



# NVIDIA cuDNN

API Reference | NVIDIA Docs

# Table of Contents

Chapter 1. Introduction.....	1
Chapter 2. Added, Deprecated, and Removed API Functions.....	2
2.1. API Changes for cuDNN 8.7.0.....	2
2.2. API Changes for cuDNN 8.5.0.....	2
2.3. API Changes for cuDNN 8.4.0.....	3
2.4. API Changes for cuDNN 8.3.0.....	3
2.5. API Changes for cuDNN 8.2.0.....	4
2.6. API Changes for cuDNN 8.1.0.....	4
2.7. API Changes for cuDNN 8.0.3.....	4
2.8. API Changes for cuDNN 8.0.2.....	5
2.9. API Changes for cuDNN 8.0.0 Preview.....	5
Chapter 3. cudnn_ops_infer.so Library.....	10
3.1. Data Type References.....	10
3.1.1. Pointer To Opaque Struct Types.....	10
3.1.2. Enumeration Types.....	12
3.2. API Functions.....	26
Chapter 4. cudnn_ops_train.so Library.....	103
4.1. API Functions.....	103
Chapter 5. cudnn_cnn_infer.so Library.....	148
5.1. Data Type References.....	148
5.1.1. Pointer To Opaque Struct Types.....	148
5.1.2. Struct Types.....	148
5.1.3. Enumeration Types.....	150
5.2. API Functions.....	153
Chapter 6. cudnn_cnn_train.so Library.....	204
6.1. Data Type References.....	204
6.1.1. Pointer To Opaque Struct Types.....	204
6.1.2. Struct Types.....	205
6.1.3. Enumeration Types.....	205
6.2. API Functions.....	238
Chapter 7. cudnn_adv_infer.so Library.....	261
7.1. Data Type References.....	261
7.1.1. Pointer To Opaque Struct Types.....	261
7.1.2. Enumeration Types.....	262

7.2. API Functions.....	269
<b>Chapter 8. cudnn_adv_train.so Library.....</b>	<b>344</b>
8.1. Data Type References.....	344
8.1.1. Enumeration Types.....	344
8.2. API Functions.....	345
<b>Chapter 9. cuDNN Backend API.....</b>	<b>407</b>
9.1. Data Type References.....	407
9.1.1. Enumeration Types.....	407
9.1.2. Data Types Found In cudnn_backend.h.....	422
9.2. API Functions.....	423
9.3. Backend Descriptor Types.....	428
9.4. Use Cases.....	472
9.4.1. Setting Up An Operation Graph For A Grouped Convolution.....	472
9.4.2. Setting Up An Engine Configuration.....	474
9.4.3. Setting Up And Executing A Plan.....	475

# List of Figures

Figure 1. Operations Per Channel For CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS.....	206
Figure 2. Multi-Head Attention Block is Tightly Coupled with the Layer Normalization and Residual Connections.....	307
Figure 3. Locations of x, y, hx, cx, hy, and cy Signals in the Multi-Layer RNN Model.....	310
Figure 4. Data Flow when the RNN Model is Bidirectional.....	311
Figure 5. Data Layout Example for cudnnSetSeqDataDescriptor().....	341
Figure 6. Locations of x, y, hx, cx, hy, cy, dx, dy, dhx, dcx, dhy, and dcy Signals a Multi- Layer RNN Model.....	379

# List of Tables

Table 1.	API functions and data types that were added in cuDNN 8.7.0.....	2
Table 2.	API functions and data types that were added in cuDNN 8.5.0.....	2
Table 3.	API functions and data types that were added in cuDNN 8.4.0.....	3
Table 4.	API functions and data types that were added in cuDNN 8.3.0.....	3
Table 5.	API functions and data types that were added in cuDNN 8.2.0.....	4
Table 6.	API functions and data types that were added in cuDNN 8.1.0.....	4
Table 7.	API functions and data types that were added in cuDNN 8.0.3.....	4
Table 8.	API functions and data types that were added in cuDNN 8.0.2.....	5
Table 9.	API functions and data types that were added in cuDNN 8.0.0 Preview.....	5
Table 10.	API functions and data types that were deprecated in cuDNN 8.0.0 Preview.....	7
Table 11.	API functions and data types that were removed in cuDNN 8.0.0 Preview.....	9
Table 12.	Supported Configurations for cudnnBatchNormalizationForwardInference().....	30
Table 13.	Supported Datatypes for cudnnOpTensor().....	68
Table 14.	Supported Configurations for cudnnBatchNormalizationBackward().....	107
Table 15.	Supported Configurations for cudnnBatchNormalizationBackwardEx().....	111
Table 16.	Supported Configurations for cudnnBatchNormalizationForwardTraining().....	114
Table 17.	Supported Configurations for cudnnBatchNormalizationForwardTrainingEx().....	118
Table 18.	Supported Configurations for cudnnNormalizationForwardTraining().....	140
Table 19.	Supported Configurations for cudnnConvolutionBackwardData().....	154
Table 20.	Supported Algorithms for cudnnConvolutionBackwardData() 2D Convolutions: wDesc: _NHWC.....	156
Table 21.	Supported Algorithms for cudnnConvolutionBackwardData() 2D Convolutions: wDesc: _NCHW.....	156
Table 22.	Supported Algorithms for cudnnConvolutionBackwardData() 3D Convolutions: wDesc: _NCHW.....	160

Table 23. Supported Algorithms for cudnnConvolutionBackwardData() 3D Convolutions: wDesc: _NHWC.....	162
Table 24. Supported Combinations of Data Types (X = CUDNN_DATA) for cudnnConvolutionBiasActivationForward().....	165
Table 25. Supported Configurations for cudnnConvolutionForward().....	168
Table 26. Supported Algorithms for cudnnConvolutionForward() 2D Convolutions: wDesc: _NCHW.....	171
Table 27. Supported Algorithms for cudnnConvolutionForward() 2D Convolutions: wDesc: _NCHWC.....	174
Table 28. Supported Algorithms for cudnnConvolutionForward() 2D Convolutions: wDesc: _NHWC.....	175
Table 29. Supported Algorithms for cudnnConvolutionForward() 3D Convolutions: wDesc: _NCHW.....	176
Table 30. Supported Algorithms for cudnnConvolutionForward() 3D Convolutions: wDesc: _NHWC.....	177
Table 31. Legend For Tables in cudnnFusedOpsConstParamLabel_t.....	208
Table 32. CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS In cudnnFusedOpsConstParamLabel_t.....	208
Table 33. Conditions for Fully Fused Fast Path (Forward) for cudnnFusedOpsConstParamLabel_t.....	212
Table 34. CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in cudnnFusedOpsConstParamLabel_t.....	213
Table 35. Conditions for Fully Fused Fast Path (Backward) for cudnnFusedOpsConstParamLabel_t.....	216
Table 36. CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOpsConstParamLabel_t.....	217
Table 37. CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOpsConstParamLabel_t.....	221
Table 38. CUDNN_FUSED_CONVOLUTION_SCALE_BIAS_ADD_RELU in cudnnFusedOpsConstParamLabel_t.....	224
Table 39. Legend For Tables in cudnnFusedOpsVariantParamLabel_t.....	227

