes[2] = CUDNN_SEQDATA_BEAM_DIM;
```

When we approach the end of the second vector, we move to the next batch, therefore:

```
axes[1] = CUDNN_SEQDATA_BATCH_DIM;
```

The last (outermost) dimension is **TIME**:

```
axes[0] = CUDNN_SEQDATA_TIME_DIM;
```

The four values of the **axes[]** array fully describe the data layout depicted in the figure.

The sequence data descriptor allows the user to select **3! = 6** different data layouts or permutations of **BEAM**, **BATCH** and **TIME** dimensions. The multi-head attention API supports all six layouts.

### Parameters

**seqDataDesc**

*Output.* Pointer to a previously created sequence data descriptor.

**dataType**

*Input.* Data type of the sequence data buffer (**CUDNN_DATA_HALF**, **CUDNN_DATA_FLOAT** or **CUDNN_DATA_DOUBLE**).

**nbDims**

*Input.* Must be **4**. The number of active dimensions in **dimA[]** and **axes[]** arrays. Both arrays should be declared to contain at least **CUDNN_SEQDATA_DIM_COUNT** elements.

**dimA[]**

*Input.* Integer array specifying sequence data dimensions. Use the **cudnnSeqDataAxis_t** enumerated type to index all active **dimA[]** elements.

**axes[]**

*Input.* Array of **cudnnSeqDataAxis_t** that defines the layout of sequence data in memory. The first **nbDims** elements of **axes[]** should be initialized with the outermost dimension in **axes[0]** and the innermost dimension in **axes[nbDims-1]**.

**seqLengthArraySize**

*Input.* Number of elements in the sequence length array, **seqLengthArray[]**.

**seqLengthArray[]**

*Input.* An integer array that defines all sequence lengths of the container.

**paddingFill**

*Input.* Must be **NULL**. Pointer to a value of **dataType** that is used to fill up output vectors beyond the valid length of each sequence or **NULL** to ignore this setting.

### Returns

**CUDNN_STATUS_SUCCESS**

All input arguments were validated and the sequence data descriptor was successfully updated.

**CUDNN_STATUS_BAD_PARAM**

An invalid input argument was found. Some examples include:

- **seqDataDesc=NULL**
- **dateType** was not a valid type of **cudnnDataType_t**
- **nbDims** was negative or zero
- **seqLengthArraySize** did not match the expected length
- some elements of **seqLengthArray[]** were invalid

**CUDNN_STATUS_NOT_SUPPORTED**

An unsupported input argument was encountered. Some examples include:

- **nbDims** is not equal to **4**
- **paddingFill** is not **NULL**

**CUDNN_STATUS_ALLOC_FAILED**

Failed to allocate storage for the sequence data descriptor object.

## 4.205. cudnnSetSpatialTransformerNdDescriptor

```
cudnnStatus_t cudnnSetSpatialTransformerNdDescriptor(
    cudnnSpatialTransformerDescriptor_t     stDesc,
    cudnnSamplerType_t                      samplerType,
    cudnnDataType_t                         dataType,
    const int                               nbDims,
    const int                               dimA[])
```

This function initializes a previously created generic spatial transformer descriptor object.

### Parameters

**stDesc**

*Input/Output*. Previously created spatial transformer descriptor object.

**samplerType**

*Input*. Enumerant to specify the sampler type.

**dataType**

*Input*. Data type.

**nbDims**

*Input*. Dimension of the transformed tensor.

**dimA**

*Input*. Array of dimension **nbDims** containing the size of the transformed tensor for every dimension.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- Either **stDesc** or **dimA** is **NULL**.
- Either **dataType** or **samplerType** has an invalid enumerant value

# 4.206. cudnnSetStream

