



CUDNN 7.4.1

DU-06702-001_v07 | November 2018

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. Thread Safety.....	7
2.6. Reproducibility (determinism).....	7
2.7. Scaling parameters alpha and beta.....	8
2.8. Tensor Core Operations.....	8
2.8.1. Tensor Core Operations Notes.....	10
2.8.2. Tensor Operations Speedup Tips.....	10
2.9. GPU and driver requirements.....	11
2.10. Backward compatibility and deprecation policy.....	11
2.11. Grouped Convolutions.....	12
2.12. API Logging.....	13
2.13. Features of RNN Functions.....	15
2.14. Mixed Precision Numerical Accuracy.....	17
Chapter 3. cuDNN Datatypes Reference	18
3.1. cudnnActivationDescriptor_t.....	18
3.2. cudnnActivationMode_t.....	18
3.3. cudnnBatchNormMode_t.....	19
3.4. cudnnBatchNormOps_t.....	20
3.5. cudnnCTCLossAlgo_t.....	20
3.6. cudnnCTCLossDescriptor_t.....	20
3.7. cudnnConvolutionBwdDataAlgoPerf_t.....	21
3.8. cudnnConvolutionBwdDataAlgo_t.....	21
3.9. cudnnConvolutionBwdDataPreference_t.....	22
3.10. cudnnConvolutionBwdFilterAlgoPerf_t.....	23
3.11. cudnnConvolutionBwdFilterAlgo_t.....	23
3.12. cudnnConvolutionBwdFilterPreference_t.....	24
3.13. cudnnConvolutionDescriptor_t.....	25
3.14. cudnnConvolutionFwdAlgoPerf_t.....	25

3.15. cudnnConvolutionFwdAlgo_t.....	26
3.16. cudnnConvolutionFwdPreference_t.....	27
3.17. cudnnConvolutionMode_t.....	27
3.18. cudnnDataType_t.....	27
3.19. cudnnDeterminism_t.....	28
3.20. cudnnDirectionMode_t.....	28
3.21. cudnnDivNormMode_t.....	29
3.22. cudnnDropoutDescriptor_t.....	29
3.23. cudnnErrQueryMode_t.....	29
3.24. cudnnFilterDescriptor_t.....	30
3.25. cudnnHandle_t.....	30
3.26. cudnnIndicesType_t.....	30
3.27. cudnnLRNMode_t.....	31
3.28. cudnnMathType_t.....	31
3.29. cudnnNanPropagation_t.....	31
3.30. cudnnOpTensorDescriptor_t.....	31
3.31. cudnnOpTensorOp_t.....	32
3.32. cudnnPersistentRNNPlan_t.....	32
3.33. cudnnPoolingDescriptor_t.....	32
3.34. cudnnPoolingMode_t.....	32
3.35. cudnnRNNAlgo_t.....	33
3.36. cudnnRNNClipMode_t.....	34
3.37. cudnnRNNDescriptor_t.....	34
3.38. cudnnRNNDataDescriptor_t.....	34
3.39. cudnnRNNInputMode_t.....	34
3.40. cudnnRNNMode_t.....	35
3.41. cudnnRNNPaddingMode_t.....	36
3.42. cudnnReduceTensorDescriptor_t.....	36
3.43. cudnnReduceTensorIndices_t.....	36
3.44. cudnnReduceTensorOp_t.....	37
3.45. cudnnSamplerType_t.....	37
3.46. cudnnSoftmaxAlgorithm_t.....	38
3.47. cudnnSoftmaxMode_t.....	38
3.48. cudnnSpatialTransformerDescriptor_t.....	38
3.49. cudnnStatus_t.....	38
3.50. cudnnTensorDescriptor_t.....	40
3.51. cudnnTensorFormat_t.....	40
Chapter 4. cuDNN API Reference.....	42
4.1. cudnnActivationBackward.....	42
4.2. cudnnActivationForward.....	44
4.3. cudnnAddTensor.....	45
4.4. cudnnBatchNormalizationBackward.....	46
4.5. cudnnBatchNormalizationBackwardEx.....	49

4.6. cudnnBatchNormalizationForwardInference.....	52
4.7. cudnnBatchNormalizationForwardTraining.....	54
4.8. cudnnBatchNormalizationForwardTrainingEx.....	57
4.9. cudnnCTCLoss.....	61
4.10. cudnnConvolutionBackwardBias.....	62
4.11. cudnnConvolutionBackwardData.....	63
4.12. cudnnConvolutionBackwardFilter.....	70
4.13. cudnnConvolutionBiasActivationForward.....	76
4.14. cudnnConvolutionForward.....	79
4.15. cudnnCreate.....	86
4.16. cudnnCreateActivationDescriptor.....	87
4.17. cudnnCreateAlgorithmDescriptor.....	87
4.18. cudnnCreateAlgorithmPerformance.....	88
4.19. cudnnCreateCTCLossDescriptor.....	88
4.20. cudnnCreateConvolutionDescriptor.....	88
4.21. cudnnCreateDropoutDescriptor.....	89
4.22. cudnnCreateFilterDescriptor.....	89
4.23. cudnnCreateLRNDescriptor.....	89
4.24. cudnnCreateOpTensorDescriptor.....	90
4.25. cudnnCreatePersistentRNNPlan.....	90
4.26. cudnnCreatePoolingDescriptor.....	91
4.27. cudnnCreateRNNDescriptor.....	91
4.28. cudnnCreateRNNDataDescriptor.....	91
4.29. cudnnCreateReduceTensorDescriptor.....	92
4.30. cudnnCreateSpatialTransformerDescriptor.....	92
4.31. cudnnCreateTensorDescriptor.....	92
4.32. cudnnDeriveBNTensorDescriptor.....	93
4.33. cudnnDestroy.....	94
4.34. cudnnDestroyActivationDescriptor.....	95
4.35. cudnnDestroyAlgorithmDescriptor.....	95
4.36. cudnnDestroyAlgorithmPerformance.....	95
4.37. cudnnDestroyCTCLossDescriptor.....	95
4.38. cudnnDestroyConvolutionDescriptor.....	96
4.39. cudnnDestroyDropoutDescriptor.....	96
4.40. cudnnDestroyFilterDescriptor.....	96
4.41. cudnnDestroyLRNDescriptor.....	96
4.42. cudnnDestroyOpTensorDescriptor.....	97
4.43. cudnnDestroyPersistentRNNPlan.....	97
4.44. cudnnDestroyPoolingDescriptor.....	97
4.45. cudnnDestroyRNNDescriptor.....	97
4.46. cudnnDestroyRNNDataDescriptor.....	98
4.47. cudnnDestroyReduceTensorDescriptor.....	98
4.48. cudnnDestroySpatialTransformerDescriptor.....	98

4.49. cudnnDestroyTensorDescriptor.....	99
4.50. cudnnDivisiveNormalizationBackward.....	99
4.51. cudnnDivisiveNormalizationForward.....	101
4.52. cudnnDropoutBackward.....	103
4.53. cudnnDropoutForward.....	104
4.54. cudnnDropoutGetReserveSpaceSize.....	106
4.55. cudnnDropoutGetStatesSize.....	106
4.56. cudnnFindConvolutionBackwardDataAlgorithm.....	107
4.57. cudnnFindConvolutionBackwardDataAlgorithmEx.....	108
4.58. cudnnFindConvolutionBackwardFilterAlgorithm.....	110
4.59. cudnnFindConvolutionBackwardFilterAlgorithmEx.....	112
4.60. cudnnFindConvolutionForwardAlgorithm.....	114
4.61. cudnnFindConvolutionForwardAlgorithmEx.....	115
4.62. cudnnFindRNNBackwardDataAlgorithmEx.....	117
4.63. cudnnFindRNNBackwardWeightsAlgorithmEx.....	123
4.64. cudnnFindRNNForwardInferenceAlgorithmEx.....	126
4.65. cudnnFindRNNForwardTrainingAlgorithmEx.....	131
4.66. cudnnGetActivationDescriptor.....	135
4.67. cudnnGetAlgorithmDescriptor.....	136
4.68. cudnnGetAlgorithmPerformance.....	136
4.69. cudnnGetAlgorithmSpaceSize.....	137
4.70. cudnnBatchNormalizationBackwardExWorkspaceSize.....	138
4.71. cudnnBatchNormalizationForwardTrainingExWorkspaceSize.....	139
4.72. cudnnGetBatchNormalizationTrainingExReserveSpaceSize.....	140
4.73. cudnnGetCTCLossDescriptor.....	141
4.74. cudnnGetCTCLossWorkspaceSize.....	142
4.75. cudnnGetCallback.....	143
4.76. cudnnGetConvolution2dDescriptor.....	144
4.77. cudnnGetConvolution2dForwardOutputDim.....	145
4.78. cudnnGetConvolutionBackwardDataAlgorithm.....	146
4.79. cudnnGetConvolutionBackwardDataAlgorithmMaxCount.....	147
4.80. cudnnGetConvolutionBackwardDataAlgorithm_v7.....	147
4.81. cudnnGetConvolutionBackwardDataWorkspaceSize.....	149
4.82. cudnnGetConvolutionBackwardFilterAlgorithm.....	150
4.83. cudnnGetConvolutionBackwardFilterAlgorithmMaxCount.....	151
4.84. cudnnGetConvolutionBackwardFilterAlgorithm_v7.....	151
4.85. cudnnGetConvolutionBackwardFilterWorkspaceSize.....	153
4.86. cudnnGetConvolutionForwardAlgorithm.....	154
4.87. cudnnGetConvolutionForwardAlgorithmMaxCount.....	155
4.88. cudnnGetConvolutionForwardAlgorithm_v7.....	156
4.89. cudnnGetConvolutionForwardWorkspaceSize.....	157
4.90. cudnnGetConvolutionGroupCount.....	158
4.91. cudnnGetConvolutionMathType.....	158

4.92. cudnnGetConvolutionNdDescriptor.....	159
4.93. cudnnGetConvolutionNdForwardOutputDim.....	160
4.94. cudnnGetCudartVersion.....	161
4.95. cudnnGetDropoutDescriptor.....	161
4.96. cudnnGetErrorString.....	162
4.97. cudnnGetFilter4dDescriptor.....	162
4.98. cudnnGetFilterNdDescriptor.....	163
4.99. cudnnGetLRNDescriptor.....	164
4.100. cudnnGetOpTensorDescriptor.....	164
4.101. cudnnGetPooling2dDescriptor.....	165
4.102. cudnnGetPooling2dForwardOutputDim.....	166
4.103. cudnnGetPoolingNdDescriptor.....	167
4.104. cudnnGetPoolingNdForwardOutputDim.....	168
4.105. cudnnGetProperty.....	169
4.106. cudnnGetRNNDataDescriptor.....	169
4.107. cudnnGetRNNDescriptor.....	171
4.108. cudnnGetRNNLinLayerBiasParams.....	172
4.109. cudnnGetRNNLinLayerMatrixParams.....	174
4.110. cudnnGetRNNParamsSize.....	176
4.111. cudnnGetRNNPaddingMode.....	177
4.112. cudnnGetRNNProjectionLayers.....	178
4.113. cudnnGetRNNTrainingReserveSize.....	178
4.114. cudnnGetRNNWorkspaceSize.....	179
4.115. cudnnGetReduceTensorDescriptor.....	180
4.116. cudnnGetReductionIndicesSize.....	181
4.117. cudnnGetReductionWorkspaceSize.....	182
4.118. cudnnGetStream.....	182
4.119. cudnnGetTensor4dDescriptor.....	183
4.120. cudnnGetTensorNdDescriptor.....	184
4.121. cudnnGetTensorSizeInBytes.....	185
4.122. cudnnGetVersion.....	185
4.123. cudnnIm2Col.....	186
4.124. cudnnLRNCrossChannelBackward.....	187
4.125. cudnnLRNCrossChannelForward.....	188
4.126. cudnnOpTensor.....	189
4.127. cudnnPoolingBackward.....	191
4.128. cudnnPoolingForward.....	193
4.129. cudnnQueryRuntimeError.....	194
4.130. cudnnRNNBackwardData.....	196
4.131. cudnnRNNBackwardDataEx.....	201
4.132. cudnnRNNBackwardWeights.....	205
4.133. cudnnRNNBackwardWeightsEx.....	208
4.134. cudnnRNNForwardInference.....	210

4.135. cudnnRNNForwardInferenceEx.....	214
4.136. cudnnRNNForwardTraining.....	218
4.137. cudnnRNNForwardTrainingEx.....	221
4.138. cudnnRNNGetClip.....	226
4.139. cudnnRNNSetClip.....	226
4.140. cudnnReduceTensor.....	227
4.141. cudnnRestoreAlgorithm.....	229
4.142. cudnnRestoreDropoutDescriptor.....	230
4.143. cudnnSaveAlgorithm.....	231
4.144. cudnnScaleTensor.....	232
4.145. cudnnSetActivationDescriptor.....	232
4.146. cudnnSetAlgorithmDescriptor.....	233
4.147. cudnnSetAlgorithmPerformance.....	233
4.148. cudnnSetCTCLossDescriptor.....	234
4.149. cudnnSetCallback.....	235
4.150. cudnnSetConvolution2dDescriptor.....	236
4.151. cudnnSetConvolutionGroupCount.....	237
4.152. cudnnSetConvolutionMathType.....	237
4.153. cudnnSetConvolutionNdDescriptor.....	238
4.154. cudnnSetDropoutDescriptor.....	239
4.155. cudnnSetFilter4dDescriptor.....	240
4.156. cudnnSetFilterNdDescriptor.....	241
4.157. cudnnSetLRNDescriptor.....	242
4.158. cudnnSetOpTensorDescriptor.....	243
4.159. cudnnSetPersistentRNNPlan.....	244
4.160. cudnnSetPooling2dDescriptor.....	244
4.161. cudnnSetPoolingNdDescriptor.....	245
4.162. cudnnSetRNNDataDescriptor.....	246
4.163. cudnnSetRNNDescriptor.....	248
4.164. cudnnSetRNNDescriptor_v5.....	249
4.165. cudnnSetRNNDescriptor_v6.....	250
4.166. cudnnSetRNNMatrixMathType.....	251
4.167. cudnnSetRNNPaddingMode.....	252
4.168. cudnnSetRNNProjectionLayers.....	252
4.169. cudnnSetReduceTensorDescriptor.....	254
4.170. cudnnSetSpatialTransformerNdDescriptor.....	255
4.171. cudnnSetStream.....	255
4.172. cudnnSetTensor.....	256
4.173. cudnnSetTensor4dDescriptor.....	257
4.174. cudnnSetTensor4dDescriptorEx.....	258
4.175. cudnnSetTensorNdDescriptor.....	259
4.176. cudnnSetTensorNdDescriptorEx.....	260
4.177. cudnnSoftmaxBackward.....	261

4.178. cudnnSoftmaxForward.....	263
4.179. cudnnSpatialTfGridGeneratorBackward.....	264
4.180. cudnnSpatialTfGridGeneratorForward.....	265
4.181. cudnnSpatialTfSamplerBackward.....	266
4.182. cudnnSpatialTfSamplerForward.....	268
4.183. cudnnTransformTensor.....	269
Chapter 5. Acknowledgments.....	271
5.1. University of Tennessee.....	271
5.2. University of California, Berkeley.....	271
5.3. Facebook AI Research, New York.....	272

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 `cudaDnnCreate()`. 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 `cudaDnnDestroy()`. 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 `cudaDnnCreate()` and `cudaDnnDestroy()` 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 `cudaDnnCreate()`.