Table 40. CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in <code>cudaFusedOpsVariantParamLabel_t</code> .....	228
Table 41. CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD in <code>cudaFusedOpsVariantParamLabel_t</code> .....	229
Table 42. CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in <code>cudaFusedOpsVariantParamLabel_t</code> .....	231
Table 43. CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in <code>cudaFusedOpsVariantParamLabel_t</code> .....	234
Table 44. CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in <code>cudaFusedOpsVariantParamLabel_t</code> .....	236
Table 45. Supported Configurations for <code>cudaConvolutionBackwardFilter()</code> .....	241
Table 46. Supported Algorithms for <code>cudaConvolutionBackwardFilter()</code> 2D Convolutions: <code>dwDesc: _NHWC</code> .....	242
Table 47. Supported Algorithms for <code>cudaConvolutionBackwardFilter()</code> 2D Convolutions: <code>dwDesc: _NCHW</code> .....	242
Table 48. Supported Algorithms for <code>cudaConvolutionBackwardFilter()</code> 3D Convolutions: <code>dwDesc: _NCHW</code> .....	246
Table 49. Supported Algorithms for <code>cudaConvolutionBackwardFilter()</code> 3D Convolutions: <code>dwDesc: _NHWC</code> .....	247
Table 50. Supported Combinations for <code>cudaSetAttnDescriptor()</code> .....	329
Table 51. Attribute Types for <code>cudaBackendAttributeType_t</code> .....	412
Table 52. BN Statistics for <code>cudaBnFinalizeStatsMode_t</code> .....	417
Table 53. CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR for Zero Batch Dimensions.....	445
Table 54. CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR for a Single Batch Dimension.....	445
Table 55. CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR for a Three Batch Dimension.....	445
Table 56. Supported Configurations for CUDNN_BACKEND_OPERATION_NORM_BACKWARD_DESCRIPTOR.....	449
Table 57. Supported Configurations for CUDNN_BACKEND_OPERATION_NORM_FORWARD_DESCRIPTOR.....	451





---

# Chapter 1. Introduction

NVIDIA® CUDA® Deep Neural Network (cuDNN) library offers a context-based API that allows for easy multithreading and (optional) interoperability with CUDA streams. This API Reference lists the datatypes and functions per library. Specifically, this reference consists of a cuDNN datatype reference section that describes the types of enums and a cuDNN API reference section that describes all routines in the cuDNN library API.

The cuDNN library as well as this API document has been split into the following libraries:

## **cuda\_ops\_infer**

This entity contains the routines related to cuDNN context creation and destruction, tensor descriptor management, tensor utility routines, and the inference portion of common machine learning algorithms such as batch normalization, softmax, dropout, and so on.

## **cuda\_ops\_train**

This entity contains common training routines and algorithms, such as batch normalization, softmax, dropout, and so on. The `cuda_ops_train` library depends on `cuda_ops_infer`.

## **cuda\_cnn\_infer**

This entity contains all routines related to convolutional neural networks needed at inference time. The `cuda_cnn_infer` library depends on `cuda_ops_infer`.

## **cuda\_cnn\_train**

This entity contains all routines related to convolutional neural networks needed during training time. The `cuda_cnn_train` library depends on `cuda_ops_infer`, `cuda_ops_train`, and `cuda_cnn_infer`.

## **cuda\_adv\_infer**

This entity contains all other features and algorithms. This includes RNNs, CTC loss, and multi-head attention. The `cuda_adv_infer` library depends on `cuda_ops_infer`.

## **cuda\_adv\_train**

This entity contains all the training counterparts of `cuda_adv_infer`. The `cuda_adv_train` library depends on `cuda_ops_infer`, `cuda_ops_train`, and `cuda_adv_infer`.

## **cudaBackend\***

Introduced in cuDNN version 8.x, this entity contains a list of valid cuDNN backend descriptor types, a list of valid attributes, a subset of valid attribute values, and a full description of each backend descriptor type and their attributes.

## **cuda**

This is an optional shim layer between the application layer and the cuDNN code. This layer opportunistically opens the correct library for the API at runtime.

---

# Chapter 2. Added, Deprecated, and Removed API Functions

## 2.1. API Changes for cuDNN 8.7.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.7.0.

Table 1. API functions and data types that were added in cuDNN 8.7.0

Backend descriptor types
<a href="#"><u>cudaRngDistribution_t</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_RNG_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_RNG_DESCRIPTOR</u></a>

## 2.2. API Changes for cuDNN 8.5.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.5.0.

Table 2. API functions and data types that were added in cuDNN 8.5.0

Backend descriptor types
<a href="#"><u>cudaBackendNormFwdPhase_t</u></a>
<a href="#"><u>cudaBackendNormMode_t</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_CONCAT_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_NORM_BACKWARD_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_NORM_FORWARD_DESCRIPTOR</u></a>

**Backend descriptor types**[CUDNN\\_BACKEND\\_OPERATION\\_SIGNAL\\_DESCRIPTOR](#)[cudaFraction\\_t](#)[cudaSignalMode\\_t](#)

## 2.3. API Changes for cuDNN 8.4.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.4.0.

Table 3. API functions and data types that were added in cuDNN 8.4.0

<b>Backend descriptor types</b>
<a href="#"><u>cudaBackendBehaviorNote_t</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_POINTWISE_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_REDUCTION_DESCRIPTOR</u></a>
<a href="#"><u>cudaBackendTensorReordering_t</u></a>
<a href="#"><u>cudaBnFinalizeStatsMode_t</u></a>
<a href="#"><u>cudaPaddingMode_t</u></a>
<a href="#"><u>cudaResampleMode_t</u></a>

## 2.4. API Changes for cuDNN 8.3.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.3.0.

Table 4. API functions and data types that were added in cuDNN 8.3.0

<b>Backend descriptor types</b>
<a href="#"><u>CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR</u></a>
<a href="#"><u>CUDNN_BACKEND_RESAMPLE_DESCRIPTOR</u></a>

## 2.5. API Changes for cuDNN 8.2.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.2.0.

Table 5. API functions and data types that were added in cuDNN 8.2.0

<b>New functions</b>
<a href="#"><u>cudaDnnGetActivationDescriptorSwishBeta()</u></a>
<a href="#"><u>cudaDnnSetActivationDescriptorSwishBeta()</u></a>

## 2.6. API Changes for cuDNN 8.1.0

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.1.0.

Table 6. API functions and data types that were added in cuDNN 8.1.0

<b>Backend descriptor types</b>
<a href="#"><u>CUDA_DNN_BACKEND_MATMUL_DESCRIPTOR</u></a>
<a href="#"><u>CUDA_DNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR</u></a>

## 2.7. API Changes for cuDNN 8.0.3

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.0.3.

Table 7. API functions and data types that were added in cuDNN 8.0.3

<b>Backend descriptor types</b>
<a href="#"><u>CUDA_DNN_BACKEND_CONVOLUTION_DESCRIPTOR</u></a>
<a href="#"><u>CUDA_DNN_BACKEND_ENGINE_DESCRIPTOR</u></a>
<a href="#"><u>CUDA_DNN_BACKEND_ENGINECFG_DESCRIPTOR</u></a>
<a href="#"><u>CUDA_DNN_BACKEND_ENGINEHEUR_DESCRIPTOR</u></a>
<a href="#"><u>CUDA_DNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR</u></a>

**Backend descriptor types**[CUDNN\\_BACKEND\\_INTERMEDIATE\\_INFO\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_KNOB\\_CHOICE\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_KNOB\\_INFO\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_LAYOUT\\_INFO\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATION\\_CONVOLUTION\\_BACKWARD\\_DATA\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATION\\_CONVOLUTION\\_BACKWARD\\_FILTER\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATION\\_CONVOLUTION\\_FORWARD\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATION\\_GEN\\_STATS\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATION\\_POINTWISE\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_OPERATIONGRAPH\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_TENSOR\\_DESCRIPTOR](#)[CUDNN\\_BACKEND\\_VARIANT\\_PACK\\_DESCRIPTOR](#)

## 2.8. API Changes for cuDNN 8.0.2

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.0.2.

Table 8. API functions and data types that were added in cuDNN 8.0.2

**New functions and data types**[cudnnRNNBackwardData\\_v8\(\)](#)[cudnnRNNBackwardWeights\\_v8\(\)](#)

## 2.9. API Changes for cuDNN 8.0.0 Preview

The following tables show which API functions were added, deprecated, and removed for the cuDNN 8.0.0 Preview Release.