```
cudnnStatus_t cudnnSetStream(
    cudnnHandle_t   handle,
    cudaStream_t    streamId)
```

This function sets the user's CUDA stream in the cuDNN handle. The new stream will be used to launch cuDNN GPU kernels or to synchronize to this stream when cuDNN kernels are launched in the internal streams. If the cuDNN library stream is not set, all kernels use the default (**NULL**) stream. Setting the user stream in the cuDNN handle guarantees the issue-order execution of cuDNN calls and other GPU kernels launched in the same stream.

### Parameters

**handle**

*Input*. Pointer to the cuDNN handle.

**streamID**

*Input*. New CUDA stream to be written to the cuDNN handle.

### Returns

**CUDNN_STATUS_BAD_PARAM**

Invalid (**NULL**) handle.

**CUDNN_STATUS_MAPPING_ERROR**

Mismatch between the user stream and the cuDNN handle context.

**CUDNN_STATUS_SUCCESS**

The new stream was set successfully.

# 4.207. cudnnSetTensor

```
cudnnStatus_t cudnnSetTensor(
    cudnnHandle_t                   handle,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y,
    const void                     *valuePtr)
```

This function sets all the elements of a tensor to a given value.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor.

**y**

*Input/Output*. Pointer to data of the tensor described by the **yDesc** descriptor.

**valuePtr**

*Input*. Pointer in host memory to a single value. All elements of the **y** tensor will be set to **value[0]**. The data type of the element in **value[0]** has to match the data type of tensor **y**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

One of the provided pointers is nil.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.208. cudnnSetTensor4dDescriptor

```
cudnnStatus_t cudnnSetTensor4dDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                     n,
    int                     c,
    int                     h,
    int                     w)
```

This function initializes a previously created generic Tensor descriptor object into a 4D tensor. The strides of the four dimensions are inferred from the format parameter and set in such a way that the data is contiguous in memory with no padding between dimensions.

The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type **datatype**.

### Parameters

**tensorDesc**

*Input/Output*. Handle to a previously created tensor descriptor.

**format**

*Input*. Type of format.

**datatype**

*Input*. Data type.

**n**

*Input*. Number of images.

**c**

*Input*. Number of feature maps per image.

**h**

*Input*. Height of each feature map.

**w**

*Input*. Width of each feature map.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the parameters **n**, **c**, **h**, **w** was negative or **format** has an invalid enumerant value or **dataType** has an invalid enumerant value.

**CUDNN_STATUS_NOT_SUPPORTED**

The total size of the tensor descriptor exceeds the maximum limit of 2 Giga-elements.

## 4.209. cudnnSetTensor4dDescriptorEx

```
cudnnStatus_t cudnnSetTensor4dDescriptorEx(
    cudnnTensorDescriptor_t     tensorDesc,
    cudnnDataType_t             dataType,
    int                         n,
    int                         c,
    int                         h,
    int                         w,
    int                         nStride,
    int                         cStride,
    int                         hStride,
    int                         wStride)
```

This function initializes a previously created generic tensor descriptor object into a 4D tensor, similarly to **cudnnSetTensor4dDescriptor** but with the strides explicitly

passed as parameters. This can be used to lay out the 4D tensor in any order or simply to define gaps between dimensions.

- At present, some cuDNN routines have limited support for strides. Those routines will return **`CUDNN_STATUS_NOT_SUPPORTED`** if a 4D tensor object with an unsupported stride is used. **`cudnnTransformTensor`** can be used to convert the data to a supported layout.
- The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type **`datatype`**.

### Parameters

**`tensorDesc`**

*Input/Output*. Handle to a previously created tensor descriptor.

**`datatype`**

*Input*. Data type.

**`n`**

*Input*. Number of images.

**`c`**

*Input*. Number of feature maps per image.

**`h`**

*Input*. Height of each feature map.

**`w`**

*Input*. Width of each feature map.

**`nStride`**

*Input*. Stride between two consecutive images.

**`cStride`**

*Input*. Stride between two consecutive feature maps.

**`hStride`**

*Input*. Stride between two consecutive rows.

**`wStride`**

*Input*. Stride between two consecutive columns.

### Returns

**`CUDNN_STATUS_SUCCESS`**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the parameters **n**, **c**, **h**, **w** or **nStride**, **cStride**, **hStride**, **wStride** is negative or **dataType** has an invalid enumerant value.

**CUDNN_STATUS_NOT_SUPPORTED**

The total size of the tensor descriptor exceeds the maximum limit of 2 Giga-elements.

# 4.210. cudnnSetTensorNdDescriptor