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 $y \geq x$)
- ▶ 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

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

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

$$y_{n, k, p, q} = \sum_c^{C_g} \sum_r^R \sum_s^S x_{n, C_g * \text{floor}(k/K_g) + c, p+r, q+s} \times w_{k, c, r, s}$$

2.3. Notation

As of CUDNN v4 we have adopted a mathematically-inspired notation for layer inputs and outputs using \mathbf{x} , \mathbf{y} , $d\mathbf{x}$, $d\mathbf{y}$, \mathbf{b} , \mathbf{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 \mathbf{w} is the matrix of filter weights, \mathbf{x} is the previous layer's data (during inference), \mathbf{y} is the next layer's data, \mathbf{b} is the bias and $*$ is the convolution operator.

In backpropagation routines the parameters keep their meanings.

The parameters $d\mathbf{x}$, $d\mathbf{y}$, $d\mathbf{w}$, $d\mathbf{b}$ always refer to the gradient of the final network error function with respect to a given parameter. So $d\mathbf{y}$ 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, $d\mathbf{Means}$ or $d\mathbf{BnBias}$ refer to gradients of the loss function wrt those parameters.



\mathbf{w} is used in the API for both the width of the \mathbf{x} tensor and convolution filter matrix. To resolve this ambiguity we use \mathbf{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 \mathbf{n} , and the second dimension defines the number of features maps \mathbf{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 decreasing 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 × 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 $\text{stride}[i-1] < \text{stride}[i] \times \text{dim}[i]$ for some of the *i* from `[1,nbDims]` interval.

2.5. 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.

2.6. 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:

- ▶ `cudaDnnConvolutionBackwardFilter` when `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` or `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3` is used
- ▶ `cudaDnnConvolutionBackwardData` when `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0` is used

- ▶ `cudaDnnPoolingBackward` when `CUDNN_POOLING_MAX` is used
- ▶ `cudaDnnSpatialTfSamplerBackward`

2.7. Scaling parameters `alpha` and `beta`

Many cuDNN routines like `cudaDnnConvolutionForward` take pointers to scaling factors (in host memory), that are used to blend computed values with initial values in the destination tensor as follows: $dstValue = alpha[0]*computedValue + beta[0]*priorDstValue$. When `beta[0]` is zero, the output is not read and may contain any uninitialized data (including NaN). The storage data type for `alpha[0]`, `beta[0]` is float for HALF and FLOAT tensors, and double for DOUBLE tensors. These parameters are passed using a host memory pointer.



For improved performance it is advised to use `beta[0] = 0.0`. Use a non-zero value for `beta[0]` only when blending with prior values stored in the output tensor is needed.

2.8. Tensor Core Operations

cuDNN v7 introduced the acceleration of compute intensive routines using Tensor Core hardware on supported GPU SM versions. Tensor Core acceleration (using Tensor Core Operations) can be exploited by the library user via the `cudaDnnMathType_t` enumerator. This enumerator specifies the available options for Tensor Core enablement and is expected to be applied on a per-routine basis.

Kernels using Tensor Core Operations for are available for both **Convolutions** and **RNNs**.

Tensor Core Operations for Convolution Functions

The below Convolution functions can be run as Tensor Core operations:

- ▶ `cudaDnnConvolutionForward`
- ▶ `cudaDnnConvolutionBackwardData`
- ▶ `cudaDnnConvolutionBackwardFilter`

Tensor Core Operations kernels will be triggered in these paths only when:

- ▶ `cudaDnnSetConvolutionMathType` is called on the appropriate convolution descriptor setting `mathType` to `CUDNN_TENSOR_OP_MATH`.
- ▶ `cudaDnnConvolutionForward` is called using `algo = CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` or `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED`
- ▶ `cudaDnnConvolutionBackwardData` using `algo = CUDNN_CONVOLUTION_BWD_DATA_ALGO_1` or `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED`, and

- ▶ `cudaConvolutionBackwardFilter` using `algo = CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` or `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAAD_NONFUSED`

For algorithms other than `*_ALGO_WINOGRAAD_NONFUSED`, the following are a few requirements to run Tensor Core operations:

- ▶ Input, Filter and Output descriptors (`xDesc`, `yDesc`, `wDesc`, `dxDesc`, `dyDesc` and `dwDesc` as applicable) have `dataType = CUDNN_DATA_HALF`.
- ▶ The number of Input and Output feature maps is a multiple of 8.
- ▶ The Filter is of type `CUDNN_TENSOR_NCHW` or `CUDNN_TENSOR_NHWC`. When using a filter of type `CUDNN_TENSOR_NHWC`, Input, Filter and Output data pointers (X, Y, W, dX, dY, and dW as applicable) need to be aligned to 128 bit boundaries.

Tensor Core Operations for RNN Functions

The RNN functions are:

- ▶ `cudaRNNForwardInference`
- ▶ `cudaRNNForwardTraining`
- ▶ `cudaRNNBackwardData`
- ▶ `cudaRNNBackwardWeights`
- ▶ `cudaRNNForwardInferenceEx`
- ▶ `cudaRNNForwardTrainingEx`
- ▶ `cudaRNNBackwardDataEx`
- ▶ `cudaRNNBackwardWeightsEx`

Tensor Core Operations kernels will be triggered in these paths only when:

- ▶ `cudaSetRNNMatrixMathType` is called on the appropriate RNN descriptor setting `mathType` to `CUDNN_TENSOR_OP_MATH`.
- ▶ All routines are called using `algo = CUDNN_RNN_ALGO_STANDARD` or `CUDNN_RNN_ALGO_PERSIST_STATIC`. (new for 7.1)
- ▶ For `algo = CUDNN_RNN_ALGO_STANDARD`, Hidden State size, Input size and Batch size are all multiples of 8. (new for 7.1)
- ▶ For `algo = CUDNN_RNN_ALGO_PERSIST_STATIC`, Hidden State size and Input size are multiples of 32, Batch size is a multiple of 8. If Batch size exceeds 96 (forward training or inference) or 32 (backward data), Batch sizes constraints may be stricter and large power-of-two Batch sizes may be needed. (new for 7.1)

See also [Features of RNN Functions](#).



For all cases, the `CUDNN_TENSOR_OP_MATH` enumerator is an indicator that the use of Tensor Cores is permissible, but not required. cuDNN may prefer not to use

Tensor Core Operations (for instance, when the problem size is not suited to Tensor Core acceleration), and instead use an alternative implementation based on regular floating point operations.

2.8.1. Tensor Core Operations Notes

Some notes on Tensor Core Operations use in cuDNN v7 on sm_70:

Tensor Core operations are supported on the Volta GPU family, those operations perform parallel floating point accumulation of multiple floating point products. Setting the math mode to `CUDNN_TENSOR_OP_MATH` indicates that the library will use Tensor Core operations as mentioned previously. The default is `CUDNN_DEFAULT_MATH`, this default indicates that the Tensor Core operations will be avoided by the library. The default mode is a serialized operation, the Tensor Core operations are parallelized operation, thus the two might result in slight different numerical results due to the different sequencing of operations. Note: The library falls back to the default math mode when Tensor Core operations are not supported or not permitted.

The result of multiplying two matrices using Tensor Core Operations is very close, but not always identical, to the product achieved using some sequence of legacy scalar floating point operations. So cuDNN requires explicit user opt-in before enabling the use of Tensor Core Operations. However, experiments training common Deep Learning models show negligible difference between using Tensor Core Operations and legacy floating point paths as measured by both final network accuracy and iteration count to convergence. Consequently, the library treats both modes of operation as functionally indistinguishable, and allows for the legacy paths to serve as legitimate fallbacks for cases in which the use of Tensor Core Operations is unsuitable.

2.8.2. Tensor Operations Speedup Tips

Some tips on Reducing Computation Time for Tensor Core Operations:

- ▶ The computation time for FP32 tensors can be reduced by selecting `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum value 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.
- ▶ When the input channel size `c` is a multiple of 32, you can use the new data type `CUDNN_DATA_INT8x32` to accelerate your convolution computation. If you are already using INT8, which is INT8x4, then to use the new INT8x32, ensure that your data is such that the input channel size `c` is a multiple of 32, instead of a multiple

of 4, as you would have had it for INT8x4. The new CUDNN_DATA_INT8x32 data type defines the data as 32-element vectors, each element being 8-bit signed integer.



This data type is only supported with the tensor format CUDNN_TENSOR_NCHW_VECT_C. See the description for [cudnnDataType_t](#).



This new data type can only be used with CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM. See [cudnnConvolutionFwdAlgo_t](#).



Note that this CUDNN_DATA_INT8x32 is only supported by sm_72.

2.9. 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.

2.10. 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_v<n>". "foo_v<n>" is retained as-is.
- ▶ The deprecated former API entry with an anterior suffix _v<f> and new API entry with suffix _v<n> are maintained in this release.
- ▶ In release **n+2**, both suffixed entries of a given entry are removed.

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

suffixes, the unsuffixed API entry is mapped to the higher numbered suffix. In that case it is strongly advised to use the non-suffixed API entry to guarantee backward compatibility with the following cuDNN release.

2.11. Grouped Convolutions

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



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 $2\times$: $(12 \times 4) / (2 \times (2 \times 6))$

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()`, `cudaDnnConvolutionBackwardData()` and `cudaDnnConvolutionBackwardFilter()`.
- ▶ 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]

2.12. API Logging

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

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

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

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

There are two methods to enable API logging.

Method 1: Using Environment Variables

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

- ▶ Set the environment variable `CUDNN_LOGINFO_DBG` to "1", and
- ▶ Set the environment variable `CUDNN_LOGDEST_DBG` to one of the following:
 - ▶ `stdout`, `stderr`, or a user-desired file path, for example, `/home/userName1/log.txt`.
- ▶ Include the conversion specifiers in the file name. For example:
 - ▶ To include date and time in the file name, use the date and time conversion specifiers: `log_%Y_%m_%d_%H_%M_%S.txt`. The conversion specifiers will be automatically replaced with the date and time when the program is initiated, resulting in `log_2017_11_21_09_41_00.txt`.
 - ▶ To include the process id in the file name, use the `%i` conversion specifier: `log_%Y_%m_%d_%H_%M_%S_%i.txt` for the result: `log_2017_11_21_09_41_00_21264.txt` when the process id is `21264`. When you have several processes running, using the process id conversion specifier will prevent these processes writing to the same file at the same time.



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

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



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

See also [Table 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 - No performance loss	- No logging output - No performance loss
CUDNN_LOGDEST_DBG=NULL	- No logging output - No performance loss	- No logging output - No performance loss
CUDNN_LOGDEST_DBG=stdout or stderr	- No logging output - No performance loss	- Logging to <code>stdout</code> or <code>stderr</code> - Some performance loss

Environment variables	CUDNN_LOGINFO_DBG=0	CUDNN_LOGINFO_DBG=1
CUDNN_LOGDEST_DBG= filename.txt	- No logging output - No performance loss	- Logging to filename.txt - Some performance loss

Method 2

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

2.13. Features of RNN Functions

The RNN functions are:

- ▶ `cudaRNNForwardInference`
- ▶ `cudaRNNForwardTraining`
- ▶ `cudaRNNBackwardData`
- ▶ `cudaRNNBackwardWeights`
- ▶ `cudaRNNForwardInferenceEx`
- ▶ `cudaRNNForwardTrainingEx`
- ▶ `cudaRNNBackwardDataEx`
- ▶ `cudaRNNBackwardWeightsEx`

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

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

Functions	Input output layout supported	Supports variable sequence length in batch	Commonly supported
<code>cudaRNNForwardInference</code>	Only Sequence major, packed (non-padded)	Only with <code>_ALGO_STANDARD</code> Require input sequences descending sorted according to length	Mode (cell type) supported: <code>CUDNN_RNN_RELU</code> , <code>CUDNN_RNN_TANH</code> , <code>CUDNN_LSTM</code> , <code>CUDNN_GRU</code> Algo supported* (see the table below for an elaboration on these algorithms):
<code>cudaRNNForwardTraining</code>			
<code>cudaRNNBackwardData</code>			
<code>cudaRNNBackwardWeights</code>			
<code>cudaRNNForwardInferenceEx</code>	Sequence major unpacked, Batch major unpacked**,	Only with <code>_ALGO_STANDARD</code> For unpacked layout**, no input sorting required.	<code>_ALGO_STANDARD</code> , <code>_ALGO_PERSIST_STATIC</code> , <code>_ALGO_PERSIST_DYNAMIC</code> Math mode supported:
<code>cudaRNNForwardTrainingEx</code>			
<code>cudaRNNBackwardDataEx</code>			

<code>cudaRNNBackwardWeightsEx</code>	Sequence major packed**	For packed layout, require input sequences descending sorted according to length	<code>CUDNN_DEFAULT_MATH</code> , <code>CUDNN_TENSOR_OP_MATH</code> (will automatically fall back if run on pre-Volta, or if algo doesn't support HMMA acceleration) <code>_ALLOW_CONVERSION</code> (may do down conversion to utilize HMMA acceleration) Direction mode supported: <code>CUDNN_UNIDIRECTIONAL</code> , <code>CUDNN_BIDIRECTIONAL</code> RNN input mode: <code>CUDNN_LINEAR_INPUT</code> , <code>CUDNN_SKIP_INPUT</code>
---------------------------------------	-------------------------	--	---

* 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 `cudaSetRNNPaddingMode`.

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

Features	<code>_ALGO_STANDARD</code>	<code>_ALGO_PERSIST_STATIC</code>	<code>_ALGO_PERSIST_DYNAMIC</code>
Half input Single accumulation Half output	Supported Half intermediate storage Single accumulation		
Single input Single accumulation Single output	Supported If running on Volta, with <code>CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION</code> ¹ , will down-convert and use half intermediate storage. Otherwise: Single intermediate storage Single accumulation		
Double input Double accumulation Double output	Supported Double intermediate storage Double accumulation	Not Supported	Supported Double intermediate storage Double accumulation
LSTM recurrent projection	Supported	Not Supported	Not Supported
LSTM cell clipping	Supported		

Variable sequence length in batch	Supported	Not Supported	Not Supported
HMMA acceleration on Volta/Xavier	Supported For half input/output, acceleration requires setting <code>CUDNN_TENSOR_OP_MATH</code> ¹ or <code>CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION</code> ¹ Acceleration requires <code>inputSize</code> and <code>hiddenSize</code> to be multiple of 8 For single input/output, acceleration requires setting <code>CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION</code> ¹ Acceleration requires <code>inputSize</code> and <code>hiddenSize</code> to be multiple of 8		Not Supported, will execute normally ignoring <code>CUDNN_TENSOR_OP_MATH</code> ¹ or <code>_ALLOW_CONVERSION</code> ¹
Other limitations		Max problem size is limited by GPU specifications.	Requires real time compilation through NVRTC

¹`CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` can be set through `cudaSetRNNMatrixMathType`.