Table 9. API functions and data types that were added in cuDNN 8.0.0 Preview

**New functions and data types**[cudnnAdvInferVersionCheck\(\)](#)[cudnnAdvTrainVersionCheck\(\)](#)

## New functions and data types

[cudaDnnBackendAttributeName\\_t](#)

[cudaDnnBackendAttributeType\\_t](#)

[cudaDnnBackendCreateDescriptor\(\)](#)

[cudaDnnBackendDescriptor\\_t](#)

[cudaDnnBackendDescriptorType\\_t](#)

[cudaDnnBackendDestroyDescriptor\(\)](#)

[cudaDnnBackendExecute\(\)](#)

[cudaDnnBackendFinalize\(\)](#)

[cudaDnnBackendGetAttribute\(\)](#)

[cudaDnnBackendHeurMode\\_t](#)

[cudaDnnBackendInitialize\(\)](#)

[cudaDnnBackendKnobType\\_t](#)

[cudaDnnBackendLayoutType\\_t](#)

[cudaDnnBackendNumericalNote\\_t](#)

[cudaDnnBackendSetAttribute\(\)](#)

[cudaDnnBuildRNNDynamic\(\)](#)

[cudaDnnCTCLoss\\_v8\(\)](#)

[cudaDnnDeriveNormTensorDescriptor\(\)](#)

[cudaDnnForwardMode\\_t](#)

[cudaDnnGenStatsMode\\_t](#)

[cudaDnnGetCTCLossDescriptor\\_v8\(\)](#)

[cudaDnnGetCTCLossDescriptorEx\(\)](#)

[cudaDnnGetCTCLossWorkspaceSize\\_v8](#)

[cudaDnnGetFilterSizeInBytes\(\)](#)

[cudaDnnGetFoldedConvBackwardDataDescriptors\(\)](#)

[cudaDnnGetNormalizationBackwardWorkspaceSize\(\)](#)

[cudaDnnGetNormalizationForwardTrainingWorkspaceSize\(\)](#)

[cudaDnnGetNormalizationTrainingReserveSpaceSize\(\)](#)

[cudaDnnGetRNNDescriptor\\_v8\(\)](#)

[cudaDnnGetRNNMatrixMathType\(\)](#)

[cudaDnnGetRNNTempSpaceSizes\(\)](#)

[cudaDnnGetRNNWeightParams\(\)](#)

[cudaDnnGetRNNWeightSpaceSize\(\)](#)

[cudaDnnLRNDescriptor\\_t](#)

[cudaDnnNormAlgo\\_t](#)

[cudaDnnNormalizationBackward\(\)](#)

[cudaDnnNormalizationForwardInference\(\)](#)

[cudaDnnNormalizationForwardTraining\(\)](#)

**New functions and data types**

<a href="#">cudnnNormMode_t</a>
<a href="#">cudnnNormOps_t</a>
<a href="#">cudnnOpsInferVersionCheck()</a>
<a href="#">cudnnOpsTrainVersionCheck()</a>
<a href="#">cudnnPointwiseMode_t</a>
<a href="#">cudnnRNNForward()</a>
<a href="#">cudnnRNNGetClip_v8()</a>
<a href="#">cudnnRNNSetClip_v8()</a>
<a href="#">cudnnSetCTCLossDescriptor_v8()</a>
<a href="#">cudnnSetRNNDescriptor_v8()</a>
<a href="#">cudnnSeverity_t</a>

For our deprecation policy, refer to the [Backward Compatibility And Deprecation Policy](#).

Table 10. API functions and data types that were deprecated in cuDNN 8.0.0 Preview

Deprecated functions and data types	Replaced with
<code>cudnnCopyAlgorithmDescriptor()</code>	
<code>cudnnCreateAlgorithmDescriptor()</code>	
<code>cudnnCreatePersistentRNNPlan()</code>	<a href="#">cudnnBuildRNNDynamic()</a>
<code>cudnnDestroyAlgorithmDescriptor()</code>	
<code>cudnnDestroyPersistentRNNPlan()</code>	
<code>cudnnFindRNNBackwardDataAlgorithmEx()</code>	
<code>cudnnFindRNNBackwardWeightsAlgorithmEx()</code>	
<code>cudnnFindRNNForwardInferenceAlgorithmEx()</code>	
<code>cudnnFindRNNForwardTrainingAlgorithmEx()</code>	
<code>cudnnGetAlgorithmDescriptor()</code>	
<code>cudnnGetAlgorithmPerformance()</code>	
<code>cudnnGetAlgorithmSpaceSize()</code>	
<code>cudnnGetRNNBackwardDataAlgorithmMaxCount()</code>	
<code>cudnnGetRNNBackwardWeightsAlgorithmMaxCount()</code>	
<ul style="list-style-type: none"> <li>▶ <code>cudnnGetRNNDescriptor_v6()</code></li> <li>▶ <code>cudnnGetRNNMatrixMathType()</code></li> <li>▶ <code>cudnnGetRNNBiasMode()</code></li> <li>▶ <code>cudnnGetRNNPaddingMode()</code></li> <li>▶ <code>cudnnGetRNNProjectionLayers()</code></li> </ul>	<a href="#">cudnnGetRNNDescriptor_v8()</a>

Deprecated functions and data types	Replaced with
<code>cudaGetRNNForwardInferenceAlgorithmMaxCount()</code>	
<code>cudaGetRNNForwardTrainingAlgorithmMaxCount()</code>	
<ul style="list-style-type: none"> <li>▶ <code>cudaGetRNNLinLayerBiasParams()</code></li> <li>▶ <code>cudaGetRNNLinLayerMatrixParams()</code></li> </ul>	<a href="#"><code>cudaGetRNNWeightParams()</code></a>
<code>cudaGetRNNParamsSize()</code>	<a href="#"><code>cudaGetRNNWeightSpaceSize()</code></a>
<ul style="list-style-type: none"> <li>▶ <code>cudaGetRNNWorkspaceSize()</code></li> <li>▶ <code>cudaGetRNNTrainingReserveSize()</code></li> </ul>	<a href="#"><code>cudaGetRNNTempSpaceSizes()</code></a>
<code>cudaPersistentRNNPlan_t</code>	
<code>cudaRestoreAlgorithm()</code>	
<ul style="list-style-type: none"> <li>▶ <code>cudaRNNBackwardData()</code></li> <li>▶ <code>cudaRNNBackwardDataEx()</code></li> </ul>	<a href="#"><code>cudaRNNBackwardData_v8()</code></a>
<ul style="list-style-type: none"> <li>▶ <code>cudaRNNBackwardWeights()</code></li> <li>▶ <code>cudaRNNBackwardWeightsEx()</code></li> </ul>	<a href="#"><code>cudaRNNBackwardWeights_v8()</code></a>
<ul style="list-style-type: none"> <li>▶ <code>cudaRNNForwardInference()</code></li> <li>▶ <code>cudaRNNForwardInferenceEx()</code></li> <li>▶ <code>cudaRNNForwardTraining()</code></li> <li>▶ <code>cudaRNNForwardTrainingEx()</code></li> </ul>	<a href="#"><code>cudaRNNForward()</code></a>
<code>cudaRNNGetClip()</code>	<a href="#"><code>cudaRNNGetClip_v8()</code></a>
<code>cudaRNNSetClip()</code>	<a href="#"><code>cudaRNNSetClip_v8()</code></a>
<code>cudaSaveAlgorithm()</code>	
<code>cudaSetAlgorithmDescriptor()</code>	
<code>cudaSetAlgorithmPerformance()</code>	
<code>cudaSetPersistentRNNPlan()</code>	
<code>cudaSetRNNAlgorithmDescriptor()</code>	
<ul style="list-style-type: none"> <li>▶ <code>cudaSetRNNBiasMode()</code></li> <li>▶ <code>cudaSetRNNDescriptor_v6()</code></li> <li>▶ <code>cudaSetRNNMatrixMathType()</code></li> <li>▶ <code>cudaSetRNNPaddingMode()</code></li> <li>▶ <code>cudaSetRNNProjectionLayers()</code></li> </ul>	<a href="#"><code>cudaSetRNNDescriptor_v8()</code></a>



Table 11. API functions and data types that were removed in cuDNN 8.0.0 Preview

Removed functions and data types
<code>cudaConvolutionBwdDataPreference_t</code>
<code>cudaConvolutionBwdFilterPreference_t</code>
<code>cudaConvolutionFwdPreference_t</code>
<code>cudaGetConvolutionBackwardDataAlgorithm()</code>
<code>cudaGetConvolutionBackwardFilterAlgorithm()</code>
<code>cudaGetConvolutionForwardAlgorithm()</code>
<code>cudaGetRNNDescriptor()</code>
<code>cudaSetRNNDescriptor()</code>

---

# Chapter 3. `cuda_ops_infer.so` Library

This entity contains the routines related to cuDNN context creation and destruction, tensor descriptor management, tensor utility routines, and the inference portion of common machine learning algorithms such as batch normalization, softmax, dropout, and so on.