```
cudnnStatus_t cudnnSetTensorNdDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnDataType_t         dataType,
    int                     nbDims,
    const int               dimA[],
    const int               strideA[])
```

This function initializes a previously created generic tensor descriptor object.

The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type **datatype**. Tensors are restricted to having at least 4 dimensions, and at most **CUDNN_DIM_MAX** dimensions (defined in **cudnn.h**). When working with lower dimensional data, it is recommended that the user create a 4D tensor, and set the size along unused dimensions to 1.

### Parameters

**tensorDesc**

*Input/Output*. Handle to a previously created tensor descriptor.

**datatype**

*Input*. Data type.

**nbDims**

*Input*. Dimension of the tensor.

Do not use 2 dimensions. Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. For more information, see cudnnGetRNNLinLayerBiasParams.

**dimA**

*Input*. Array of dimension **nbDims** that contain the size of the tensor for every dimension. Size along unused dimensions should be set to 1.

**strideA**

*Input*. Array of dimension **nbDims** that contain the stride of the tensor for every dimension.

### Returns

**CUDNN_STATUS_SUCCESS**

The object was set successfully.

**CUDNN_STATUS_BAD_PARAM**

At least one of the elements of the array **dimA** was negative or zero, or **dataType** has an invalid enumerant value.

**CUDNN_STATUS_NOT_SUPPORTED**

The parameter **nbDims** is outside the range **[4, CUDNN_DIM_MAX]**, or the total size of the tensor descriptor exceeds the maximum limit of 2 Giga-elements.

## 4.211. cudnnSetTensorNdDescriptorEx

```
cudnnStatus_t cudnnSetTensorNdDescriptorEx(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                     nbDims,
    const int               dimA[])
```

This function initializes an n-D tensor descriptor.

### Parameters

**tensorDesc**

*Output*. Pointer to the tensor descriptor struct to be initialized.

**format**

*Input*. Tensor format.

**dataType**

*Input*. Tensor data type.

**nbDims**

*Input*. Dimension of the tensor.

Do not use 2 dimensions. Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. For more information, see cudnnGetRNNLinLayerBiasParams.

**dimA**

*Input*. Array containing the size of each dimension.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was successful.

**CUDNN_STATUS_BAD_PARAM**

Tensor descriptor was not allocated properly; or input parameters are not set correctly.

**CUDNN_STATUS_NOT_SUPPORTED**

Dimension size requested is larger than maximum dimension size supported.

## 4.212. cudnnSetTensorTransformDescriptor

```
cudnnStatus_t cudnnSetTensorTransformDescriptor(
 cudnnTensorTransformDescriptor_t transformDesc,
 const uint32_t nbDims,
 const cudnnTensorFormat_t destFormat,
 const int32_t padBeforeA[],
 const int32_t padAfterA[],
 const uint32_t foldA[],
 const cudnnFoldingDirection_t direction);
```

his function initializes a tensor transform descriptor that was previously created using the cudnnCreateTensorTransformDescriptor function.

### Parameters

**transformDesc**

*Output*. The tensor transform descriptor to be initialized.

**nbDims**

*Input*. The dimensionality of the transform operands. Must be greater than 2. For more information, see the Tensor Descriptor section from the *cuDNN Developer Guide*.

**destFormat**

*Input*. The desired destination format.

**padBeforeA[]**

*Input*. An array that contains the amount of padding that should be added before each dimension. Set to **NULL** for no padding.

**padAfterA[]**

*Input*. An array that contains the amount of padding that should be added after each dimension. Set to **NULL** for no padding.

**foldA[]**

*Input*. An array that contains the folding parameters for each spatial dimension (dimensions 2 and up). Set to **NULL** for no folding.

**direction**

*Input*. Selects folding or unfolding. This input has no effect when folding parameters are all <= 1. For more information, see cudnnFoldingDirection_t.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

The parameter **transformDesc** is **NULL**, or if **direction** is invalid, or **nbDims** is <= 2.

**CUDNN_STATUS_NOT_SUPPORTED**

If the dimension size requested is larger than maximum dimension size supported (meaning, one of the **nbDims** is larger than **CUDNN_DIM_MAX**), or if **destFromat** is something other than **NCHW** or **NHWC**.

## 4.213. cudnnSoftmaxBackward