2.14. Mixed Precision Numerical Accuracy

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

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

Chapter 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 a 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 (new for 7.1)

Selects the identity function, intended for bypassing the activation step in `cudaDnnConvolutionBiasActivationForward()`. (The `cudaDnnConvolutionBiasActivationForward()` function must use `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`.) Does not work with `cudaDnnActivationForward()` or `cudaDnnActivationBackward()`.

3.3. `cudaDnnBatchNormMode_t`

`cudaDnnBatchNormMode_t` is an enumerated type used to specify the mode of operation in `cudaDnnBatchNormalizationForwardInference()`, `cudaDnnBatchNormalizationForwardTraining()`, `cudaDnnBatchNormalizationBackward()` and `cudaDnnDeriveBNTensorDescriptor()` routines.

Values

CUDNN_BATCHNORM_PER_ACTIVATION

Normalization is performed per-activation. This mode is intended to be used after non-convolutional network layers. In this mode the tensor dimensions of `bnBias` and `bnScale`, the parameters used in the `cudaDnnBatchNormalization*` 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`, `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: `cudaDnnBatchNormalizationForwardTraining()`, and `cudaDnnBatchNormalizationBackward()`. In the case of `cudaDnnBatchNormalizationBackward()`, the `savedMean` and `savedInvVariance` arguments should not be NULL.

The rest of this section applies for 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 `cudaQueryRuntimeError()` to check if a numerical overflow occurred in this mode.

3.4. 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.

3.5. 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

3.6. 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, `cudnnDestroyCTCLossDescriptor()` is used to destroy this instance.

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 run 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 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. 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. Significant workspace may be needed to store intermediate results. The results are deterministic.

3.9. 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 how much workspace is needed to execute it.

CUDNN_CONVOLUTION_BWD_DATA_SPECIFY_WORKSPACE_LIMIT

In this configuration, the routine

`cudaGetConvolutionBackwardDataAlgorithm()` will return the fastest algorithm that fits within the memory limit that the user provided.

3.10. `cudaConvolutionBwdFilterAlgoPerf_t`

`cudaConvolutionBwdFilterAlgoPerf_t` is a structure containing performance results returned by `cudaFindConvolutionBackwardFilterAlgorithm()` or heuristic results returned by `cudaGetConvolutionBackwardFilterAlgorithm_v7()`.

Data Members

`cudaConvolutionBwdFilterAlgo_t algo`

The algorithm run to obtain the associated performance metrics.

`cudaStatus_t status`

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

- ▶ `CUDNN_STATUS_ALLOC_FAILED` if any error occurred during workspace allocation or if 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 `cudaConvolutionBackwardFilter()`.

`float time`

The execution time of `cudaConvolutionBackwardFilter()` (in milliseconds).

`size_t memory`

The workspace size (in bytes).

`cudaDeterminism_t determinism`

The determinism of the algorithm.

`cudaMathType_t mathType`

The math type provided to the algorithm.

`int reserved[3]`

Reserved space for future properties.

3.11. `cudaConvolutionBwdFilterAlgo_t`

cudaConvolutionBwdFilterAlgo_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. 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. 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. Significant workspace may be needed to store intermediate results. The results are deterministic.

3.12. cudaConvolutionBwdFilterPreference_t

cudaConvolutionBwdFilterPreference_t is an enumerated type used by **cudaGetConvolutionBackwardFilterAlgorithm()** 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 **cudaGetConvolutionBackwardFilterAlgorithm()** 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

`cudaGetConvolutionBackwardFilterAlgorithm()` will return the fastest algorithm regardless how much workspace is needed to execute it.

CUDNN_CONVOLUTION_BWD_FILTER_SPECIFY_WORKSPACE_LIMIT

In this configuration, the routine

`cudaGetConvolutionBackwardFilterAlgorithm()` will return the fastest algorithm that fits within the memory limit that the user provided.

3.13. `cudaConvolutionDescriptor_t`

`cudaConvolutionDescriptor_t` is a pointer to an opaque structure holding the description of a convolution operation. `cudaCreateConvolutionDescriptor()` is used to create one instance, and `cudaSetConvolutionNdDescriptor()` or `cudaSetConvolution2dDescriptor()` must be used to initialize this instance.

3.14. `cudaConvolutionFwdAlgoPerf_t`

`cudaConvolutionFwdAlgoPerf_t` is a structure containing performance results returned by `cudaFindConvolutionForwardAlgorithm()` or heuristic results returned by `cudaGetConvolutionForwardAlgorithm_v7()`.

Data Members

`cudaConvolutionFwdAlgo_t algo`

The algorithm run to obtain the associated performance metrics.

`cudaStatus_t status`

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

- ▶ `CUDNN_STATUS_ALLOC_FAILED` if any error occurred during workspace allocation or if 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 `cudaConvolutionForward()`.

`float time`

The execution time of `cudaConvolutionForward()` (in milliseconds).

`size_t memory`

The workspace size (in bytes).

`cudaDeterminism_t determinism`

The determinism of the algorithm.

cudaMathType_t mathType

The math type provided to the algorithm.

int reserved[3]

Reserved space for future properties.

3.15. 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 the 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 (e.g 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. Significant workspace may be needed to store intermediate results.

3.16. 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 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.

3.17. 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.

3.18. 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 32-bit single-precision floating point (**float**).

CUDNN_DATA_DOUBLE

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

CUDNN_DATA_HALF

The data is 16-bit floating point.

CUDNN_DATA_INT8

The data is 8-bit signed integer.

CUDNN_DATA_UINT8 (new for 7.1)

The data is 8-bit unsigned integer.

CUDNN_DATA_INT32

The data is 32-bit signed integer.

CUDNN_DATA_INT8x4

The data is 32-bit elements each composed of 4 8-bit signed integer. 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 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,” i.e., CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM. See [cudnnConvolutionFwdAlgo_t](#).

CUDNN_DATA_UINT8x4 (new for 7.1)

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

3.19. cudnnDeterminism_t

cudnnDeterminism_t is an enumerated type used to indicate if the computed results are deterministic (reproducible). See section 2.5 (Reproducibility) for more details on determinism.

Values

CUDNN_NON_DETERMINISTIC

Results are not guaranteed to be reproducible

CUDNN_DETERMINISTIC

Results are guaranteed to be reproducible

3.20. cudnnDirectionMode_t

cudaDirectionMode_t is an enumerated type used to specify the recurrence pattern in the `cudaRNNForwardInference()`, `cudaRNNForwardTraining()`, `cudaRNNBackwardData()` and `cudaRNNBackwardWeights()` routines.

Values

CUDNN_UNIDIRECTIONAL

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

CUDNN_BIDIRECTIONAL

Each layer of the 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.

3.21. cudaDivNormMode_t

cudaDivNormMode_t is an enumerated type used to specify the mode of operation in `cudaDivisiveNormalizationForward()` and `cudaDivisiveNormalizationBackward()`.

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. Note that 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 `cudaDivisiveNormalizationBackward()`.

3.22. cudaDropoutDescriptor_t

cudaDropoutDescriptor_t is a pointer to an opaque structure holding the description of a dropout operation. `cudaCreateDropoutDescriptor()` is used to create one instance, `cudaSetDropoutDescriptor()` is used to initialize this instance, `cudaDestroyDropoutDescriptor()` is used to destroy this instance, `cudaGetDropoutDescriptor()` is used to query fields of a previously initialized instance, `cudaRestoreDropoutDescriptor()` is used to restore an instance to a previously saved off state.

3.23. cudaErrQueryMode_t

cudaErrQueryMode_t is an enumerated type passed to `cudaQueryRuntimeError()` 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.

3.24. 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.

3.25. 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.

3.26. 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 long indices

CUDNN_16BIT_INDICES

Compute unsigned short indices

CUDNN_8BIT_INDICES

Compute unsigned char indices

3.27. 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].

3.28. 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.

3.29. 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

3.30. cudnnOpTensorDescriptor_t

cudnnOpTensorDescriptor_t is a pointer to an opaque structure holding the description of a Tensor Ccore Operation, used as a parameter to **cudnnOpTensor()**.

`cudaCreateOpTensorDescriptor()` is used to create one instance, and `cudaSetOpTensorDescriptor()` must be used to initialize this instance.

3.31. `cudaOpTensorOp_t`

`cudaOpTensorOp_t` is an enumerated type used to indicate the Tensor Core Operation to be used by the `cudaOpTensor()` routine. This enumerated type is used as a field for the `cudaOpTensorDescriptor_t` descriptor.

Values

`CUDA_OP_TENSOR_ADD`

The operation to be performed is addition

`CUDA_OP_TENSOR_MUL`

The operation to be performed is multiplication

`CUDA_OP_TENSOR_MIN`

The operation to be performed is a minimum comparison

`CUDA_OP_TENSOR_MAX`

The operation to be performed is a maximum comparison

`CUDA_OP_TENSOR_SQRT`

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

`CUDA_OP_TENSOR_NOT`

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

3.32. `cudaPersistentRNNPlan_t`

`cudaPersistentRNNPlan_t` is a pointer to an opaque structure holding a plan to execute a dynamic persistent RNN. `cudaCreatePersistentRNNPlan()` is used to create and initialize one instance.

3.33. `cudaPoolingDescriptor_t`

`cudaPoolingDescriptor_t` is a pointer to an opaque structure holding the description of a pooling operation. `cudaCreatePoolingDescriptor()` is used to create one instance, and `cudaSetPoolingNdDescriptor()` or `cudaSetPooling2dDescriptor()` must be used to initialize this instance.

3.34. `cudaPoolingMode_t`

`cudaPoolingMode_t` is an enumerated type passed to `cudaSetPoolingDescriptor()` to select the pooling method to be used by `cudaPoolingForward()` and `cudaPoolingBackward()`.

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.

3.35. `cudaRNNAngo_t`

`cudaRNNAngo_t` is an enumerated type used to specify the algorithm used in the `cudaRNNAngoForwardInference()`, `cudaRNNAngoForwardTraining()`, `cudaRNNAngoBackwardData()` and `cudaRNNAngoBackwardWeights()` 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 (ie. 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 optimized 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.

3.36. `cudaRNNClipMode_t`

`cudaRNNClipMode_t` is an enumerated type used to select the LSTM cell clipping mode. It is used with `cudaRNNSetClip()`, `cudaRNNGetClip()` functions, and internally within LSTM cells.

Values

`CUDNN_RNN_CLIP_NONE`

Disables LSTM cell clipping.

`CUDNN_RNN_CLIP_MINMAX`

Enables LSTM cell clipping.

3.37. `cudaRNNDescriptor_t`

`cudaRNNDescriptor_t` is a pointer to an opaque structure holding the description of an RNN operation. `cudaCreateRNNDescriptor()` is used to create one instance, and `cudaSetRNNDescriptor()` must be used to initialize this instance.

3.38. `cudaRNNDataDescriptor_t`

`cudaRNNDataDescriptor_t` is a pointer to an opaque structure holding the description of a RNN data set. The function `cudaCreateRNNDataDescriptor()` is used to create one instance, and `cudaSetRNNDataDescriptor()` must be used to initialize this instance.

3.39. `cudaRNNInputMode_t`

`cudaRNNInputMode_t` is an enumerated type used to specify the behavior of the first layer in the `cudaRNNForwardInference()`, `cudaRNNForwardTraining()`, `cudaRNNBackwardData()` and `cudaRNNBackwardWeights()` 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.

3.40. 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 \mathbf{h}_t for a given iteration can be computed from the recurrent input \mathbf{h}_{t-1} and the previous layer input \mathbf{x}_t given matrices \mathbf{W} , \mathbf{R} and biases \mathbf{b}_W , \mathbf{b}_R from the following equation:

$$\mathbf{h}_t = \text{ReLU}(\mathbf{W}_i \mathbf{x}_t + \mathbf{R}_i \mathbf{h}_{t-1} + \mathbf{b}_{W_i} + \mathbf{b}_{R_i})$$

Where $\text{ReLU}(\mathbf{x}) = \max(\mathbf{x}, 0)$.

CUDNN_RNN_TANH

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

In the forward pass the output \mathbf{h}_t for a given iteration can be computed from the recurrent input \mathbf{h}_{t-1} and the previous layer input \mathbf{x}_t given matrices \mathbf{W} , \mathbf{R} and biases \mathbf{b}_W , \mathbf{b}_R from the following equation:

$$\mathbf{h}_t = \tanh(\mathbf{W}_i \mathbf{x}_t + \mathbf{R}_i \mathbf{h}_{t-1} + \mathbf{b}_{W_i} + \mathbf{b}_{R_i})$$

Where \tanh is the hyperbolic tangent function.

CUDNN_LSTM

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

In the forward pass the output \mathbf{h}_t and cell output \mathbf{c}_t for a given iteration can be computed from the recurrent input \mathbf{h}_{t-1} , the cell input \mathbf{c}_{t-1} and the previous layer input \mathbf{x}_t given matrices \mathbf{W} , \mathbf{R} and biases \mathbf{b}_W , \mathbf{b}_R from the following equations:

$$\begin{aligned} i_t &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{R}_i \mathbf{h}_{t-1} + \mathbf{b}_{W_i} + \mathbf{b}_{R_i}) \\ f_t &= \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{R}_f \mathbf{h}_{t-1} + \mathbf{b}_{W_f} + \mathbf{b}_{R_f}) \\ o_t &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{R}_o \mathbf{h}_{t-1} + \mathbf{b}_{W_o} + \mathbf{b}_{R_o}) \\ c'_t &= \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{R}_c \mathbf{h}_{t-1} + \mathbf{b}_{W_c} + \mathbf{b}_{R_c}) \\ c_t &= f_t \circ c_{t-1} + i_t \circ c'_t \\ h_t &= o_t \circ \tanh(c_t) \end{aligned}$$

Where σ is the sigmoid operator: $\sigma(\mathbf{x}) = 1 / (1 + e^{-\mathbf{x}})$, \circ represents a point-wise multiplication and \tanh is the hyperbolic tangent function. \mathbf{i}_t , \mathbf{f}_t , \mathbf{o}_t , \mathbf{c}'_t represent the input, forget, output and new gates respectively.

CUDNN_GRU

A three-gate network consisting of Gated Recurrent Units.

In the forward pass the output \mathbf{h}_t for a given iteration can be computed from the recurrent input \mathbf{h}_{t-1} and the previous layer input \mathbf{x}_t given matrices \mathbf{W} , \mathbf{R} and biases \mathbf{b}_W , \mathbf{b}_R from the following equations:

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

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

3.41. 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.

3.42. 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.

3.43. 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.

3.44. 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

3.45. 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.

3.46. 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**

3.47. 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.

3.48. 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, **cudnnDestroySpatialTransformerDescriptor()** is used to destroy this instance.

3.49. cudnnStatus_t

cudaStatus_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 **cudaCreate()** fails or when **cudaCreate()** 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 **cudaCreate()** 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.

3.50. `cudaTensorDescriptor_t`

`cudaCreateTensorDescriptor_t` is a pointer to an opaque structure holding the description of a generic n-D dataset. `cudaCreateTensorDescriptor()` is used to create one instance, and one of the routines `cudaSetTensorNdDescriptor()`, `cudaSetTensor4dDescriptor()` or `cudaSetTensor4dDescriptorEx()` must be used to initialize this instance.

3.51. `cudaTensorFormat_t`

`cudaTensorFormat_t` is an enumerated type used by `cudaSetTensor4dDescriptor()` to create a tensor with a pre-defined layout.