## 3.1. Data Type References

These are the data type references in the `cuda_ops_infer.so` library.

### 3.1.1. Pointer To Opaque Struct Types

These are the pointers to the opaque struct types in the `cuda_ops_infer.so` library.

#### 3.1.1.1. `cudaActivationDescriptor_t`

`cudaActivationDescriptor_t` is a pointer to an opaque structure holding the description of an activation operation. [`cudaCreateActivationDescriptor\(\)`](#) is used to create one instance, and [`cudaSetActivationDescriptor\(\)`](#) must be used to initialize this instance.

#### 3.1.1.2. `cudaCTCLossDescriptor_t`

`cudaCTCLossDescriptor_t` is a pointer to an opaque structure holding the description of a CTC loss operation. [`cudaCreateCTCLossDescriptor\(\)`](#) is used to create one instance, [`cudaSetCTCLossDescriptor\(\)`](#) is used to initialize this instance, and [`cudaDestroyCTCLossDescriptor\(\)`](#) is used to destroy this instance.

#### 3.1.1.3. `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,

[cudnnGetDropoutDescriptor\(\)](#) is used to query fields of a previously initialized instance, [cudnnRestoreDropoutDescriptor\(\)](#) is used to restore an instance to a previously saved off state.

### 3.1.1.4. [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.

### 3.1.1.5. [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.

### 3.1.1.6. [cudnnLRNDescriptor\\_t](#)

`cudnnLRNDescriptor_t` is a pointer to an opaque structure holding the parameters of a local response normalization. [cudnnCreateLRNDescriptor\(\)](#) is used to create one instance, and the routine [cudnnSetLRNDescriptor\(\)](#) must be used to initialize this instance.

### 3.1.1.7. [cudnnOpTensorDescriptor\\_t](#)

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

### 3.1.1.8. [cudnnPoolingDescriptor\\_t](#)

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

### 3.1.1.9. [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.

### 3.1.1.10. `cudaSpatialTransformerDescriptor_t`

`cudaSpatialTransformerDescriptor_t` is a pointer to an opaque structure holding the description of a spatial transformation operation. [`cudaCreateSpatialTransformerDescriptor\(\)`](#) is used to create one instance, [`cudaSetSpatialTransformerNdDescriptor\(\)`](#) is used to initialize this instance, and [`cudaDestroySpatialTransformerDescriptor\(\)`](#) is used to destroy this instance.

### 3.1.1.11. `cudaTensorDescriptor_t`

`cudaTensorDescriptor_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.

### 3.1.1.12. `cudaTensorTransformDescriptor_t`

`cudaTensorTransformDescriptor_t` is an opaque structure containing the description of the tensor transform. Use the [`cudaCreateTensorTransformDescriptor\(\)`](#) function to create an instance of this descriptor, and [`cudaDestroyTensorTransformDescriptor\(\)`](#) function to destroy a previously created instance.

## 3.1.2. Enumeration Types

These are the enumeration types in the `cuda_ops_infer.so` library.

### 3.1.2.1. `cudaActivationMode_t`

`cudaActivationMode_t` is an enumerated type used to select the neuron activation function used in [`cudaActivationForward\(\)`](#), [`cudaActivationBackward\(\)`](#), and [`cudaConvolutionBiasActivationForward\(\)`](#).

#### Values

**CUDNN\_ACTIVATION\_SIGMOID**

Selects the sigmoid function.

**CUDNN\_ACTIVATION\_RELU**

Selects the rectified linear function.

**CUDNN\_ACTIVATION\_TANH**

Selects the hyperbolic tangent function.

**CUDNN\_ACTIVATION\_CLIPPED\_RELU**

Selects the clipped rectified linear function.

**CUDNN\_ACTIVATION\_ELU**

Selects the exponential linear function.

**CUDNN\_ACTIVATION\_IDENTITY**

Selects the identity function, intended for bypassing the activation step in [cudnnConvolutionBiasActivationForward\(\)](#). (The [cudnnConvolutionBiasActivationForward\(\)](#) function must use CUDNN\_CONVOLUTION\_FWD\_ALGO\_IMPLICIT\_PRECOMP\_GEMM.) Does not work with [cudnnActivationForward\(\)](#) or [cudnnActivationBackward\(\)](#).

**CUDNN\_ACTIVATION\_SWISH**

Selects the swish function.

### 3.1.2.2. [cudnnAlgorithm\\_t](#)

This function has been deprecated in cuDNN 8.0.

### 3.1.2.3. [cudnnBatchNormMode\\_t](#)

[cudnnBatchNormMode\\_t](#) is an enumerated type used to specify the mode of operation in [cudnnBatchNormalizationForwardInference\(\)](#), [cudnnBatchNormalizationForwardTraining\(\)](#), [cudnnBatchNormalizationBackward\(\)](#) and [cudnnDeriveBNTensorDescriptor\(\)](#) routines.

#### Values

**CUDNN\_BATCHNORM\_PER\_ACTIVATION**

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

**CUDNN\_BATCHNORM\_SPATIAL**

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

**CUDNN\_BATCHNORM\_SPATIAL\_PERSISTENT**

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

An optimized path may be selected for `CUDNN_DATA_FLOAT` and `CUDNN_DATA_HALF` types, compute capability 6.0 or higher for the following two batch normalization API calls: [cudnnBatchNormalizationForwardTraining\(\)](#) and [cudnnBatchNormalizationBackward\(\)](#). In the case of [cudnnBatchNormalizationBackward\(\)](#), the `savedMean` and `savedInvVariance` arguments should not be `NULL`.

The rest of this section applies to `NCHW` mode only: This mode may use a scaled atomic integer reduction that is deterministic but imposes more restrictions on the input data range. When a numerical overflow occurs, the algorithm may produce NaN-s or Inf-s (infinity) in output buffers.

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

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

### 3.1.2.4. `cudaBatchNormOps_t`

`cudaBatchNormOps_t` is an enumerated type used to specify the mode of operation in `cudaGetBatchNormalizationForwardTrainingExWorkspaceSize()`, `cudaBatchNormalizationForwardTrainingEx()`, `cudaGetBatchNormalizationBackwardExWorkspaceSize()`, `cudaBatchNormalizationBackwardEx()`, and `cudaGetBatchNormalizationTrainingExReserveSpaceSize()` 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.

### 3.1.2.5. `cudaCTCLossAlgo_t`

`cudaCTCLossAlgo_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.

### 3.1.2.6. `cudaDataType_t`

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

#### Values

**CUDNN\_DATA\_FLOAT**

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

**CUDNN\_DATA\_DOUBLE**

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

**CUDNN\_DATA\_HALF**

The data is a 16-bit floating-point.

**CUDNN\_DATA\_INT8**

The data is an 8-bit signed integer.

**CUDNN\_DATA\_INT32**

The data is a 32-bit signed integer.

**CUDNN\_DATA\_INT8x4**

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

**CUDNN\_DATA\_UINT8**

The data is an 8-bit unsigned integer.

**CUDNN\_DATA\_UINT8x4**

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

**CUDNN\_DATA\_INT8x32**

The data is 32-element vectors, each element being an 8-bit signed integer. This data type is only supported with the tensor format `CUDNN_TENSOR_NCHW_VECT_C`.