```
cudnnStatus_t cudnnSoftmaxBackward(
    cudnnHandle_t                      handle,
    cudnnSoftmaxAlgorithm_t            algorithm,
    cudnnSoftmaxMode_t                 mode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      yDesc,
    const void                        *yData,
    const cudnnTensorDescriptor_t      dyDesc,
    const void                        *dy,
    const void                        *beta,
    const cudnnTensorDescriptor_t      dxDesc,
    void                              *dx)
```

This routine computes the gradient of the softmax function.

- In-place operation is allowed for this routine; meaning, **dy** and **dx** pointers may be equal. However, this requires **dyDesc** and **dxDesc** descriptors to be identical (particularly, the strides of the input and output must match for in-place operation to be allowed).
- All tensor formats are supported for all modes and algorithms with 4 and 5D tensors. Performance is expected to be highest with **NCHW fully-packed** tensors. For more than 5 dimensions tensors must be packed in their spatial dimensions.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**algorithm**

*Input*. Enumerant to specify the softmax algorithm.

**mode**

*Input*. Enumerant to specify the softmax mode.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**yDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**y**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **yDesc**.

**dyDesc**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**dy**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **dyData**.

**dxDesc**

*Input*. Handle to the previously initialized output differential tensor descriptor.

**dx**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **dxDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions **n**, **c**, **h**, **w** of the **yDesc**, **dyDesc** and **dxDesc** tensors differ.
- The strides **nStride**, **cStride**, **hStride**, **wStride** of the **yDesc** and **dyDesc** tensors differ.
- The **datatype** of the three tensors differs.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.214. cudnnSoftmaxForward

```
cudnnStatus_t cudnnSoftmaxForward(
    cudnnHandle_t                    handle,
    cudnnSoftmaxAlgorithm_t          algorithm,
    cudnnSoftmaxMode_t               mode,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This routine computes the softmax function.

All tensor formats are supported for all modes and algorithms with 4 and 5D tensors. Performance is expected to be highest with **NCHW fully-packed** tensors. For more than 5 dimensions tensors must be packed in their spatial dimensions

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**algorithm**

*Input*. Enumerant to specify the softmax algorithm.

**mode**

*Input*. Enumerant to specify the softmax mode.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows:

```
dstValue = alpha[0]*result + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- The dimensions **n**, **c**, **h**, **w** of the input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differ.
- The parameters **algorithm** or **mode** have an invalid enumerant value.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.215. cudnnSpatialTfGridGeneratorBackward

```
cudnnStatus_t cudnnSpatialTfGridGeneratorBackward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t   stDesc,
    const void                                 *dgrid,
    void                                       *dtheta)
```

This function computes the gradient of a grid generation operation.

Only 2d transformation is supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**stDesc**

*Input*. Previously created spatial transformer descriptor object.

**dgrid**

*Input*. Data pointer to GPU memory contains the input differential data.

**dtheta**

*Output*. Data pointer to GPU memory contains the output differential data.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is **NULL**.
- One of the parameters **dgrid** or **dtheta** is **NULL**.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimension of transformed tensor specified in **stDesc** > 4.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.216. cudnnSpatialTfGridGeneratorForward