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.

Chapter 4.

CUDNN API REFERENCE

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

4.1. cudnnActivationBackward

```
cudaStatus_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; i.e. `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 in-place operation to be allowed).



All tensor formats are supported for 4 and 5 dimensions, however 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. 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: $\text{dstValue} = \text{alpha}[0] * \text{result} + \text{beta}[0] * \text{priorDstValue}$. Refer to this section for additional details.

yDesc

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

The possible error values returned by this function and their meanings are listed below.

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 tensors 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 tensors differ.
- ▶ The **datatype** of the input tensor and output tensors 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; i.e., **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 in-place operation to be allowed).



All tensor formats are supported for 4 and 5 dimensions, however 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. 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: $dstValue = alpha[0]*result + beta[0]*priorDstValue$. [Please refer to this section for additional details.](#)

xDesc

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

The possible error values returned by this function and their meanings are listed below.

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 tensors differ.
- ▶ The **datatype** of the input tensor and output tensors differs.
- ▶ The strides **nStride, cStride, hStride, wStride** of the input tensor and output tensors differ and in-place operation is used (i.e., **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. See [cudnnHandle_t](#).

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: $\text{dstValue} = \text{alpha}[0] * \text{srcValue} + \text{beta}[0] * \text{priorDstValue}$. Refer to this section for additional details.

aDesc

Input. Handle to a previously initialized tensor descriptor. 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.

The possible error values returned by this function and their meanings are listed below.

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 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.



See `cudaDeriveBNTensorDescriptor` for the secondary tensor descriptor generation for the parameters using in this function.



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`.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor. See `cudaHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). See `cudaBatchNormMode_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:

$$\text{dstValue} = \text{alphaDataDiff}[0] * \text{resultValue} + \text{betaDataDiff}[0] * \text{priorDstValue}.$$

Refer to this section for additional details.

***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:

$$\text{dstValue} = \text{alphaParamDiff}[0] * \text{resultValue} + \text{betaParamDiff}[0] * \text{priorDstValue}.$$

Refer to this section for additional details.

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. 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 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. The value should be more than the value that is defined for `CUDNN_BN_MIN_EPSILON` in the header file `cudnn.h`. 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 has 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.
- ▶ 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 below 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 `cudaDnnGetBatchNormalizationBackwardExWorkspaceSize()`.
- ▶ **reserveSpaceSizeInBytes** is equal or larger than the amount required by `cudaDnnGetBatchNormalizationTrainingExReserveSpaceSize()`.
- ▶ The content in **reserveSpace** stored by `cudaDnnBatchNormalizationForwardTrainingEx()` must be preserved.

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

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. See `cudaDnnHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). See `cudaDnnBatchNormMode_t`.

bnOps

Input. Mode of operation for the fast NHWC kernel. See `cudaDnnBatchNormOps_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. Refer to this section for additional details.

***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. Refer to [this section](#) for additional details.

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, 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**. See [cudnnTensorDescriptor_t](#).

dBnScaleBiasDesc

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

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. Minimum allowed value is CUDNN_BN_MIN_EPSILON defined in cudnn.h. 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. 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 `cudaBatchNormalizationBackward()`.

workspaceSizeInBytes

Input. The size of the workspace. 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**. Must be equal or larger than the amount required by `cudaGetBatchNormalizationTrainingExReserveSpaceSize()`.

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.
- ▶ 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. cudaBatchNormalizationForwardInference

```

cudaStatus_t cudaBatchNormalizationForwardInference(
    cudaHandle_t          handle,
    cudaBatchNormMode_t  mode,
    const void            *alpha,
    const void            *beta,
    const cudaTensorDescriptor_t xDesc,
    const void            *x,
    const cudaTensorDescriptor_t yDesc,
    void                  *y,
    const cudaTensorDescriptor_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.



See `cudaDeriveBNTensorDescriptor` for the secondary tensor descriptor generation for the parameters using in this function.



Only 4D and 5D tensors are supported.



The input transformation performed by this function is defined as:

$$y = \text{beta} * y + \text{alpha} * [\text{bnBias} + (\text{bnScale} * (\text{x} - \text{estimatedMean}) / \sqrt{\text{epsilon} + \text{estimatedVariance}})]$$


The `epsilon` value has to be the same during training, backpropagation and inference.



For training phase use `cudaBatchNormalizationForwardTraining`.



Higher performance can be obtained when HW-packed tensors are used for all of `x` and `dx`.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor. See `cudaHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). See `cudaBatchNormMode_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:

$$\text{dstValue} = \text{alpha}[0] * \text{resultValue} + \text{beta}[0] * \text{priorDstValue}.$$

Refer to this [section for additional details](#).

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 descriptor 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 **cudaBatchNormalizationForwardTraining()** call, should be passed as inputs here.

epsilon

Input. Epsilon value used in the batch normalization formula. The value should be more than the value defined for CUDNN_BN_MIN_EPSILON in the cudnn.h header file.

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.
- ▶ 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.



See [cudnnDeriveBNTensorDescriptor](#) for the secondary tensor descriptor generation for the parameters using in this function.



Only 4D and 5D tensors are supported.



The `epsilon` value has to be the same during training, backpropagation and inference.



For inference phase use [cudnnBatchNormalizationForwardInference](#).



Higher performance can be obtained when HW-packed tensors are used for both `x` and `y`.

Parameters

handle

Handle to a previously created cuDNN library descriptor. See [cudnnHandle_t](#).

mode

Mode of operation (spatial or per-activation). 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. Refer to [this section for additional details](#).

xDesc, yDesc

Tensor descriptors and pointers in device memory for the layer's `x` and `y` data. 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:

$$\text{runningMean} = \text{runningMean} * (1 - \text{factor}) + \text{newMean} * \text{factor}$$

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

$$\text{CMA}[n] = (\mathbf{x}[1] + \dots + \mathbf{x}[n]) / n. \text{ This is proved below:}$$

$$\text{Writing } \text{CMA}[n+1] = (n * \text{CMA}[n] + \mathbf{x}[n+1]) / (n+1)$$

$$= ((n+1) * \text{CMA}[n] - \text{CMA}[n]) / (n+1) + \mathbf{x}[n+1] / (n+1)$$

$$= \text{CMA}[n] * (1 - 1 / (n+1)) + \mathbf{x}[n+1] * 1 / (n+1)$$

$$= \text{CMA}[n] * (1 - \text{factor}) + \mathbf{x}[n+1] * \text{factor}.$$

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 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. The value should be more than the value defined for CUDNN_BN_MIN_EPSILON in the cudnn.h header file. 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

`cudaBatchNormalizationBackward()` 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 `cudaBatchNormalizationBackward()` 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.
- ▶ 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 is NULL.
- ▶ Exactly one of `resultRunningMean`, `resultRunningInvVariance` pointers is NULL.
- ▶ `epsilon` value is less than CUDNN_BN_MIN_EPSILON.
- ▶ Dimensions or data types mismatch for `xDesc`, `yDesc`

4.8. cudaBatchNormalizationForwardTrainingEx

```

cudaStatus_t cudaBatchNormalizationForwardTrainingEx(
    cudaHandle_t          handle,
    cudaBatchNormMode_t  mode,
    cudaBatchNormOps_t   bnOps,
    const void           *alpha,
    const void           *beta,
    const cudaTensorDescriptor_t xDesc,
    const void           *xData,
    const cudaTensorDescriptor_t zDesc,
    const void           *zData,
    const cudaTensorDescriptor_t yDesc,
    void                 *yData,
    const cudaTensorDescriptor_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 below 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

Input. Handle to a previously created cuDNN library descriptor. See `cudnnHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). 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. Refer to this section for additional details.

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 exact 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. 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 the device memory for the for the batch normalization scale and bias data. In the [original paper](#) bias is referred to as beta and scale as gamma. Note that **bnBiasData** parameter can replace the previous operation'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 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. Minimum allowed value is CUDNN_BN_MIN_EPSILON defined in cudnn.h. Same **epsilon** value should be used in forward and backward functions.

***saveMean, *saveInvVariance**

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. 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 the ***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.

- ▶ 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 is NULL.
- ▶ Exactly one of **resultRunningMeanData**, **resultRunningInvVarianceData** pointers is NULL.
- ▶ **epsilon** value is less than CUDNN_BN_MIN_EPSILON.
- ▶ Dimensions or data types mismatch for **xDesc**, **yDesc**

4.9. 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.

Parameters

handle

Input. Handle to a previously created cuDNN context. See [cudnnHandle_t](#).

probsDesc

Input. Handle to the previously initialized probabilities tensor descriptor. See [cudnnTensorDescriptor_t](#).

probs

Input. Pointer to a previously initialized probabilities tensor.

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.

algo

Input. Enumerant that specifies the chosen CTC loss algorithm. See [cudnnCTCLossAlgo_t](#).

ctcLossDesc

Input. Handle to the previously initialized CTC loss descriptor. 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**.

The possible error values returned by this function and their meanings are listed below.

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

4.10. 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. 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: $\text{dstValue} = \alpha[0] * \text{result} + \beta[0] * \text{priorDstValue}$. Refer to this section for additional details.

dyDesc

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

The possible error values returned by this function and their meanings are listed below.

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 are different.

4.11. 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. 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: $\text{dstValue} = \text{alpha}[0] * \text{result} + \text{beta}[0] * \text{priorDstValue}$. [Refer to this section for additional details.](#)

wDesc

Input. Handle to a previously initialized filter descriptor. 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. See [cudnnTensorDescriptor_t](#).

dy

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

convDesc

Input. Previously initialized convolution descriptor. See [cudnnConvolutionDescriptor_t](#).

algo

Input. Enumerant that specifies which backward data convolution algorithm should be used to compute the results. 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.

TABLE OF THE SUPPORTED CONFIGURATIONS

This function supports the following combinations of data types for **wDesc**, **dyDesc**, **convDesc**, and **dxDesc**. See the following table for a list of the supported configurations.

Data Type Configurations	wDesc's, dyDesc's and dxDesc's Data Type	convDesc's Data Type
TRUE_HALF_CONFIG (only supported on architectures with true fp16 support, i.e., 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



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.

TABLE OF THE SUPPORTED ALGORITHMS

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_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**)

FOR 2D CONVOLUTIONS.

Filter descriptor <code>wDesc</code> : <code>_NHWC</code> . See cudnnTensorFormat_t .					
Algo Name (see below for 3D Convolutions)	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configurations Supported	Important
<code>_ALGO_1</code>		NHWC HWC- packed	NHWC HWC- packed	- <code>TRUE_HALF_CONFIG</code> , - <code>PSEUDO_HALF_CONFIG</code> , and - <code>FLOAT_CONFIG</code>	
Filter descriptor <code>wDesc</code> : <code>_NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configurations Supported	Important
<code>_ALGO_0</code>	No	NCHW CHW- packed	All except <code>_NCHW_VECT</code>	- <code>PSEUDO_HALF_CONFIG</code> , - <code>FLOAT_CONFIG</code> , and - <code>DOUBLE_CONFIG</code>	- Dilation : greater than 0 for all dimensions - <code>convDesc</code> Group Count Support: Greater than 0.
<code>_ALGO_1</code>	Yes	NCHW CHW- packed	All except <code>_NCHW_VECT</code>	- <code>TRUE_HALF_CONFIG</code> , - <code>PSEUDO_HALF_CONFIG</code> , - <code>FLOAT_CONFIG</code> , and - <code>DOUBLE_CONFIG</code>	- Dilation : 1 for all dimensions - <code>convDesc</code> Group Count Support: Greater than 0.
<code>_FFT</code>	Yes	NCHW CHW- packed	NCHW HW- packed	- <code>PSEUDO_HALF_CONFIG</code> and - <code>FLOAT_CONFIG</code>	- Dilation : 1 for all dimensions - <code>convDesc</code> Group Count Support: Greater than 0. - <code>dxDesc</code> 's feature map height + 2 * <code>convDesc</code> 's zero-padding height must equal 256 or less - <code>dxDesc</code> 's feature map width + 2 * <code>convDesc</code> 's zero-padding width must equal 256 or less

					<ul style="list-style-type: none"> - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter height must be greater than convDesc's zero-padding height - wDesc's filter width must be greater than convDesc's zero-padding width
_FFT_TILING	Yes	NCHW CHW-packed	NCHW HW-packed	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG - DOUBLE_CONFIG is also supported when the task can be handled by 1D FFT, ie, one of the filter dimension, width or height is 1. 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - When neither of wDesc's filter dimension is 1, the filter width and height must not be larger than 32 - When either of wDesc's filter dimension is 1, the largest filter dimension should not exceed 256 - convDesc's vertical and horizontal filter stride must equal 1 when either the filter width or filter height is 1, otherwise the stride can be 1 or 2 - wDesc's filter height must be greater than convDesc's zero-padding height - wDesc's filter width must be greater than convDesc's zero-padding width
_WINOGRAD	Yes	NCHW CHW-packed	All except _NCHW_VECT	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG and 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions

				- FLOAT_CONFIG	- convDesc Group Count Support: Greater than 0. - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter height must be 3 - wDesc's filter width must be 3
_WINOGRAD_NONFUSED		NCHW CHW-packed	All except _NCHW_VECT	- TRUE_HALF_CONFIG, - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG	- Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter (height, width) must be (3,3) or (5,5) - If wDesc's filter (height, width) is (5,5) then the data type config TRUE_HALF_CONFIG is not supported

FOR 3D CONVOLUTIONS.

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 Support	Important
_ALGO_0	Yes	NCDHW CDHW-packed	All except _NCDHW_VECT	- PSEUDO_HALF_CONFIG, - FLOAT_CONFIG, and - 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, and	- Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0.

				- DOUBLE_CONFIG.	
_FFT_TILING	Yes	NCDHW CDHW- packed	NCDHW DHW- packed	- PSEUDO_HALF_CONFIG, - FLOAT_CONFIG, and - DOUBLE_CONFIG.	- Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - wDesc 's filter height must equal 16 or less - wDesc 's filter width must equal 16 or less - wDesc 's filter depth must equal 16 or less - convDesc 's must have all filter strides equal to 1 - wDesc 's filter height must be greater than convDesc 's zero-padding height - wDesc 's filter width must be greater than convDesc 's zero-padding width - wDesc 's filter depth must be greater than convDesc 's 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).

- ▶ **dyDescs** 's spatial sizes do not match with the expected size as determined by `cudaGetConvolutionNdForwardOutputDim`

CUDNN_STATUS_NOT_SUPPORTED

At least one of the following conditions are met:

- ▶ **dyDesc** or **dxDesc** have 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 exhaustive list of parameter support for 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.

4.12. `cudaConvolutionBackwardFilter`

```

cudaStatus_t cudaConvolutionBackwardFilter(
    cudaHandle_t          handle,
    const void            *alpha,
    const cudaTensorDescriptor_t xDesc,
    const void            *x,
    const cudaTensorDescriptor_t dyDesc,
    const void            *dy,
    const cudaConvolutionDescriptor_t convDesc,
    cudaConvolutionBwdFilterAlgo_t algo,
    void                 *workSpace,
    size_t                workSpaceSizeInBytes,
    const void            *beta,
    const cudaFilterDescriptor_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 `cudaConvolutionForward()`. 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. See `cudaHandle_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: $dstValue = alpha[0]*result + beta[0]*priorDstValue$. Refer to this section for additional details.

xDesc

Input. Handle to a previously initialized tensor descriptor. 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. See [cudnnConvolutionDescriptor_t](#).

algo

Input. Enumerant that specifies which convolution algorithm should be used to compute the results. 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. See [cudnnFilterDescriptor_t](#).

dw

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

TABLE OF THE SUPPORTED CONFIGURATIONS

This function supports the following combinations of data types for **xDesc**, **dyDesc**, **convDesc**, and **dwDesc**. See the following table for a list of the supported configurations.