Moreover, this data type can only be used with `algo 1`, meaning,

`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`. For more information, refer to [cudnnConvolutionFwdAlgo\\_t](#).

**CUDNN\_DATA\_BFLOAT16**

The data is a 16-bit quantity, with 7 mantissa bits, 8 exponent bits, and 1 sign bit.

**CUDNN\_DATA\_INT64**

The data is a 64-bit signed integer.

**CUDNN\_DATA\_BOOLEAN**

The data is a boolean (`bool`).

Note that for type `CUDNN_TYPE_BOOLEAN`, elements are expected to be "packed": that is, one byte contains 8 elements of type `CUDNN_TYPE_BOOLEAN`. Further, within each byte, elements are indexed from the least significant bit to the most significant bit. For example, a 1 dimensional tensor of 8 elements containing 01001111 has value 1 for elements 0 through 3, 0 for elements 4 and 5, 1 for element 6 and 0 for element 7.

Tensors with more than 8 elements simply use more bytes, where the order is also from least significant to most significant byte. Note, CUDA is little-endian, meaning that the least significant byte has the lower memory address. For example, in the case of 16 elements, 01001111 11111100 has value 1 for elements 0 through 3, 0 for elements 4 and 5, 1 for element 6 and 0 for element 7, value 0 for elements 8 and 9, 1 for elements 10 through 15.

**CUDNN\_DATA\_FP8\_E4M3**

The data is an 8-bit quantity, with 3 mantissa bits, 4 exponent bits, and 1 sign bit.

**CUDNN\_DATA\_FP8\_E5M2**

The data is an 8-bit quantity, with 2 mantissa bits, 5 exponent bits, and 1 sign bit.

**CUDNN\_DATA\_FAST\_FLOAT\_FOR\_FP8**

The data type is a higher throughput but lower precision compute type (compared to CUDNN\_DATA\_FLOAT) used for FP8 tensor core operations

### 3.1.2.7. cudnnDeterminism\_t

cudnnDeterminism\_t is an enumerated type used to indicate if the computed results are deterministic (reproducible). For more information, refer to [Reproducibility \(Determinism\)](#).

#### Values

**CUDNN\_NON\_DETERMINISTIC**

Results are not guaranteed to be reproducible.

**CUDNN\_DETERMINISTIC**

Results are guaranteed to be reproducible.


### 3.1.2.8. cudnnDivNormMode\_t

cudnnDivNormMode\_t is an enumerated type used to specify the mode of operation in [cudnnDivisiveNormalizationForward\(\)](#) and [cudnnDivisiveNormalizationBackward\(\)](#).

#### Values

**CUDNN\_DIVNORM\_PRECOMPUTED\_MEANS**

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

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

### 3.1.2.9. cudnnErrQueryMode\_t

cudnnErrQueryMode\_t is an enumerated type passed to [cudnnQueryRuntimeError\(\)](#) to select the remote kernel error query mode.

#### Values

**CUDNN\_ERRQUERY\_RAWCODE**

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



**CUDNN\_ERRQUERY\_NONBLOCKING**

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

**CUDNN\_ERRQUERY\_BLOCKING**

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

**3.1.2.10. cudnnFoldingDirection\_t**

`cudnnFoldingDirection_t` is an enumerated type used to select the folding direction. For more information, refer to [cudnnTensorTransformDescriptor\\_t](#).

**Data Member**

**CUDNN\_TRANSFORM\_FOLD = 0U**

Selects folding.

**CUDNN\_TRANSFORM\_UNFOLD = 1U**

Selects unfolding.

**3.1.2.11. cudnnIndicesType\_t**

`cudnnIndicesType_t` is an enumerated type used to indicate the data type for the indices to be computed by the [cudnnReduceTensor\(\)](#) routine. This enumerated type is used as a field for the [cudnnReduceTensorDescriptor\\_t](#) descriptor.

**Values**

**CUDNN\_32BIT\_INDICES**

Compute unsigned int indices.

**CUDNN\_64BIT\_INDICES**

Compute unsigned long indices.

**CUDNN\_16BIT\_INDICES**

Compute unsigned short indices.

**CUDNN\_8BIT\_INDICES**

Compute unsigned char indices.

**3.1.2.12. 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 the tensor's dimension `dimA[1]`.

### 3.1.2.13. cudnnMathType\_t

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

#### Values

##### **CUDNN\_DEFAULT\_MATH**

Tensor Core operations are not used on pre-NVIDIA A100 GPU devices. On A100 GPU architecture devices, Tensor Core TF32 operation is permitted.

##### **CUDNN\_TENSOR\_OP\_MATH**

The use of Tensor Core operations is permitted but will not actively perform datatype down conversion on tensors in order to utilize Tensor Cores.

##### **CUDNN\_TENSOR\_OP\_MATH\_ALLOW\_CONVERSION**

The use of Tensor Core operations is permitted and will actively perform datatype down conversion on tensors in order to utilize Tensor Cores.

##### **CUDNN\_FMA\_MATH**

Restricted to only kernels that use FMA instructions.

On pre-NVIDIA A100 GPU devices, `CUDNN_DEFAULT_MATH` and `CUDNN_FMA_MATH` have the same behavior: Tensor Core kernels will not be selected. With NVIDIA Ampere architecture and CUDA toolkit 11, `CUDNN_DEFAULT_MATH` permits TF32 Tensor Core operation and `CUDNN_FMA_MATH` does not. The TF32 behavior for `CUDNN_DEFAULT_MATH` and the other Tensor Core math types can be explicitly disabled by the environment variable `NVIDIA_TF32_OVERRIDE=0`.

### 3.1.2.14. 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.

### 3.1.2.15. cudnnNormAlgo\_t

`cudnnNormAlgo_t` is an enumerated type used to specify the algorithm to execute the normalization operation.

#### Values

##### **CUDNN\_NORM\_ALGO\_STANDARD**

Standard normalization is performed.

### CUDNN\_NORM\_ALGO\_PERSIST

This mode is similar to CUDNN\_NORM\_ALGO\_STANDARD, however it only supports CUDNN\_NORM\_PER\_CHANNEL and 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 normalization API calls: [cudnnNormalizationForwardTraining\(\)](#) and [cudnnNormalizationBackward\(\)](#). In the case of [cudnnNormalizationBackward\(\)](#), the `savedMean` and `savedInvVariance` arguments should not be NULL.

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

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

For finite but very large input values, the algorithm may encounter overflows more frequently due to a lower dynamic range and emit Inf-s/NaN-s while CUDNN\_NORM\_ALGO\_STANDARD will produce finite results. The user can invoke [cudnnQueryRuntimeError\(\)](#) to check if a numerical overflow occurred in this mode.

### 3.1.2.16. cudnnNormMode\_t

`cudnnNormMode_t` is an enumerated type used to specify the mode of operation in [cudnnNormalizationForwardInference\(\)](#), [cudnnNormalizationForwardTraining\(\)](#), [cudnnBatchNormalizationBackward\(\)](#), [cudnnGetNormalizationForwardTrainingWorkspaceSize\(\)](#), [cudnnGetNormalizationBackwardWorkspaceSize\(\)](#), and [cudnnGetNormalizationTrainingReserveSpaceSize\(\)](#) routines.

#### Values

##### CUDNN\_NORM\_PER\_ACTIVATION

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

##### CUDNN\_NORM\_PER\_CHANNEL

Normalization is performed per-channel over N+spatial dimensions. This mode is intended for use after convolutional layers (where spatial invariance is desired). In this mode, the `normBias` and `normScale` tensor dimensions are 1xCx1x1.

### 3.1.2.17. cudnnNormOps\_t

`cudnnNormOps_t` is an enumerated type used to specify the mode of operation in [cudnnGetNormalizationForwardTrainingWorkspaceSize\(\)](#),

[cudaNormalizationForwardTraining\(\)](#),  
[cudaGetNormalizationBackwardWorkspaceSize\(\)](#), [cudaNormalizationBackward\(\)](#),  
and [cudaGetNormalizationTrainingReserveSpaceSize\(\)](#) functions.