```
cudnnStatus_t cudnnSpatialTfGridGeneratorForward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t  stDesc,
    const void                                *theta,
    void                                      *grid)
```

This function generates a grid of coordinates in the input tensor corresponding to each pixel from the output tensor.

Only 2d transformation is supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**stDesc**

*Input*. Previously created spatial transformer descriptor object.

**theta**

*Input*. Affine transformation matrix. It should be of size **n*2*3** for a 2d transformation, where **n** is the number of images specified in **stDesc**.

**grid**

*Output*. A grid of coordinates. It is of size **n*h*w*2** for a 2d transformation, where **n**, **h**, **w** is specified in **stDesc**. In the 4th dimension, the first coordinate is **x**, and the second coordinate is **y**.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is **NULL**.
- One of the parameters **grid** or **theta** is **NULL**.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimension of transformed tensor specified in **stDesc** > 4.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.217. cudnnSpatialTfSamplerBackward

```
cudnnStatus_t cudnnSpatialTfSamplerBackward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t  stDesc,
    const void                                *alpha,
    const cudnnTensorDescriptor_t              xDesc,
    const void                                *x,
    const void                                *beta,
    const cudnnTensorDescriptor_t              dxDesc,
    void                                      *dx,
    const void                                *alphaDgrid,
    const cudnnTensorDescriptor_t              dyDesc,
    const void                                *dy,
    const void                                *grid,
    const void                                *betaDgrid,
    void                                      *dgrid)
```

This function computes the gradient of a sampling operation.

Only 2d transformation is supported.

### Parameters

**`handle`**

*Input*. Handle to a previously created cuDNN context.

**`stDesc`**

*Input*. Previously created spatial transformer descriptor object.

**`alpha, beta`**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*srcValue + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**`xDesc`**

*Input*. Handle to the previously initialized input tensor descriptor.

**`x`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`xDesc`**.

**`dxDesc`**

*Input*. Handle to the previously initialized output differential tensor descriptor.

**`dx`**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **`dxDesc`**.

**`alphaDgrid, betaDgrid`**

*Input*. Pointers to scaling factors (in host memory) used to blend the gradient outputs dgrid with prior value in the destination pointer as follows:

```
dstValue = alpha[0]*srcValue + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**`dyDesc`**

*Input*. Handle to the previously initialized input differential tensor descriptor.

**`dy`**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **`dyDesc`**.

**grid**

*Input*. A grid of coordinates generated by **cudnnSpatialTfGridGeneratorForward**.

**dgrid**

*Output*. Data pointer to GPU memory contains the output differential data.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is **NULL**.
- One of the parameters **x**, **dx**, **y**, **dy**, **grid**, **dgrid** is **NULL**.
- The dimension of **dy** differs from those specified in **stDesc**.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimension of transformed tensor > 4.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.218. cudnnSpatialTfSamplerForward

```
cudnnStatus_t cudnnSpatialTfSamplerForward(
    cudnnHandle_t                                 handle,
    const cudnnSpatialTransformerDescriptor_t     stDesc,
    const void                                   *alpha,
    const cudnnTensorDescriptor_t                 xDesc,
    const void                                   *x,
    const void                                   *grid,
    const void                                   *beta,
    cudnnTensorDescriptor_t                       yDesc,
    void                                         *y)
```

This function performs a sampler operation and generates the output tensor using the grid given by the grid generator.

Only 2d transformation is supported.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**stDesc**

*Input*. Previously created spatial transformer descriptor object.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*srcValue + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to the previously initialized input tensor descriptor.

**x**

*Input*. Data pointer to GPU memory associated with the tensor descriptor **xDesc**.

**grid**

*Input*. A grid of coordinates generated by **cudnnSpatialTfGridGeneratorForward**.

**yDesc**

*Input*. Handle to the previously initialized output tensor descriptor.

**y**

*Output*. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

### Returns

**CUDNN_STATUS_SUCCESS**

The call was successful.