Data Type Configurations	xDesc 's, dyDesc 's and dwDesc 's Data Type	convDesc 's Data Type
TRUE_HALF_CONFIG (only supported on architectures with true fp16 support, i.e., compute capability 5.3 and later).	CUDNN_DATA_HALF	CUDNN_DATA_HALF
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT

Data Type Configurations	xDesc's, dyDesc's and dwDesc's Data Type	convDesc's Data Type
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE


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

TABLE OF THE SUPPORTED ALGORITHMS

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_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**)

FOR 2D CONVOLUTIONS.

Filter descriptor dwDesc: _NHWC . See cudnnTensorFormat_t .					
Algo Name (see below for 3D Convolutions)	Deterministic (Yes or No)	Tensor Formats Supported for xDesc	Tensor Formats Supported for dyDesc	Data Type Configurations Supported	Important
_ALGO_0 , and _ALGO_1		NHWC HWC- packed	NHWC HWC- packed	- PSEUDO_HALF_CONFIG, and - FLOAT_CONFIG	
Filter descriptor wDesc: _NCHW .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for xDesc	Tensor Formats Supported for dyDesc	Data Type Configurations Supported	Important
_ALGO_0	No	All except _NCHW_VECT_C	NCHW CHW- packed	- PSEUDO_HALF_CONFIG	- Dilation : greater than 1 for all dimensions

				<ul style="list-style-type: none"> - FLOAT_CONFIG, and - DOUBLE_CONFIG 	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - TRUE_HALF_CONFIG - PSEUDO_HALF_CONFIG - FLOAT_CONFIG, and - DOUBLE_CONFIG 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0.
_FFT	Yes	NCHW CHW-packed	NCHW CHW-packed	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - xDesc's feature map height + 2 * convDesc's zero-padding height must equal 256 or less - xDesc's feature map width + 2 * convDesc's zero-padding width must equal 256 or less - convDesc's vertical and horizontal filter stride must equal 1 - dwDesc's filter height must be greater than convDesc's zero-padding height - dwDesc's filter width must be greater than convDesc's zero-padding width
_ALGO_3	Yes	All except _NCHW_VECT_P	NCHW CHW-packed	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions

				<ul style="list-style-type: none"> - FLOAT_CONFIG, and - DOUBLE_CONFIG 	<ul style="list-style-type: none"> - convDesc Group Count Support: Greater than 0.
_WINOGRAD_NONFUSEDs		All except _NCHW_VECT_PACKED	NCHW CHW-packed	<ul style="list-style-type: none"> - TRUE_HALF_CONFIG - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter (height, width) must be (3,3) or (5,5) - If wDesc's filter (height, width) is (5,5), then the data type config TRUE_HALF_CONFIG is not supported.
_FFT_TILING	Yes	NCHW CHW-packed	NCHW CHW-packed	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG - FLOAT_CONFIG, and - DOUBLE_CONFIG 	<ul style="list-style-type: none"> - Dilation: 1 for all dimensions - convDesc Group Count Support: Greater than 0. - xDesc's width or height must equal 1 - dyDesc's width or height must equal 1 (the same dimension as in xDesc.) The other dimension must be less than or equal to 256, i.e., the largest 1D tile size currently supported. - convDesc's vertical and horizontal filter stride must equal 1 - dwDesc's filter height must be greater than convDesc's zero-padding height.

					- dwDesc 's filter width must be greater than convDesc 's zero-padding width.
--	--	--	--	--	---

FOR 3D CONVOLUTIONS.

Filter descriptor wDesc : _NCHW					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for xDesc	Tensor Formats Supported for dyDesc	Data Type Configurations Support	Important
_ALGO_0	No	All except _NCDHW_VECT	NCDHW -packed	- PSEUDO_HALF_CONFIG - FLOAT_CONFIG , and - 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 , and - DOUBLE_CONFIG	- Dilation : 1 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.

4.13. 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 $\mathbf{y} = \text{act} (\alpha_1 * \text{conv}(\mathbf{x}) + \alpha_2 * \mathbf{z} + \text{bias})$.



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`. See also the documentation for the function `cudnnSetActivationDescriptor()`.

Parameters

handle

Input. Handle to a previously created cuDNN context. 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 described by the above equation. [Please refer to this section for additional details.](#)

xDesc

Input. Handle to a previously initialized tensor descriptor. 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. See [cudnnFilterDescriptor_t](#).

w

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

convDesc

Input. Previously initialized convolution descriptor. See [cudnnConvolutionDescriptor_t](#).

algo

Input. Enumerant that specifies which convolution algorithm should be used to compute the results. 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. 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 `cudaDnnConvolutionForward()`. The following table specifies the supported combinations of data types for **x**, **y**, **z**, **bias**, and **alpha1/alpha2**.

Table Key: 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_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

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

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: **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. See the following for some examples of non-supported configurations:

- ▶ 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.

4.14. 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. 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: $\text{dstValue} = \text{alpha}[0] * \text{result} + \text{beta}[0] * \text{priorDstValue}$. Refer to this section for additional details.

xDesc

Input. Handle to a previously initialized tensor descriptor. 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. See [cudnnFilterDescriptor_t](#).

w

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

convDesc

Input. Previously initialized convolution descriptor. See [cudnnConvolutionDescriptor_t](#).

algo

Input. Enumerant that specifies which convolution algorithm should be used to compute the results. 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.

TABLE OF THE SUPPORTED CONFIGURATIONS

This function supports the following combinations of data types for **xDesc**, **wDesc**, **convDesc**, and **yDesc**. See the following table for a list of the supported configurations.

Data Type Configurations	xDesc and wDesc	convDesc	yDesc
TRUE_HALF_CONFIG (only supported on architectures with true fp16 support, i.e., 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
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
INT8_CONFIG (only supported on architectures with DP4A support, i.e.,	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_INT8

Data Type Configurations	xDesc and wDesc	convDesc	yDesc
compute capability 6.1 and later).			
INT8_EXT_CONFIG (only supported on architectures with DP4A support, i.e., compute capability 6.1 and later).	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
INT8x4_CONFIG (only supported on architectures with DP4A support, i.e., 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, i.e., compute capability 6.1 and later).	CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
UINT8x4_CONFIG (new for 7.1) (only supported on architectures with DP4A support, i.e., compute capability 6.1 and later).	CUDNN_DATA_UINT8x4	CUDNN_DATA_INT32	CUDNN_DATA_UINT8x4
UINT8x4_EXT_CONFIG (new for 7.1) (only supported on architectures with DP4A support, i.e., compute capability 6.1 and later).	CUDNN_DATA_UINT8x4	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT



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

TABLE OF THE SUPPORTED ALGORITHMS

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**)

FOR 2D CONVOLUTIONS.

Filter descriptor <code>wDesc</code> : <code>_NCHW</code> . See cudnnTensorFormat_t . <code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name (see below for 3D Convolutions)	Tensor Formats Supported for <code>xDesc</code>	Tensor Formats Supported for <code>yDesc</code>	Data Type Configurations Supported	Important
_IMPLICIT_GEMM	All except <code>_NCHW_VECT_C</code> .	All except <code>_NCHW_VECT_C</code> .	- PSEUDO_HALF_CONFIG, - FLOAT_CONFIG, and - DOUBLE_CONFIG	Dilation: Greater than 1 for all dimensions.
_IMPLICIT_PRECOMP_GEMM			- TRUE_HALF_CONFIG, - PSEUDO_HALF_CONFIG, - FLOAT_CONFIG, and - DOUBLE_CONFIG.	Dilation: 1 for all dimensions.
_GEMM			- PSEUDO_HALF_CONFIG, - FLOAT_CONFIG, and - DOUBLE_CONFIG	Dilation: 1 for all dimensions.
_FFT	NCHW HW-packed	NCHW HW-packed	- PSEUDO_HALF_CONFIG and - FLOAT_CONFIG	Dilation: 1 for all dimensions. - <code>xDesc</code> 's feature map height + 2 * <code>convDesc</code> 's zero-padding height must equal 256 or less - <code>xDesc</code> 's feature map width + 2 * <code>convDesc</code> 's zero-padding width must equal 256 or less


				<ul style="list-style-type: none"> - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter height must be greater than convDesc's zero-padding height - wDesc's filter width must be greater than convDesc's zero-padding width
_FFT_TILING			<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG <p>DOUBLE_CONFIG is also supported when the task can be handled by 1D FFT, i.e., one of the filter dimension, width or height is 1.</p>	<p>Dilation: 1 for all dimensions.</p> <ul style="list-style-type: none"> - When neither of wDesc's filter dimension is 1, the filter width and height must not be larger than 32 - When either of wDesc's filter dimension is 1, the largest filter dimension should not exceed 256 - convDesc's vertical and horizontal filter stride must equal 1 when either the filter width or filter height is 1, otherwise the stride can be 1 or 2 - wDesc's filter height must be greater than convDesc's zero-padding height - wDesc's filter width must be greater than convDesc's zero-padding width
_WINOGRAD	All except: _NCHW_VECT_C	All except: _NCHW_VECT_C	<ul style="list-style-type: none"> - PSEUDO_HALF_CONFIG and - FLOAT_CONFIG 	<p>Dilation: 1 for all dimensions.</p> <ul style="list-style-type: none"> - convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter height must be 3 - wDesc's filter width must be 3
_WINOGRAD_NONFUSED			<ul style="list-style-type: none"> - TRUE_HALF_CONFIG 	<p>Dilation: 1 for all dimensions.</p>

			- PSEUDO_HALF_CONFIG and - FLOAT_CONFIG	- convDesc's vertical and horizontal filter stride must equal 1 - wDesc's filter (height, width) must be (3,3) or (5,5) - If wDesc's filter (height, width) is (5,5), then data type config TRUE_HALF_CONFIG is not supported
_DIRECT	Currently not implemented in cuDNN.			
Filter descriptor wDesc: _NHWC convDesc Group count support: Greater than 0.				
Algo Name	xDesc	yDesc	Data Type Configurations Support	Important
_IMPLICIT_GEMM	NCHWC HWC-packed	NCHWC HWC-packed	- PSEUDO_HALF_CONFIG and - FLOAT_CONFIG	Dilation: Greater than 1 for all dimensions.
Filter descriptor wDesc: _NHWC convDesc Group count support: Greater than 0.				
Algo Name	xDesc	yDesc	Data Type Configurations Support	Important
_IMPLICIT_PRECOMP_GEMM	NCHWC	NHWC	- INT8_CONFIG, - INT8_EXT_CONFIG, - INT8x4_CONFIG, - INT8x4_EXT_CONFIG, - UINT8x4_CONFIG, and - UINT8x4_EXT_CONFIG	Dilation: 1 for all dimensions. Input and output features maps must be multiple of 4.

FOR 3D CONVOLUTIONS.

Filter descriptor wDesc: _NCHW convDesc Group count support: Greater than 0, for all algos.				
Algo Name	xDesc	yDesc	Data Type Configurations Support	Important

<code>_IMPLICIT_GEMM</code>	All except <code>_NCHW_VECT_C</code> .	All except <code>_NCHW_VECT_C</code> .	- <code>PSEUDO_HALF_CONFIG</code>	Dilation: Greater than 1 for all dimensions.
<code>_IMPLICIT_PRECOMP_GEMM</code>			- <code>FLOAT_CONFIG</code> , and - <code>DOUBLE_CONFIG</code> .	Dilation: 1 for all dimensions.
<code>_FFT_TILING</code>	NCDHW DHW-packed	NCDHW DHW-packed	- <code>PSEUDO_HALF_CONFIG</code> - <code>FLOAT_CONFIG</code> , and - <code>DOUBLE_CONFIG</code> .	Dilation: 1 for all dimensions. - <code>wDesc</code> 's filter height must equal 16 or less - <code>wDesc</code> 's filter width must equal 16 or less - <code>wDesc</code> 's filter depth must equal 16 or less - <code>convDesc</code> 's must have all filter strides equal to 1 - <code>wDesc</code> 's filter height must be greater than <code>convDesc</code> 's zero-padding height - <code>wDesc</code> 's filter width must be greater than <code>convDesc</code> 's zero-padding width - <code>wDesc</code> 's filter depth must be greater than <code>convDesc</code> 's zero-padding width

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

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`'s 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
- ▶ **yDescs**'s spatial sizes do not match with the expected size as determined by **cudaGetConvolutionNdForwardOutputDim**
- ▶ 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 occurred during the texture binding of the filter data.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.15. cudnnCreate

```
cudaStatus_t cudnnCreate(cudaHandle_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 (e.g., 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. See [cudaHandle_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.

4.16. cudnnCreateActivationDescriptor

```

cudnnStatus_t cudnnCreateActivationDescriptor(
    cudnnActivationDescriptor_t *activationDesc)

```

This function creates a activation descriptor object by allocating the memory needed to hold its opaque structure. See [cudnnActivationDescriptor_t](#).

Returns**CUDNN_STATUS_SUCCESS**

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.17. cudnnCreateAlgorithmDescriptor

```

cudnnStatus_t cudnnCreateAlgorithmDescriptor(
    cudnnAlgorithmDescriptor_t *algoDesc)

```

(New for 7.1)

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.

4.18. cudnnCreateAlgorithmPerformance

```

cudnnStatus_t cudnnCreateAlgorithmPerformance(
    cudnnAlgorithmPerformance_t *algoPerf,
    int numberToCreate)

```

(New for 7.1)

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.

4.19. cudnnCreateCTCLossDescriptor

```

cudnnStatus_t cudnnCreateCTCLossDescriptor(
    cudnnCTCLossDescriptor_t* ctcLossDesc)

```

This function creates a CTC loss function descriptor. .

Parameters

ctcLossDesc

Output. CTC loss descriptor to be set. 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.

4.20. cudnnCreateConvolutionDescriptor

```

cudnnStatus_t cudnnCreateConvolutionDescriptor(
    cudnnConvolutionDescriptor_t *convDesc)

```


This function creates a convolution descriptor object by allocating the memory needed to hold its opaque structure. See [cudnnConvolutionDescriptor_t](#).

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

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. 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. See [cudnnFilterDescriptor_t](#).

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.23. 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.

4.24. cudnnCreateOpTensorDescriptor

```

cudnnStatus_t cudnnCreateOpTensorDescriptor(
    cudnnOpTensorDescriptor_t* opTensorDesc)

```

This function creates a Tensor Pointwise math descriptor. 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.

4.25. 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. 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.

4.26. 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.

4.27. 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.

4.28. 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.

4.29. 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.

Parameters

None.

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.

4.30. 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.

4.31. 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 be all zero.

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.

4.32. 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, 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, and 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 `cudaCreateTensorDescriptor`.



`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.

4.33. `cudaDestroy`

```
cudaStatus_t cudaDestroy(cudaHandle_t handle)
```

This function releases resources used by the cuDNN handle. This function is usually the last call with a particular handle to the cuDNN handle. Because `cudaCreate` allocates some internal resources, the release of those resources by calling `cudaDestroy` will implicitly call `cudaDeviceSynchronize`; therefore, the recommended best practice is to call `cudaCreate/cudaDestroy` 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.

4.34. 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.

4.35. cudnnDestroyAlgorithmDescriptor

```

cudnnStatus_t cudnnDestroyAlgorithmDescriptor(
    cudnnActivationDescriptor_t algorithmDesc)

```

(New for 7.1)

This function destroys a previously created algorithm descriptor object.

Returns**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

4.36. cudnnDestroyAlgorithmPerformance

```

cudnnStatus_t cudnnDestroyAlgorithmPerformance(
    cudnnAlgorithmPerformance_t algoPerf)

```

(New for 7.1)

This function destroys a previously created algorithm descriptor object.

Returns**CUDNN_STATUS_SUCCESS**

The object was destroyed successfully.

4.37. 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.

4.38. cudnnDestroyConvolutionDescriptor

```

cudnnStatus_t cudnnDestroyConvolutionDescriptor(
    cudnnConvolutionDescriptor_t convDesc)

```

This function destroys a previously created convolution descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.39. 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.

4.40. cudnnDestroyFilterDescriptor

```

cudnnStatus_t cudnnDestroyFilterDescriptor(
    cudnnFilterDescriptor_t filterDesc)

```

This function destroys a previously created Tensor4D descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.41. 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.

4.42. 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.

4.43. 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.

4.44. 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.

4.45. 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.

4.46. 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.

4.47. 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.

4.48. 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.

4.49. cudnnDestroyTensorDescriptor

```
cudaStatus_t cudnnDestroyTensorDescriptor(cudaTensorDescriptor_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.