## Values

**CUDNN\_NORM\_OPS\_NORM**

Only normalization is performed.

**CUDNN\_NORM\_OPS\_NORM\_ACTIVATION**

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

**CUDNN\_NORM\_OPS\_NORM\_ADD\_ACTIVATION**

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

### 3.1.2.18. [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

**CUDNN\_OP\_TENSOR\_ADD**

The operation to be performed is addition.

**CUDNN\_OP\_TENSOR\_MUL**

The operation to be performed is multiplication.

**CUDNN\_OP\_TENSOR\_MIN**

The operation to be performed is a minimum comparison.

**CUDNN\_OP\_TENSOR\_MAX**

The operation to be performed is a maximum comparison.

**CUDNN\_OP\_TENSOR\_SQRT**

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

**CUDNN\_OP\_TENSOR\_NOT**

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

### 3.1.2.19. [cudaPoolingMode\\_t](#)

[cudaPoolingMode\\_t](#) is an enumerated type passed to [cudaSetPooling2dDescriptor\(\)](#) 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.

### 3.1.2.20. **cudaReduceTensorIndices\_t**

`cudaReduceTensorIndices_t` is an enumerated type used to indicate whether indices are to be computed by the `cudaReduceTensor()` routine. This enumerated type is used as a field for the `cudaReduceTensorDescriptor_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.

### 3.1.2.21. **cudaReduceTensorOp\_t**

`cudaReduceTensorOp_t` is an enumerated type used to indicate the Tensor Core operation to be used by the `cudaReduceTensor()` routine. This enumerated type is used as a field for the `cudaReduceTensorDescriptor_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 the sum of squares.

**CUDNN\_REDUCE\_TENSOR\_MUL\_NO\_ZEROS**

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

**3.1.2.22. cudnnRNNAIgo\_t**

`cudnnRNNAIgo_t` is an enumerated type used to specify the algorithm used in the [cudnnRNNForwardInference\(\)](#), [cudnnRNNForwardTraining\(\)](#), [cudnnRNNBackwardData\(\)](#) and [cudnnRNNBackwardWeights\(\)](#) routines.

**Values****CUDNN\_RNN\_ALGO\_STANDARD**

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

**CUDNN\_RNN\_ALGO\_PERSIST\_STATIC**

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

`CUDNN_RNN_ALGO_PERSIST_STATIC` is only supported on devices with compute capability  $\geq 6.0$ .

**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 (meaning, a small minibatch). When using `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` persistent kernels are prepared at runtime and are able to optimize using the specific parameters of the network and active GPU. As such, when using `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` a one-time plan preparation stage must be executed. These plans can then be reused in repeated calls with the same model parameters.

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

`CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is only supported on devices with compute capability  $\geq 6.0$  on Linux machines.

**3.1.2.23. 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.

## 3.1.2.24. **cudnnSeverity\_t**

`cudnnSeverity_t` is an enumerated type passed to the customized callback function for logging that users may set. This enumerate describes the severity level of the item, so the customized logging call back may react differently.

## Values

### **CUDNN\_SEV\_FATAL**

This value indicates a fatal error emitted by cuDNN.

### **CUDNN\_SEV\_ERROR**

This value indicates a normal error emitted by cuDNN.

### **CUDNN\_SEV\_WARNING**

This value indicates a warning emitted by cuDNN.

### **CUDNN\_SEV\_INFO**

This value indicates a piece of information (for example, API log) emitted by cuDNN.

## 3.1.2.25. **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`.

## 3.1.2.26. **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 dimension  $c$ .

**3.1.2.27. cudnnStatus\_t**

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

**Values****CUDNN\_STATUS\_SUCCESS**

The operation was completed successfully.

**CUDNN\_STATUS\_NOT\_INITIALIZED**

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

**CUDNN\_STATUS\_ALLOC\_FAILED**

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

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

**CUDNN\_STATUS\_BAD\_PARAM**

An incorrect value or parameter was passed to the function.

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

**CUDNN\_STATUS\_ARCH\_MISMATCH**

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

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

**CUDNN\_STATUS\_MAPPING\_ERROR**

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

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

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

**CUDNN\_STATUS\_EXECUTION\_FAILED**

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



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

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

**CUDNN\_STATUS\_INTERNAL\_ERROR**

An internal cuDNN operation failed.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The functionality requested is not presently supported by cuDNN.

**CUDNN\_STATUS\_LICENSE\_ERROR**

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

**CUDNN\_STATUS\_RUNTIME\_PREREQUISITE\_MISSING**

A runtime library required by cuDNN cannot be found in the predefined search paths. These libraries are `libcuda.so` (`nvcuda.dll`) and `libnVRTC.so` (`nVRTC64_<Major Release Version><Minor Release Version>_0.dll` and `nVRTC-builtins64_<Major Release Version><Minor Release Version>.dll`).

**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.

### 3.1.2.28. `cudaTensorFormat_t`

`cudaTensorFormat_t` is an enumerated type used by `cudaSetTensor4dDescriptor()` to create a tensor with a pre-defined layout. For a detailed explanation of how these tensors are arranged in memory, refer to [Data Layout Formats](#).

#### Values

**CUDNN\_TENSOR\_NCHW**

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

**CUDNN\_TENSOR\_NHWC**

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

### CUDNN\_TENSOR\_NCHW\_VECT\_C

This tensor format specifies that the data is laid out in the following order: batch size, feature maps, rows, columns. However, each element of the tensor is a vector of multiple feature maps. The length of the vector is carried by the data type of the tensor. The strides are implicitly defined in such a way that the data are contiguous in memory with no padding between images, feature maps, rows, and columns; the columns are the inner dimension and the images are the outermost dimension. This format is only supported with tensor data types CUDNN\_DATA\_INT8x4, CUDNN\_DATA\_INT8x32, and CUDNN\_DATA\_UINT8x4.

The CUDNN\_TENSOR\_NCHW\_VECT\_C can also be interpreted in the following way: The NCHW INT8x32 format is really  $N \times (C/32) \times H \times W \times 32$  (32 Cs for every W), just as the NCHW INT8x4 format is  $N \times (C/4) \times H \times W \times 4$  (4 Cs for every W). Hence, the VECT\_C name - each W is a vector (4 or 32) of Cs.

## 3.2. API Functions

These are the API functions in the cuda\_ops\_infer.so library.

### 3.2.1. cudnnActivationForward()

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

```

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)
    
```

In-place operation is allowed for this routine; meaning, xData and yData pointers may be equal. However, this requires xDesc and yDesc descriptors to be identical (particularly, the strides of the input and output must match for an in-place operation to be allowed).

All tensor formats are supported for 4 and 5 dimensions, however, the best performance is obtained when the strides of xDesc and yDesc are equal and HW-packed. For more than 5 dimensions the tensors must have their spatial dimensions packed.

#### Parameters

##### handle

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

##### activationDesc

*Input.* Activation descriptor. For more information, refer to [cudnnActivationDescriptor\\_t](#).

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc**

*Input.* Handle to the previously initialized input tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**x**

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

**yDesc**

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

**y**

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

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

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

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**3.2.2. cudnnAddTensor ()**

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.

```
cudaStatus_t cudaAddTensor(
    cudaHandle_t          handle,
    const void            *alpha,
    const cudaTensorDescriptor_t aDesc,
    const void            *A,
    const void            *beta,
```

```
const cudnnTensorDescriptor_t  cDesc,
void                            *C)
```

Only 4D and 5D tensors are supported. Beyond these dimensions, this routine is not supported.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

### alpha, beta

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

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

For more information, refer to [Scaling Parameters](#).

### aDesc

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudnnTensorDescriptor\\_t](#).

### A

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

### cDesc

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

### C

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The function executed successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration.