**CUDNN_STATUS_BAD_PARAM**

At least one of the following conditions are met:

- **handle** is **NULL**.
- One of the parameters **x**, **y** or **grid** is **NULL**.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration. See the following for some examples of non-supported configurations:

- The dimension of transformed tensor > 4.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

# 4.219. cudnnTransformTensor

```
cudnnStatus_t cudnnTransformTensor(
    cudnnHandle_t                    handle,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This function copies the scaled data from one tensor to another tensor with a different layout. Those descriptors need to have the same dimensions but not necessarily the same strides. The input and output tensors must not overlap in any way (meaning, tensors cannot be transformed in place). This function can be used to convert a tensor with an unsupported format to a supported one.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context.

**alpha, beta**

*Input*. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows:

```
dstValue = alpha[0]*srcValue + beta[0]*priorDstValue
```

For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

**xDesc**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**x**

*Input*. Pointer to data of the tensor described by the **xDesc** descriptor.

**yDesc**

*Input*. Handle to a previously initialized tensor descriptor. For more information, see cudnnTensorDescriptor_t.

**y**

*Output*. Pointer to data of the tensor described by the **yDesc** descriptor.

### Returns

**CUDNN_STATUS_SUCCESS**

The function launched successfully.

**CUDNN_STATUS_NOT_SUPPORTED**

The function does not support the provided configuration.

**CUDNN_STATUS_BAD_PARAM**

The dimensions **n**, **c**, **h**, **w** or the **dataType** of the two tensor descriptors are different.

**CUDNN_STATUS_EXECUTION_FAILED**

The function failed to launch on the GPU.

## 4.220. cudnnTransformTensorEx

```
cudnnStatus_t cudnnTransformTensorEx(
 cudnnHandle_t handle,
 const cudnnTensorTransformDescriptor_t transDesc,

 const void *alpha,
 const cudnnTensorDescriptor_t srcDesc,
 const void *srcData,
 const void *beta,
 const cudnnTensorDescriptor_t destDesc,
 void *destData);
```

This function converts the tensor layouts between different formats. It can be used to convert a tensor with an unsupported layout format to a tensor with a supported layout format.

This function copies the scaled data from the input tensor **srcDesc** to the output tensor **destDesc** with a different layout. The tensor descriptors of **srcDesc** and **destDesc** should have the same dimensions but need not have the same strides.

The **srcDesc** and **destDesc** tensors must not overlap in any way (meaning, tensors cannot be transformed in place).

When performing a folding transform or a zero-padding transform, the scaling factors **(alpha,beta)** should be set to (1, 0). However, unfolding transforms support any **(alpha,beta)** values. This function is thread safe.

### Parameters

**handle**

*Input*. Handle to a previously created cuDNN context. For more information, see cudnnHandle_t.

**transDesc**

*Input*. A descriptor containing the details of the requested tensor transformation. For more information, see cudnnTensorTransformDescriptor_t.

**alpha, beta**

*Input*. Pointers, in the host memory, to the scaling factors used to scale the data in the input tensor **srcDesc**. **beta** is used to scale the destination tensor, while **alpha** is used to scale the source tensor. For more information, see the Scaling Parameters section in the *cuDNN Developer Guide*.

The beta scaling value is not honored in the folding and zero-padding cases. Unfolding supports any (**alpha**, **beta**).

**srcDesc, destDesc**

*Input*. Handles to the previously initialed tensor descriptors. **srcDesc** and **destDesc** must not overlap. For more information, see cudnnTensorDescriptor_t.

**srcData, destData**

*Input*. Pointers, in the host memory, to the data of the tensor described by **srcDesc** and **destDesc** respectively.

### Returns

**CUDNN_STATUS_SUCCESS**

The function was launched successfully.

**CUDNN_STATUS_BAD_PARAM**

A parameter is uninitialized or initialized incorrectly, or the number of dimensions is different between **srcDesc** and **destDesc**.

**CUDNN_STATUS_NOT_SUPPORTED**

Function does not support the provided configuration. Also, in the folding and padding paths, any value other than **A=1** and **B=0** will result in a **CUDNN_STATUS_NOT_SUPPORTED**.

**CUDNN_STATUS_EXECUTION_FAILED**

Function failed to launch on the GPU.

# Chapter 5. ACKNOWLEDGMENTS

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

## 5.1. 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.
```


## 5.2. 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.
```


## 5.3. 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

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

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

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

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

## Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NvCaffe, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

## Copyright

© 2019 NVIDIA Corporation. All rights reserved.

**www.nvidia.com**