4.50. cudnnDivisiveNormalizationBackward

```
cudaStatus_t cudnnDivisiveNormalizationBackward(
    cudaHandle_t             handle,
    cudaLRNDescriptor_t     normDesc,
    cudaDivNormMode_t       mode,
    const void               *alpha,
    const cudaTensorDescriptor_t xDesc,
    const void               *x,
    const void               *means,
    const void               *dy,
    void                     *temp,
    void                     *temp2,
    const void               *beta,
    const cudaTensorDescriptor_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: $\text{dstValue} = \text{alpha}[0] * \text{resultValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

xDesc, x, means

Input. Tensor descriptor and pointers in device memory for the layer's x and means data. Note: 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 layer's resulting cumulative gradients dx and $dMeans$ ($dLoss/dx$ and $dLoss/dMeans$). Both share the same descriptor.

Possible error values returned by this function and their meanings are listed below.

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 , $temp$, $temp2$, 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. See the following for some examples of non-supported configurations:

- ▶ Any of the input and output tensor strides mismatch (for the same dimension).

4.51. 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 Divisive Normalization only implements the $x/\max(c, \sigma_x)$ portion of the computation, where σ_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:

$$x_m = x - \text{mean}(x);$$

$$y = x_m / \max(c, \sigma(x_m));$$

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: $\text{dstValue} = \text{alpha}[0] * \text{resultValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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.

Possible error values returned by this function and their meanings are listed below.

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. See the following for some examples of non-supported configurations:

- ▶ Any of the input and output tensor strides mismatch (for the same dimension).

4.52. 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 **cudaDnnDropoutForward** and has not been changed.

reserveSpaceSizeInBytes

Input. Specifies size in bytes of the provided memory for the reserve space

The possible error values returned by this function and their meanings are listed below.

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 than the value returned by **cudaDnnDropoutGetReserveSpaceSize**
- ▶ **cudaDnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-NULL **states** argument

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.53. cudaDnnDropoutForward

```

cudaDnnStatus_t cudaDnnDropoutForward(
    cudaDnnHandle_t          handle,
    const cudaDnnDropoutDescriptor_t dropoutDesc,
    const cudaDnnTensorDescriptor_t xdesc,
    const void              *x,
    const cudaDnnTensorDescriptor_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 **cudaDnnSetDropoutDescriptor**, the approximately **dropout** fraction of **x** values will be replaced by 0, and the rest will

be scaled by $1/(1-\text{dropout})$. This function should not be running concurrently with another `cudaDnnDropoutForward` function using the same `states`.



Better performance is obtained for fully packed tensors



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 contents of `reserveSpace` do not change between `cudaDnnDropoutForward` and `cudaDnnDropoutBackward` calls.

`reserveSpaceSizeInBytes`

Input. Specifies size in bytes of the provided memory for the reserve space.

The possible error values returned by this function and their meanings are listed below.

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 than the value returned by **cudaDnnDropoutGetReserveSpaceSize**.
- ▶ **cudaDnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-NULL **states** argument.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.54. cudaDnnDropoutGetReserveSpaceSize

```
cudaDnnStatus_t cudaDnnDropoutGetReserveSpaceSize(
    cudaDnnTensorDescriptor_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 **cudaDnnDropoutForward** and **cudaDnnDropoutBackward**, and its contents is expected to remain unchanged between **cudaDnnDropoutForward** and **cudaDnnDropoutBackward** 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**.

The possible error values returned by this function and their meanings are listed below.

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

4.55. cudaDnnDropoutGetStatesSize

```
cudaDnnStatus_t cudaDnnDropoutGetStatesSize(
    cudaDnnHandle_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 **cudaDnnDropoutForward** function.

Parameters

handle

Input. Handle to a previously created cuDNN context.

sizeInBytes

Output. Amount of GPU memory needed to store random generator states.

The possible error values returned by this function and their meanings are listed below.

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

4.56. 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 cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for `cudnnConvolutionBackwardData()`, using memory allocated via `cudaMalloc()` and outputs performance metrics to a user-allocated array of `cudnnConvolutionBwdDataAlgoPerf_t`. These metrics are written in sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudnnGetConvolutionBackwardMaxCount()`.



This function is host blocking.



It is recommend 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.57. 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 cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for `cudaConvolutionBackwardData`, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of `cudaConvolutionBwdDataAlgoPerf_t`. These metrics are written in sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudaGetConvolutionBackwardMaxCount()`.



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**

The possible error values returned by this function and their meanings are listed below.

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.

4.58. 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 cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for **cudnnConvolutionBackwardFilter()**, using GPU memory allocated via **cudaMalloc()**, and outputs performance metrics to a user-allocated array of **cudnnConvolutionBwdFilterAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest compute time.

The total number of resulting algorithms can be queried through the API `cudaDnnGetConvolutionBackwardMaxCount()`.



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.

The possible error values returned by this function and their meanings are listed below.

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.

4.59. 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 cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for **cudnnConvolutionBackwardFilter**, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of **cudnnConvolutionBwdFilterAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API **cudnnGetConvolutionBackwardMaxCount()**.



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**

The possible error values returned by this function and their meanings are listed below.

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.

4.60. 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 cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for `cudnnConvolutionForward()`, using memory allocated via `cudaMalloc()`, and outputs performance metrics to a user-allocated array of `cudnnConvolutionFwdAlgoPerf_t`. These metrics are written in sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudnnGetConvolutionForwardMaxCount()`.



This function is host blocking.



It is recommend 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.61. 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 available cuDNN algorithms (including CUDNN_TENSOR_OP_MATH and CUDNN_DEFAULT_MATH versions of algorithms where CUDNN_TENSOR_OP_MATH may be available) for **cudnnConvolutionForward**, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of **cudnnConvolutionFwdAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest

compute time. The total number of resulting algorithms can be queried through the API `cudaDnnGetConvolutionForwardMaxCount()`.



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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.62. 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 dhDesc,
    const void             *dh,
    const cudnnTensorDescriptor_t dcDesc,
    const void             *dc,
    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)

```

(New for 7.1)

This function attempts all available cuDNN algorithms for `cudaNNRNNBackwardData`, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of `cudaNNAlgorithmPerformance_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 `cudaNNGetRNNWorkspaceSize()` 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 `cudaNNSetRNNDescriptor` call used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the second dimension should match the `hiddenSize` argument passed to `cudaNNSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the second dimension should match double the `hiddenSize` argument passed to `cudaNNSetRNNDescriptor`.

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 `cudaNNSetRNNDescriptor` call used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the second dimension should match the `hiddenSize` argument passed to `cudaNNSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the second dimension should match double the `hiddenSize` argument passed to `cudaNNSetRNNDescriptor`.

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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 searched 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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.63. 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)

```

(New for 7.1)

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 **cudaGetRNNWorkspaceSize()** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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 searched 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**

The possible error values returned by this function and their meanings are listed below.

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.

4.64. 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)

```

(New for 7.1)

This function attempts all available cuDNN algorithms for **cudaRNNForwardInference**, 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 **cudaGetRNNWorkspaceSize()** 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 **cudaSetRNNDesc** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDesc**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDesc**.

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

cudaSetRNNDescriptor 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 be 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 searched 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.

4.65. 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)

```

(New for 7.1)

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 **cudaGetRNNWorkspaceSize()** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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

cudaSetRNNDescriptor 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 searched 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**

The possible error values returned by this function and their meanings are listed below.

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.

4.66. 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`.

The possible error values returned by this function and their meanings are listed below.

Returns

`CUDNN_STATUS_SUCCESS`

The object was queried successfully.

4.67. cudnnGetAlgorithmDescriptor

```

cudnnStatus_t cudnnGetAlgorithmDescriptor(
    const cudnnAlgorithmDescriptor_t  algoDesc,
    cudnnAlgorithm_t                  *algorithm)

```

(New for 7.1)

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.

4.68. cudnnGetAlgorithmPerformance

```

cudnnStatus_t cudnnGetAlgorithmPerformance(
    const cudnnAlgorithmPerformance_t  algoPerf,
    cudnnAlgorithmDescriptor_t*        algoDesc,
    cudnnStatus_t*                     status,
    float*                               time,
    size_t*                             memory)

```

(New for 7.1)

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.

4.69. cudnnGetAlgorithmSpaceSize

```

cudnnStatus_t cudnnGetAlgorithmSpaceSize(
    cudnnHandle_t      handle,
    cudnnAlgorithmDescriptor_t algoDesc,
    size_t*            algoSpaceSizeInBytes)

```

(New for 7.1)

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

Ouput. 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.

4.70. cudnnBatchNormalizationBackwardExWorkspaceSize

```

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 `cudnnGetBatchNormalizationBackwardEx()` 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. See `cudnnHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). 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.

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**. 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: 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 `cudaGetBatchNormalizationBackwardEx()` function with the specified `bnOps` input setting.

Possible error values returned by this function and their meanings are listed below.

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

4.71. `cudaGetBatchNormalizationForwardTrainingExWorkspaceSize`

```

cudaStatus_t cudaGetBatchNormalizationForwardTrainingExWorkspaceSize(
    cudaHandle_t          handle,
    cudaBatchNormMode_t  mode,
    cudaBatchNormOps_t   bnOps,
    const cudaTensorDescriptor_t xDesc,
    const cudaTensorDescriptor_t zDesc,
    const cudaTensorDescriptor_t yDesc,
    const cudaTensorDescriptor_t bnScaleBiasMeanVarDesc,
    const cudaActivationDescriptor_t activationDesc,
    size_t                *sizeInBytes);

```

This function returns the amount of GPU memory workspace the user should allocate to be able to call `cudaGetBatchNormalizationForwardTrainingEx()` function for the specified `bnOps` input setting. The workspace allocated should then be passed by the user to the function `cudaGetBatchNormalizationForwardTrainingEx()`.

Parameters**handle**

Input. Handle to a previously created cuDNN library descriptor. See `cudaHandle_t`.

mode

Input. Mode of operation (spatial or per-activation). See `cudaBatchNormMode_t`.

bnOps

Input. Mode of operation for the fast NHWC kernel. See `cudaBatchNormOps_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. 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: 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** 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.
- ▶ Dimensions or data types mismatch for **xDesc**, **yDesc**.

4.72. [cudnnGetBatchNormalizationTrainingExReserveSpaceSize](#)

```

cudnnStatus_t cudnnGetBatchNormalizationTrainingExReserveSpaceSize(
    cudnnHandle_t          handle,
    cudnnBatchNormMode_t  mode,
    cudnnBatchNormOps_t   bnOps,
    const cudnnActivationDescriptor_t  activationDesc,
    const cudnnActivationDescriptor_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. See [cudnnHandle_t](#).

mode

Input. Mode of operation (spatial or per-activation). 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.

xDesc

Tensor descriptors and pointers in device memory for the layer's x data. See [cudnnTensorDescriptor_t](#).

activationDesc

Input. Tensor descriptor for the activation operation.

*sizeInBytes

Output. Amount of GPU memory reserved.

Possible error values returned by this function and their meanings are listed below.

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.)

4.73. cudnnGetCTCLossDescriptor

```
cudnnStatus_t cudnnGetCTCLossDescriptor(
    cudnnCTCLossDescriptor_t    ctcLossDesc,
    cudnnDataType_t*           compType)
```

This function returns 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.

4.74. 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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.75. cudnnGetCallback

```

cudnnStatus_t cudnnGetCallback(
    unsigned      mask,
    void          **udata,
    cudnnCallback_t  fptr)

```

(New for 7.1)

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.

4.76. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The operation was successful.

CUDNN_STATUS_BAD_PARAM

The parameter **convDesc** is nil.

4.77. 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 followed:

```

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.

The possible error values returned by this function and their meanings are listed below.

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.

4.78. 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. 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

The possible error values returned by this function and their meanings are listed below.

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.

4.79. 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.

4.80. 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 fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, please 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.81. `cudaGetConvolutionBackwardDataWorkspaceSize`

```

cudaStatus_t cudaGetConvolutionBackwardDataWorkspaceSize(
    cudaHandle_t          handle,
    const cudaFilterDescriptor_t  wDesc,
    const cudaTensorDescriptor_t  dyDesc,
    const cudaConvolutionDescriptor_t  convDesc,
    const cudaTensorDescriptor_t  dxDesc,
    cudaConvolutionBwdDataAlgo_t  algo,
    size_t                *sizeInBytes)

```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call `cudaConvolutionBackwardData` with the specified algorithm. The workspace allocated will then be passed to the routine `cudaConvolutionBackwardData`. The specified algorithm can be the result of the call to `cudaGetConvolutionBackwardDataAlgorithm` 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`

The possible error values returned by this function and their meanings are listed below.

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.

4.82. 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, please 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.83. cudnnGetConvolutionBackwardFilterAlgorithmMaxCount

```

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.

4.84. 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, please 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.85. 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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.86. 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, please 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 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

The possible error values returned by this function and their meanings are listed below.

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.

4.87. 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.

4.88. 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 fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, please 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.89. 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**

The possible error values returned by this function and their meanings are listed below.

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.

4.90. 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.

4.91. 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.

4.92. 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**.