### CUDNN\_STATUS\_BAD\_PARAM

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

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

## 3.2.3. cudnnBatchNormalizationForwardInference ()

This function performs the forward batch normalization layer computation for the inference phase. This layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```
cudnnStatus_t cudnnBatchNormalizationForwardInference(
    cudnnHandle_t          handle,
    cudnnBatchNormMode_t  mode,
```

```

const void          *alpha,
const void          *beta,
const cudnnTensorDescriptor_t xDesc,
const void          *x,
const cudnnTensorDescriptor_t yDesc,
void               *y,
const cudnnTensorDescriptor_t bnScaleBiasMeanVarDesc,
const void         *bnScale,
const void         *bnBias,
const void         *estimatedMean,
const void         *estimatedVariance,
double             epsilon)

```

Only 4D and 5D tensors are supported.

The input transformation performed by this function is defined as:

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

For the training phase, refer to [cudnnBatchNormalizationForwardTraining\(\)](#).

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

For more information, refer to [cudnnDeriveBNTensorDescriptor\(\)](#) for the secondary tensor descriptor generation for the parameters used in this function.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

### mode

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [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:

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

For more information, refer to [Scaling Parameters](#).

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

### bnScaleBiasMeanVarDesc, bnScale, bnBias

*Inputs.* Tensor descriptors and pointers in device memory for the batch normalization scale and bias parameters (in the [Batch Normalization: Accelerating Deep Network](#)

[Training by Reducing Internal Covariate Shift](#) 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. Its value should be equal to or greater than the value defined for `CUDNN_BN_MIN_EPSILON` in `cuda.h`.

**Supported configurations**

This function supports the following combinations of data types for various descriptors.

Table 12. Supported Configurations for `cudaBatchNormalizationForwardInference()`

Data Type Configurations	xDesc	bnScaleBiasMean	alpha, beta	yDesc
INT8_CONFIG	CUDNN_DATA_INT8	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_INT8
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_BFLOAT16

**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`, and `estimatedInvVariance` is NULL.
- ▶ The number of `xDesc` or `yDesc` tensor descriptor dimensions is not within the range of [4, 5] (only 4D and 5D tensors are supported.)
- ▶ `bnScaleBiasMeanVarDesc` dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.

- ▶ `epsilon` value is less than `CUDNN_BN_MIN_EPSILON`.
- ▶ Dimensions or data types mismatch for `xDesc`, `yDesc`.

### 3.2.4. `cudaCopyAlgorithmDescriptor()`

This function has been deprecated in cuDNN 8.0.

### 3.2.5. `cudaCreate()`

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.

```
cudaStatus_t cudaCreate(cudaHandle_t *handle)
```

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

For a given device, multiple cuDNN handles with different configurations (for example, different current CUDA streams) may be created. Because `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.

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

#### Parameters

##### **handle**

*Output.* Pointer to pointer where to store the address to the allocated cuDNN handle.

For more information, refer to [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.

### 3.2.6. **cudaCreateActivationDescriptor()**

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

[cudaActivationDescriptor\\_t](#).

```
cudaStatus_t cudaCreateActivationDescriptor(
    cudaActivationDescriptor_t *activationDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.7. **cudaCreateAlgorithmDescriptor()**

This function has been deprecated in cuDNN 8.0.

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

```
cudaStatus_t cudaCreateAlgorithmDescriptor(
    cudaAlgorithmDescriptor_t *algoDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.8. **cudaCreateAlgorithmPerformance()**

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

```
cudaStatus_t cudaCreateAlgorithmPerformance(
    cudaAlgorithmPerformance_t *algoPerf,
    int numberToCreate)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.



### 3.2.9. cudnnCreateDropoutDescriptor()

This function creates a generic dropout descriptor object by allocating the memory needed to hold its opaque structure. For more information, refer to [cudnnDropoutDescriptor\\_t](#).

```
cudaStatus_t cudnnCreateDropoutDescriptor(
    cudnnDropoutDescriptor_t *dropoutDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.10. cudnnCreateFilterDescriptor()

This function creates a filter descriptor object by allocating the memory needed to hold its opaque structure. For more information, refer to [cudnnFilterDescriptor\\_t](#).

```
cudaStatus_t cudnnCreateFilterDescriptor(
    cudnnFilterDescriptor_t *filterDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.11. cudnnCreateLRNDescriptor()

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.

```
cudaStatus_t cudnnCreateLRNDescriptor(
    cudnnLRNDescriptor_t *poolingDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.12. cudnnCreateOpTensorDescriptor()

This function creates a tensor pointwise math descriptor. For more information, refer to [cudnnOpTensorDescriptor\\_t](#).

```
cudaStatus_t cudnnCreateOpTensorDescriptor(
```

```
cudaOpTensorDescriptor_t* opTensorDesc)
```

## 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.

## 3.2.13. cudnnCreatePoolingDescriptor()

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

```
cudaStatus_t cudnnCreatePoolingDescriptor(
    cudaPoolingDescriptor_t* poolingDesc)
```

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was created successfully.

### CUDNN\_STATUS\_ALLOC\_FAILED

The resources could not be allocated.

## 3.2.14. cudnnCreateReduceTensorDescriptor()

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

```
cudaStatus_t cudnnCreateReduceTensorDescriptor(
    cudaReduceTensorDescriptor_t* reduceTensorDesc)
```

## 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.

### 3.2.15. cudnnCreateSpatialTransformerDescriptor()

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

```
cudaStatus_t cudnnCreateSpatialTransformerDescriptor(
    cudaSpatialTransformerDescriptor_t *stDesc)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### 3.2.16. cudnnCreateTensorDescriptor()

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

```
cudaStatus_t cudnnCreateTensorDescriptor(
    cudaTensorDescriptor_t *tensorDesc)
```

#### Parameters

**tensorDesc**

*Output.* 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.

### 3.2.17. cudnnCreateTensorTransformDescriptor()

This function creates a tensor transform descriptor object by allocating the memory needed to hold its opaque structure. The tensor data is initialized to be all zero. Use the [cudnnSetTensorTransformDescriptor\(\)](#) function to initialize the descriptor created by this function.

```
cudaStatus_t cudnnCreateTensorTransformDescriptor(
    cudaTensorTransformDescriptor_t *transformDesc);
```

#### Parameters

**transformDesc**

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The descriptor object was created successfully.

### CUDNN\_STATUS\_BAD\_PARAM

The transformDesc is NULL.

### CUDNN\_STATUS\_ALLOC\_FAILED

The memory allocation failed.

## 3.2.18. cudnnDeriveBNTensorDescriptor()

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

```

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

Use the tensor descriptor produced by this function as the `bnScaleBiasMeanVarDesc` parameter for the [cudnnBatchNormalizationForwardInference\(\)](#) and [cudnnBatchNormalizationForwardTraining\(\)](#) functions, and as the `bnScaleBiasDiffDesc` parameter in the [cudnnBatchNormalizationBackward\(\)](#) function.

The resulting dimensions will be:

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

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



### Note:

- ▶ Only 4D and 5D tensors are supported.
- ▶ The `derivedBnDesc` should be first created using [cudnnCreateTensorDescriptor\(\)](#).
- ▶ `xDesc` is the descriptor for the layer's `x` data and has to be set up 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.

## 3.2.19. cudnnDeriveNormTensorDescriptor()

This function derives tensor descriptors for the normalization `mean`, `invariance`, `normBias`, and `normScale` subtensors from the layer's `x` data descriptor and `norm` mode. `normalization`, `mean`, and `invariance` share the same descriptor while `bias` and `scale` share the same descriptor.