4.93. cudnnGetConvolutionNdForwardOutputDim

```

cudnnStatus_t cudnnGetConvolutionNdForwardOutputDim(
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t inputTensorDesc,
    const cudnnFilterDescriptor_t filterDesc,
    int nbDims,
    int tensorOutputDimA[])

```

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 followed:

```

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

tensorOutputDimA

Output. Array of dimensions **nbDims** that contains on exit of this routine the sizes of the output tensor

The possible error values returned by this function and their meanings are listed below.

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.

4.94. 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.

4.95. cudnnGetDropoutDescriptor

```
cudaStatus_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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The call was successful.

CUDNN_STATUS_BAD_PARAM

One or more of the arguments was an invalid pointer.

4.96. cudnnGetErrorString

```
const char * cudnnGetErrorString(cudnnStatus_t status)
```

This function converts the cuDNN status code to a NUL 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 enumerated status code.

Returns

Pointer to a static, NUL terminated string with the status name.

4.97. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

4.98. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

The parameter **nbDimsRequested** is negative.

4.99. 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).

Possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

Function completed successfully.

4.100. cudnnGetOpTensorDescriptor

```

cudnnStatus_t cudnnGetOpTensorDescriptor(
    const cudnnOpTensorDescriptor_t opTensorDesc,
    cudnnOpTensorOp_t               *opTensorOp,
    cudnnDataType_t                 *opTensorCompType,
    cudnnNanPropagation_t           *opTensorNanOpt)

```

This function returns 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.

4.101. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

4.102. 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 followed:

```
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.

The possible error values returned by this function and their meanings are listed below.

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).

4.103. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was queried successfully.

CUDNN_STATUS_NOT_SUPPORTED

The parameter **nbDimsRequested** is greater than CUDNN_DIM_MAX.

4.104. 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 followed:

```
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.

The possible error values returned by this function and their meanings are listed below.

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**.

4.105. 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. Enumerated 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.

4.106. 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 (i.e. embedding size) of the input or output tensor at each timestep.

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 (i.e. number of timesteps) of each sequence. This is allowed to be a NULL pointer if **arrayLengthRequested** is zero.

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.

4.107. cudnnGetRNNDescrptor

```

cudnnStatus_t cudnnGetRNNDescrptor(
    cudnnHandle_t          handle,
    cudnnRNNDescrptor_t   rnnDesc,
    int *                  hiddenSize,
    int *                  numLayers,
    cudnnDropoutDescrptor_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 `cudnnSetRNNDescrptor()`. 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 where the size of the hidden state should be stored (the same value is used in every layer).

numLayers

Output. Pointer where the number of RNN layers should be stored.

dropoutDesc

Output. Pointer where the handle to a previously configured dropout descriptor should be stored.

inputMode

Output. Pointer where the mode of the first RNN layer should be saved.

direction

Output. Pointer where RNN uni-directional/bi-directional mode should be saved.

mode

Output. Pointer where RNN cell type should be saved.

algo

Output. Pointer where RNN algorithm type should be stored.

dataType

Output. Pointer 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 `cudaGetRNNDescriptor()` function is NULL.

4.108. `cudaGetRNNLinLayerBiasParams`

```

cudaStatus_t cudaGetRNNLinLayerBiasParams (
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             pseudoLayer,
    const cudaTensorDescriptor_t  xDesc,
    const cudaFilterDescriptor_t  wDesc,
    const void            *w,
    const int             linLayerID,
    cudaFilterDescriptor_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 `cudaGetRNNLinLayerBiasParams()` function was changed in cuDNN version 7.1.1 to match the behavior of `cudaGetRNNLinLayerMatrixParams()`.

The `cudaGetRNNLinLayerBiasParams()` 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 returned the total number of vector elements in `linLayerBiasDesc` as follows: `filterDimA[0]=total_size`, `filterDimA[1]=1`, `filterDimA[2]=1` (see the description of the `cudaGetFilterNdDescriptor()` 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 `cudaGetFilterNdDescriptor()`. Note that the RNN implementation in cuDNN uses two bias vectors before the cell non-linear function (see equations in Chapter 3 describing the `cudaRNNMode_t` enumerated type).

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 RNN-s, 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 RNN-s there are twice as many pseudo-layers

in comparison to "physical" layers (pseudoLayer=0 and pseudoLayer=1 are both input layers; pseudoLayer=0 refers to the forward part and 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** a value of 0 references the bias applied to the input from the previous layer, a value of 1 references the bias applied to the recurrent input.
- ▶ 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, value of 4, 5, 6 and 7 reference bias applied to the recurrent input.
 - ▶ 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, value of 3, 4 and 5 reference bias applied to the recurrent input.
 - ▶ 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.

Please refer to [Chapter 3](#) for additional details on modes.

linLayerBiasDesc

Output. Handle to a previously created filter descriptor.

linLayerBias

Output. Data pointer to GPU memory associated with the filter descriptor **linLayerBiasDesc**.

The possible error values returned by this function and their meanings are listed below.

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.

4.109. 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,
const 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, e.g 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 RNN-s, 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 RNN-s there are twice as many pseudo-layers in comparison to "physical" layers (pseudoLayer=0 and pseudoLayer=1 are both input layers; pseudoLayer=0 refers to the forward part and 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** a value of 0 references the matrix multiplication applied to the input from the previous layer, a value of 1 references the matrix multiplication applied to the recurrent input.
- ▶ If **mode** in **rnnDesc** was set to **CUDNN_LSTM** values of 0-3 reference matrix multiplications applied to the input from the previous layer, value of 4-7 reference matrix multiplications applied to the recurrent input.
 - ▶ 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.
 - ▶ Value 8 references the "recurrent" projection matrix when enabled by the `cudaSetRNNProjectionLayers()` function.
- ▶ If **mode** in **rnnDesc** was set to **CUDNN_GRU** values of 0-2 reference matrix multiplications applied to the input from the previous layer, value of 3-5 reference matrix multiplications applied to the recurrent input.
 - ▶ 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.

Please refer to Chapter 3 for additional details on modes.

linLayerMatDesc

Output. Handle to a previously created filter descriptor. When the weight matrix does not exist, the returned filter 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.110. 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.111. 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.

4.112. cudnnGetRNNProjectionLayers

```

cudnnStatus_t cudnnGetRNNProjectionLayers(
    cudnnHandle_t      handle,
    cudnnRNNDescriptor_t rnnDesc,
    int                *recProjSize,
    int                *outProjSize)

```

(New for 7.1)

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.

4.113. 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 **cudaDnnGetRNNWorkspaceSize()** 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.114. 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 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 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.

The possible error values returned by this function and their meanings are listed below.

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.

4.115. cudnnGetReduceTensorDescriptor

```
cudaStatus_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.

4.116. 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.

4.117. 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.

4.118. 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 was 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.

4.119. 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.

The possible error values returned by this function and their meanings are listed below.

Returns

CUDNN_STATUS_SUCCESS

The operation succeeded.

4.120. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The results were returned successfully.

CUDNN_STATUS_BAD_PARAM

Either **tensorDesc** or **nbDims** pointer is NULL.

4.121. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The results were returned successfully.

4.122. 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.

4.123. 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 $\text{batch_size} \times \text{y_height} \times \text{y_width}$ and width of $\text{input_channels} \times \text{filter_height} \times \text{filter_width}$, where batch_size is xDesc 's first dimension, $\text{y_height}/\text{y_width}$ are computed from `cudnnGetConvolutionNdForwardOutputDim()`, input_channels is xDesc 's second dimension, $\text{filter_height}/\text{filter_width}$ are wDesc 's 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.

4.124. 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_LRNCROSSCHANNEL_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: $\text{dstValue} = \text{alpha}[0] * \text{resultValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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).

Possible error values returned by this function and their meanings are listed below.

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 \mathbf{x} , \mathbf{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.

4.125. 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: $\text{dstValue} = \text{alpha}[0] * \text{resultValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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.

Possible error values returned by this function and their meanings are listed below.

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.

4.126. 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 $C = \text{op}(\alpha_1[0] * A, \alpha_2[0] * B) + \beta[0] * C$, given tensors **A**, **B**, and **C** and scaling factors α_1 , α_2 , and β . The op to use is indicated by the descriptor `opTensorDesc`. Currently-supported ops are listed by the `cudnnOpTensorOp_t` enum.

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** must match. If the data type of the destination tensor **C** is double, then the data type of the input tensors also must be double.

If the data type of the destination tensor **C** is double, then `opTensorCompType` in `opTensorDesc` must be double. Else `opTensorCompType` must be float.

If the input tensor **B** is the same tensor as the destination tensor **C**, then the input tensor **A** also must be the same tensor as the destination tensor **C**.



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.

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 indicated by the above op equation. [Please refer to this section for additional details.](#)

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.

The possible error values returned by this function and their meanings are listed below.

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 conditions in the above paragraphs are unmet.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.127. 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: $\text{dstValue} = \text{alpha}[0] * \text{result} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.128. 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 (i.e., 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: $\text{dstValue} = \alpha[0] * \text{result} + \beta[0] * \text{priorDstValue}$. [Refer to this section for additional details.](#)

xDesc

Input. Handle to the previously initialized input tensor descriptor. Must be of type `FLOAT`, or `DOUBLE`, or `HALF`, or `INT8`. 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`, or `DOUBLE`, or `HALF`, or `INT8`. See `cudaDataType_t`.

y

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

The possible error values returned by this function and their meanings are listed below.

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. 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.

4.129. cudnnQueryRuntimeError

```

cudaStatus_t cudnnQueryRuntimeError(
    cudaHandle_t      handle,
    cudaStatus_t      *rstatus,
    cudaErrQueryMode_t mode,
    cudaRuntimeTag_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.

`cudaDnnQueryRuntimeError` can be used in polling and blocking software control flows. There are two polling modes (`CUDNN_ERRQUERY_RAWCODE`, `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 `cudaDnnQueryRuntimeError` 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, i.e., `cudaDnnSetStream` is invoked between functions that report runtime kernel errors and the `cudaDnnQueryRuntimeError` 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 `cudaDnnQueryRuntimeError`.



The `cudaDnnQueryRuntimeError` function should be used in conjunction with `cudaDnnBatchNormalizationForwardTraining` and `cudaDnnBatchNormalizationBackward` when the `cudaDnnBatchNormMode_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.

The possible error values returned by this function and their meanings are listed below.

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.

4.130. 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 *dchDesc,
    const void             *dch,
    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**, **dch**, weights **w** and input gradients **dx**, **dhx**, **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. See [cudnnHandle_t](#).

rnnDesc

Input. A previously initialized RNN descriptor. 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 `cudaGetRNNWorkspaceSize()` 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). See `cudaTensorDescriptor_t`. The second dimension of the tensor depends on the **direction** argument passed to the `cudaSetRNNDescriptor` call used to initialize **rnnDesc**:

- ▶ If **direction** is `CUDNN_UNIDIRECTIONAL` the second dimension should match the **hiddenSize** argument passed to `cudaSetRNNDescriptor`.
- ▶ If **direction** is `CUDNN_BIDIRECTIONAL` the second dimension should match double the **hiddenSize** argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` call used to initialize **rnnDesc**:

- ▶ If **direction** is `CUDNN_UNIDIRECTIONAL` the second dimension should match the **hiddenSize** argument passed to `cudaSetRNNDescriptor`.
- ▶ If **direction** is `CUDNN_BIDIRECTIONAL` the second dimension should match double the **hiddenSize** argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` call used to initialize **rnnDesc**:

- ▶ If **direction** is `CUDNN_UNIDIRECTIONAL` the first dimension should match the **numLayers** argument passed to `cudaSetRNNDescriptor`.
- ▶ If **direction** is `CUDNN_BIDIRECTIONAL` the first dimension should match double the **numLayers** argument passed to `cudaSetRNNDescriptor`.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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. See [cudaFilterDescriptor_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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.131. 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 `n`th 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 (i.e. time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function **cudaRNNBackwardData**, 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 **cudaRNNForwardTrainingEx**.

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 **cudaRNNForwardTrainingEx**.

dyDesc

Input. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, **vectorSize** and **seqLengthArray** need to match the **yDesc** previously passed to **cudaRNNForwardTrainingEx**.

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 **cudaSetRNNDesc** call used to initialize **rnnDesc**. Moreover:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDesc**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDesc**.

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. Moreover:

- ▶ If RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the third dimension must match the `recProjSize` argument passed to `cudaSetRNNProjectionLayers` call used to set `rnnDesc`.
- ▶ Otherwise, the third dimension must match the `hiddenSize` argument passed to the `cudaSetRNNDescriptor` 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 `cudaSetRNNDescriptor` call used to initialize `rnnDesc`. Moreover:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument passed to `cudaSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the first dimension should match double the `numLayers` argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` 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 `cudaRNNForwardTrainingEx`.

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

cudaRNNForwardTrainingEx, or be **NULL** if **NULL** was previously passed to **cudaRNNForwardTrainingEx**.

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 **cudaRNNForwardTrainingEx**.

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 **cudaRNNForwardTrainingEx**, or be **NULL** if **NULL** was previously passed to **cudaRNNForwardTrainingEx**.

dxDesc

Input. A previously initialized RNN data descriptor. The **dataType**, **layout**, **maxSeqLength**, **batchSize**, **vectorSize** and **seqLengthArray** need to match that of **dxDesc** previously passed to **cudaRNNForwardtrainingEx**.

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.

4.132. 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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**

The possible error values returned by this function and their meanings are listed below.

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.

4.133. 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 *n*th 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 (i.e. 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 **cudaDnnRNNForwardTrainingEx**.

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 **cudaDnnRNNForwardTrainingEx**.

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 **cudaDnnRNNForwardTrainingEx**.

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 **cudaDnnRNNForwardTrainingEx**, or be NULL if NULL was previously passed to **cudaDnnRNNForwardTrainingEx**.

yDesc

Input. A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into **cudaDnnRNNForwardTrainingEx**.

y

Input. Data pointer to GPU memory associated with the output tensor descriptor **yDesc**. Must contain the exact same data previously produced by **cudaDnnRNNForwardTrainingEx**.

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.

4.134. 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**, **cx**, weights **w** and outputs **y**, **hy**, **cy**. **workspace** is required for

intermediate storage. This function does not store intermediate data required for training; `cudaRNNForwardTraining` 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 `cudaGetRNNWorkspaceSize()` 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 `cudaSetRNNDescriptor` call used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument passed to `cudaSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the first dimension should match double the `numLayers` argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.

- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.135. 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 *n*th 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 (i.e. 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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. In specific:

- ▶ If RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the third dimension must match the **recProjSize** argument passed to **cudaSetRNNProjectionLayers** call used to set **rnnDesc**.
- ▶ Otherwise, the third dimension must match the **hiddenSize** argument passed to the **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

The second dimension must match the **batchSize** parameter in **xDesc**.
The third dimension must match the **hiddenSize** argument passed to the **cudaSetRNNDescriptor** 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. In specific:

- ▶ For uni-directional network, if RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the parameter **vectorSize** must match the **recProjSize** argument passed to **cudaSetRNNProjectionLayers** call used to set **rnnDesc**. If the network is bidirectional, then multiply the value by 2.
- ▶ Otherwise, for uni-directional network, the parameter **vectorSize** must match the **hiddenSize** argument passed to the **cudaSetRNNDescriptor** 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 be 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.

4.136. 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**, **cx**, weights **w** and outputs **y**, **hy**, **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 iteration (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 `cudaSetRNNDescriptor` call used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument passed to `cudaSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the first dimension should match double the `numLayers` argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` 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 `cudaSetRNNDescriptor` call used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument passed to `cudaSetRNNDescriptor`.
- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` the first dimension should match double the `numLayers` argument passed to `cudaSetRNNDescriptor`.

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 `cudaSetRNNDescriptor` 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the second dimension should match the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the second dimension should match double the **hiddenSize** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.

- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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**

The possible error values returned by this function and their meanings are listed below.

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.

4.137. cudnnRNNForwardTrainingEx

```
cudaStatus_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 **n**th 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 (i.e. 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**. Moreover:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** then the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** then the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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. Moreover:

- ▶ If RNN mode is **CUDNN_LSTM** and **LSTM** projection is enabled, the third dimension must match the **recProjSize** argument passed to **cudaSetRNNProjectionLayers** call used to set **rnnDesc**.
- ▶ Otherwise, the third dimension must match the **hiddenSize** argument passed to the **cudaSetRNNDescriptor** 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 **cudaSetRNNDescriptor** call used to initialize **rnnDesc**. Moreover:

- ▶ If **direction** is **CUDNN_UNIDIRECTIONAL** the first dimension should match the **numLayers** argument passed to **cudaSetRNNDescriptor**.
- ▶ If **direction** is **CUDNN_BIDIRECTIONAL** the first dimension should match double the **numLayers** argument passed to **cudaSetRNNDescriptor**.

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 **cudaSetRNNDescriptor** 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 RNN mode is **CUDNN_LSTM** and whether LSTM projection is enabled and whether the network is bidirectional. In specific:

- ▶ For uni-directional network, if RNN mode is **CUDNN_LSTM** and LSTM projection is enabled, the parameter **vectorSize** must match the **recProjSize** argument passed to **cudaSetRNNProjectionLayers** call used to set **rnnDesc**. If the network is bidirectional, then multiply the value by 2.
- ▶ Otherwise, for uni-directional network, the parameter **vectorSize** must match the **hiddenSize** argument passed to the **cudaSetRNNDescriptor** 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 be 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`, `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.

4.138. 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.

4.139. 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 `cudaRNNSetClip()` 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 `cudaNanPropagation_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.

4.140. `cudaReduceTensor`

```

cudaStatus_t cudaReduceTensor(
    cudaHandle_t          handle,
    const cudaReduceTensorDescriptor_t
    void                *indices,
    size_t              indicesSizeInBytes,
    void                *workspace,
    size_t              workspaceSizeInBytes,
    const void          *alpha,
    const cudaTensorDescriptor_t
    void                aDesc,
    const void          *A,
    const void          *beta,
    const cudaTensorDescriptor_t
    void                cDesc,
    void                *C)

```

This function reduces tensor A by implementing the equation $C = \alpha * \text{reduce op} (A) + \beta * C$, 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 `cudaReduceTensorOp_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 `cudaReduceTensorIndices_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 `cudaIndicesType_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, i.e. 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 indicated by the above op equation. [Please refer to this section for additional details.](#)

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.

The possible error values returned by this function and their meanings are listed below.

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.

4.141. cudnnRestoreAlgorithm

```

cudnnStatus_t cudnnRestoreAlgorithm(
    cudnnHandle_t      handle,
    void*              algoSpace,
    size_t             algoSpaceSizeInBytes,
    cudnnAlgorithmDescriptor_t algoDesc)

```

(New for 7.1)

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.

4.142. 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 `cudaSetDropoutDescriptor`.

The possible error values returned by this function and their meanings are listed below.

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_INVALID_VALUE

States buffer size (as indicated in `stateSizeInBytes`) is too small.

4.143. `cudaSaveAlgorithm`

```
cudaStatus_t cudaSaveAlgorithm(
    cudaHandle_t      handle,
    cudaAlgorithmDescriptor_t algoDesc,
    void*             algoSpace,
    size_t            algoSpaceSizeInBytes)
```

(New for 7.1)

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.

4.144. 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 Host memory to a single value that all elements of the tensor will be scaled with. [Please refer to this section for additional details.](#)

The possible error values returned by this function and their meanings are listed below.

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.

4.145. cudnnSetActivationDescriptor

```

cudnnStatus_t cudnnSetActivationDescriptor(
    cudnnActivationDescriptor_t activationDesc,
    cudnnActivationMode_t      mode,
    cudnnNanPropogation_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 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**.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

mode or **reluNanOpt** has an invalid enumerant value.

4.146. cudnnSetAlgorithmDescriptor

```

cudnnStatus_t cudnnSetAlgorithmDescriptor(
    cudnnAlgorithmDescriptor_t    algorithmDesc,
    cudnnAlgorithm_t              algorithm)

```

(New for 7.1)

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.

4.147. cudnnSetAlgorithmPerformance

```

cudnnStatus_t cudnnSetAlgorithmPerformance(
    cudnnAlgorithmPerformance_t    algoPerf,
    cudnnAlgorithmDescriptor_t      algoDesc,
    cudnnStatus_t                  status,

```

```
float           time,
size_t         memory)
```

(New for 7.1)

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 enumerant value.

4.148. cudnnSetCTCLossDescriptor

```
cudaStatus_t cudnnSetCTCLossDescriptor(
    cudnnCTCLossDescriptor_t   ctcLossDesc,
    cudnnDataType_t           compType)
```

This function sets a CTC loss function descriptor.

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.

4.149. cudnnSetCallback

```

cudnnStatus_t cudnnSetCallback(
    unsigned      mask,
    void         *udata,
    cudnnCallback_t  fptr)

```

(New for 7.1)

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 (see Section 2.11).

`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 `CUDNN_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 `cudaDebug_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 `"\n"`, and the end of message is terminated by `"\n\n"`. User may select what is necessary to show in the log, and may reformat the string.

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

4.150. cudnnSetConvolution2dDescriptor

```

cudaStatus_t cudnnSetConvolution2dDescriptor(
    cudaConvolutionDescriptor_t    convDesc,
    int                            pad_h,
    int                            pad_w,
    int                            u,
    int                            v,
    int                            dilation_h,
    int                            dilation_w,
    cudaConvolutionMode_t          mode,
    cudaDataType_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.

The possible error values returned by this function and their meanings are listed below.

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.

4.151. 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

4.152. 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 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.

4.153. 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 datatype in which the computation will be done.

The possible error values returned by this function and their meanings are listed below.

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.

4.154. 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 **states** argument is equal to NULL, random number generator states won't be initialized, and only **dropout** value will be set. No other function should be writing to the memory pointed at by **states** argument while this function is running. The user is expected not to change 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 size in bytes of the provided memory for the states

seed

Input. Seed used to initialize random number generator states.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The call was successful.

CUDNN_STATUS_INVALID_VALUE

stateSizeInBytes is less than the value returned by `cudaDnnDropoutGetStatesSize`.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU

4.155. `cudaDnnSetFilter4dDescriptor`

```

cudaDnnStatus_t cudaDnnSetFilter4dDescriptor(
    cudaDnnFilterDescriptor_t  filterDesc,
    cudaDnnDataType_t         dataType,
    cudaDnnTensorFormat_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 `cudaDnnConvolutionForward`, `cudaDnnConvolutionBackwardData` and `cudaDnnConvolutionBackwardFilter`; please refer to the documentation for each function for more information.

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 enumerated values allowed by `cudaDnnTensorFormat_t` descriptor, then the layout of the filter is in the form of KCRS (K represents the number of output feature maps, C the number of input feature maps, R the number of rows per filter, and S 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. See also the description for `cudaTensorFormat_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.

The possible error values returned by this function and their meanings are listed below.

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.

4.156. cudnnSetFilterNdDescriptor

```

cudaStatus_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 `cudaConvolutionForward`, `cudaConvolutionBackwardData` and `cudaConvolutionBackwardFilter`; please refer to the documentation for each function for more information.

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 enumerated values allowed by **cudaTensorFormat_t** descriptor, then the layout of the filter is as follows:

- ▶ For N=4, i.e., for a 4D filter descriptor, the filter layout is in the form of KCRS (K represents the number of output feature maps, C the number of input feature maps, R the number of rows per filter, and S the number of columns per filter.)
- ▶ For N=3, i.e., for 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, i.e., for a 4D filter descriptor, the filter layout is in the form of KRSC.
- ▶ For N=3, i.e., for 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. See also the description for **cudaTensorFormat_t**.

nbDims

Input. Dimension of the filter.

filterDimA

Input. Array of dimension **nbDims** containing the size of the filter for each dimension.

The possible error values returned by this function and their meanings are listed below.

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.

4.157. cudnnSetLRNDescriptor

```

cudaStatus_t cudnnSetLRNDescriptor(
    cudaLRNDescriptor_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 `cuda.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 `cudaCreateLRNDescriptor`.

`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 $(\text{window width})^{\#\text{spatialDimensions}}$ for DivisiveNormalization. By default this value is set to $1e-4$ in `cudaCreateLRNDescriptor`.

`lrnBeta`

Input. Value of the beta power parameter in the normalization formula. By default this value is set to 0.75 in `cudaCreateLRNDescriptor`.

`lrnK`

Input. Value of the k parameter in normalization formula. By default this value is set to 2.0.

Possible error values returned by this function and their meanings are listed below.

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.

4.158. `cudaSetOpTensorDescriptor`

```
cudaStatus_t cudaSetOpTensorDescriptor(
    cudaOpTensorDescriptor_t opTensorDesc,
    cudaOpTensorOp_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.

4.159. 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`.

4.160. 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.

The possible error values returned by this function and their meanings are listed below.

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 enumerant value.

4.161. 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 (i.e., negative striding size is not allowed).

Returns

CUDNN_STATUS_SUCCESS

The object was initialized successfully.

CUDNN_STATUS_NOT_SUPPORTED

If ($\text{nbDims} > \text{CUDNN_DIM_MAX} - 2$).

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 enumerant value.

4.162. 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. See [cudnnRNNDataDescriptor_t](#).

dataType

Input. The datatype of the RNN data tensor. 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 (i.e. embedding size) of the input or output tensor at each timestep.

seqLengthArray

Input. An integer array with **batchSize** number of elements. Describes the length (i.e. number of timesteps) 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 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**, **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.

4.163. 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    dataType)
  
```

This function initializes a previously created RNN descriptor object.



Larger networks (e.g., longer sequences, 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. (e.g., bidirectional).

mode

Input. Specifies the type of RNN to compute.

dataType

Input. Math precision.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

Either at least one of the parameters **hiddenSize**, **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

4.164. 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        dataType)

```

This function initializes a previously created RNN descriptor object.



Larger networks (e.g., longer sequences, 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 (e.g., 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. (e.g., bidirectional)

mode

Input. Specifies the type of RNN to compute.

dataType

Input. Compute precision.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

Either at least one of the parameters **hiddenSize**, **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, **algo**, **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

4.165. cudnnSetRNNDescriptor_v6

```

cudnnStatus_t cudnnSetRNNDescriptor_v6(
    cudnnHandle_t      handle,
    cudnnRNNDescriptor_t  rnnDesc,
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnRNNInputMode_t  inputMode,
    cudnnDirectionMode_t direction,
    cudnnRNNMode_t       mode,
    cudnnRNNAlgo_t       algo,
    cudnnDataType_t      dataType)

```

This function initializes a previously created RNN descriptor object.



Larger networks (e.g., longer sequences, 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 (e.g., 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. (e.g., bidirectional)

mode

Input. Specifies the type of RNN to compute.

algo

Input. Specifies which RNN algorithm should be used to compute the results.

dataType

Input. Compute precision.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

Either at least one of the parameters **hiddenSize**, **numLayers** was zero or negative, one of **inputMode**, **direction**, **mode**, **algo**, **dataType** has an invalid enumerant value, **dropoutDesc** is an invalid dropout descriptor or **rnnDesc** has not been created correctly.

4.166. cudnnSetRNNMatrixMathType

```

cudnnStatus_t cudnnSetRNNMatrixMathType(
    cudnnRNNDescriptor_t    rnnDesc,
    cudnnMathType_t         mType)

```

This function sets the preferred option to use NVIDIA Tensor Cores accelerators on Volta GPU-s (SM 7.0 or higher). When the **mType** parameter is **CUDNN_TENSOR_OP_MATH**, inference and training RNN API-s 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 GEMM-s (general matrix-matrix multiplications). This option has an “advisory” status meaning that Tensor Cores may not be utilized, e.g., 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.

4.167. 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. See the description for **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.

4.168. cudnnSetRNNProjectionLayers

```

cudnnStatus_t cudnnSetRNNProjectionLayers(
    cudnnHandle_t    handle,
    cudnnRNNDescriptor_t  rnnDesc,

```

```
int         recProjSize,
int         outProjSize)
```

(New for 7.1)

The `cudaSetRNNProjectionLayers()` function should be called after `cudaSetRNNDescrptor()` 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 the chapter describing the `cudaRNNMode_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 `cudaSetRNNDescrptor()` 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 `cudaSetRNNDescrptor()`.

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 (e.g., 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.

4.169. 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).

4.170. 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.

The possible error values returned by this function and their meanings are listed below.

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

4.171. 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.

4.172. 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**.

The possible error values returned by this function and their meanings are listed below.

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. 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAMAt 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 maximim limit of 2 Giga-elements.

4.174. 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 Tensor4D 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.

The possible error values returned by this function and their meanings are listed below.

Returns**CUDNN_STATUS_SUCCESS**

The object was set successfully.

CUDNN_STATUS_BAD_PARAMAt 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.

4.175. 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. See also the `cudaGetRNNLinLayerBiasParams()`.

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.

The possible error values returned by this function and their meanings are listed below.

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.

4.176. 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. See also the `cudaGetRNNLinLayerBiasParams()`.

dimA

Input. Array containing 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.

4.177. 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; i.e., `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: $\text{dstValue} = \text{alpha}[0] * \text{result} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

`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`.

The possible error values returned by this function and their meanings are listed below.

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.

4.178. 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: $\text{dstValue} = \alpha[0] * \text{result} + \beta[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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**.

The possible error values returned by this function and their meanings are listed below.

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.

4.179. 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.

The possible error values returned by this function and their meanings are listed below.

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**, **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.

4.180. 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 .

The possible error values returned by this function and their meanings are listed below.

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**, **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.

4.181. 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: $\text{dstValue} = \text{alpha}[0] * \text{srcValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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: $\text{dstValue} = \text{alpha}[0] * \text{srcValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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 **cudaSpatialTfGridGeneratorForward**.

dgrid

Output. Data pointer to GPU memory contains the output differential data.

The possible error values returned by this function and their meanings are listed below.

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.

4.182. 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: $\text{dstValue} = \text{alpha}[0] * \text{srcValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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 **cudaSpatialTfGridGeneratorForward**.**yDesc***Input.* Handle to the previously initialized output tensor descriptor.**y***Output.* Data pointer to GPU memory associated with the output tensor descriptor **yDesc**.

The possible error values returned by this function and their meanings are listed below.

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**, **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.

4.183. 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 (i.e., 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: $\text{dstValue} = \text{alpha}[0] * \text{srcValue} + \text{beta}[0] * \text{priorDstValue}$. [Please refer to this section for additional details.](#)

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.

The possible error values returned by this function and their meanings are listed below.

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.

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

Copyright

© 2018 NVIDIA Corporation. All rights reserved.