```

cudnnStatus_t CUDNNWINAPI
cudnnDeriveNormTensorDescriptor(cudnnTensorDescriptor_t derivedNormScaleBiasDesc,
                                cudnnTensorDescriptor_t derivedNormMeanVarDesc,
                                const cudnnTensorDescriptor_t xDesc,
                                cudnnNormMode_t mode,
                                int groupCnt)

```

Use the tensor descriptor produced by this function as the `normScaleBiasDesc` or `normMeanVarDesc` parameter for the [cudnnNormalizationForwardInference\(\)](#) and [cudnnNormalizationForwardTraining\(\)](#) functions, and as the `dNormScaleBiasDesc` and `normMeanVarDesc` parameters in the [cudnnNormalizationBackward\(\)](#) function.

The resulting dimensions will be:

- ▶ 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for `CUDNN_NORM_PER_ACTIVATION`
- ▶ 1xCxHxW for 4D and 1xCxDxHxW for 5D for `CUDNN_NORM_PER_CHANNEL` 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 `derivedNormScaleBiasDesc` and `derivedNormMeanVarDesc` should be created first using [cudnnCreateTensorDescriptor\(\)](#).
- ▶ `xDesc` is the descriptor for the layer's `x` data and has to be set up with proper dimensions prior to calling this function.

## Parameters

### derivedNormScaleBiasDesc

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

### derivedNormMeanVarDesc

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

### xDesc

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

### mode

*Input.* The normalization layer mode of operation.

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid batch normalization mode.

**3.2.20. cudnnDestroy()**

This function releases the resources used by the cuDNN handle. This function is usually the last call made to cuDNN with a particular handle. Because `cudnnCreate()` allocates 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.

```
cudaStatus_t cudnnDestroy(cudaHandle_t handle)
```

**Parameters**

**handle**

*Input.* The cuDNN handle to be destroyed.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The cuDNN context destruction was successful.

**3.2.21. cudnnDestroyActivationDescriptor()**

This function destroys a previously created activation descriptor object.

```
cudaStatus_t cudnnDestroyActivationDescriptor(
    cudnnActivationDescriptor_t activationDesc)
```

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

**3.2.22. cudnnDestroyAlgorithmDescriptor()**

This function has been deprecated in cuDNN 8.0.

This function destroys a previously created algorithm descriptor object.

```
cudaStatus_t cudnnDestroyAlgorithmDescriptor(
    cudnnActivationDescriptor_t algorithmDesc)
```

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.23. cudnnDestroyAlgorithmPerformance ()

This function destroys a previously created algorithm descriptor object.

```
cudaStatus_t cudnnDestroyAlgorithmPerformance(
    cudaAlgorithmPerformance_t algoPerf)
```

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.24. cudnnDestroyDropoutDescriptor ()

This function destroys a previously created dropout descriptor object.

```
cudaStatus_t cudnnDestroyDropoutDescriptor(
    cudaDropoutDescriptor_t dropoutDesc)
```

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.25. cudnnDestroyFilterDescriptor ()

This function destroys a filter object.

```
cudaStatus_t cudnnDestroyFilterDescriptor(
    cudaFilterDescriptor_t filterDesc)
```

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.26. cudnnDestroyLRNDescriptor ()

This function destroys a previously created LRN descriptor object.

```
cudaStatus_t cudnnDestroyLRNDescriptor(
    cudaLRNDescriptor_t lrnDesc)
```

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.27. cudnnDestroyOpTensorDescriptor()

This function deletes a tensor pointwise math descriptor object.

```
cudaStatus_t cudnnDestroyOpTensorDescriptor(
    cudnnOpTensorDescriptor_t opTensorDesc)
```

#### 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.

### 3.2.28. cudnnDestroyPoolingDescriptor()

This function destroys a previously created pooling descriptor object.

```
cudaStatus_t cudnnDestroyPoolingDescriptor(
    cudnnPoolingDescriptor_t poolingDesc)
```

#### Returns

##### CUDNN\_STATUS\_SUCCESS

The object was destroyed successfully.

### 3.2.29. cudnnDestroyReduceTensorDescriptor()

This function destroys a previously created reduce tensor descriptor object. When the input pointer is NULL, this function performs no destroy operation.

```
cudaStatus_t cudnnDestroyReduceTensorDescriptor(
    cudnnReduceTensorDescriptor_t tensorDesc)
```

#### Parameters

##### tensorDesc

*Input.* Pointer to the reduce tensor descriptor object to be destroyed.

#### Returns

##### CUDNN\_STATUS\_SUCCESS

The object was destroyed successfully.

### 3.2.30. cudnnDestroySpatialTransformerDescriptor()

This function destroys a previously created spatial transformer descriptor object.

```
cudaStatus_t cudnnDestroySpatialTransformerDescriptor(
    cudnnSpatialTransformerDescriptor_t stDesc)
```



## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.31. cudnnDestroyTensorDescriptor()

This function destroys a previously created tensor descriptor object. When the input pointer is `NULL`, this function performs no destroy operation.

```
cudaStatus_t cudnnDestroyTensorDescriptor(cudaTensorDescriptor_t tensorDesc)
```

## Parameters

**tensorDesc**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was destroyed successfully.

### 3.2.32. cudnnDestroyTensorTransformDescriptor()

Destroys a previously created tensor transform descriptor.

```
cudaStatus_t cudnnDestroyTensorTransformDescriptor(
    cudaTensorTransformDescriptor_t transformDesc);
```

## Parameters

**transformDesc**

*Input.* The tensor transform descriptor to be destroyed.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The descriptor was destroyed successfully.

### 3.2.33. cudnnDivisiveNormalizationForward()

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 the [What is the Best Multi-Stage Architecture for Object Recognition](#) paper. Note that `DivisiveNormalization` only implements the  $x/\max(c, \sigma_x)$  portion of the computation, where  $\sigma_x$  is the variance over the spatial neighborhood of  $x$ .

```
cudaStatus_t cudnnDivisiveNormalizationForward(
    cudaHandle_t          handle,
    cudaLRNDescriptor_t  normDesc,
    cudaDivNormMode_t    mode,
    const void           *alpha,
    const cudaTensorDescriptor_t xDesc,
```

```

const void          *x,
const void          *means,
void               *temp,
void               *temp2,
const void          *beta,
const cudnnTensorDescriptor_t yDesc,
void               *y)

```

The full LCN (Local Contrastive Normalization) computation can be implemented as a two-step process:

```

x_m = x - mean(x);
y = x_m / max(c, sigma(x_m));

```

The  $x - \text{mean}(x)$  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`.



Note: Supported tensor formats are NCHW for 4D and NCDHW for 5D with any non-overlapping non-negative strides. Only 4D and 5D tensors are supported.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor.

### normDesc

*Input.* Handle to a previously initialized LRN parameter descriptor. This descriptor is used for both LRN and `DivisiveNormalization` layers.

### divNormMode

*Input.* `DivisiveNormalization` layer mode of operation. Currently only `CUDNN_DIVNORM_PRECOMPUTED_MEANS` is implemented. Normalization is performed using the means input tensor that is expected to be precomputed by the user.

### alpha, beta

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

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

For more information, refer to [Scaling Parameters](#).

### xDesc, yDesc

*Input.* Tensor descriptor objects for the input and output tensors. Note that `xDesc` is shared between `x`, `means`, `temp`, and `temp2` tensors.

### x

*Input.* Input tensor data pointer in device memory.

### means

*Input.* Input means tensor data pointer in device memory. Note that this tensor can be `NULL` (in that case its values are assumed to be zero during the computation). This tensor also doesn't have to contain `means`, these can be any values, a frequently used variation is a result of convolution with a normalized positive kernel (such as Gaussian).

**temp, temp2**

*Workspace.* Temporary tensors in device memory. These are used for computing intermediate values during the forward pass. These tensors do not have to be preserved as inputs from forward to the backward pass. Both use `xDesc` as their descriptor.

**y**

*Output.* Pointer in device memory to a tensor for the result of the forward `DivisiveNormalization` computation.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the tensor pointers `x`, `y`, `temp`, and `temp2` is `NULL`.
- ▶ Number of input tensor or output tensor dimensions is outside of `[4, 5]` range.
- ▶ A mismatch in dimensions between any two of the input or output tensors.
- ▶ For in-place computation when pointers `x == y`, a mismatch in strides between the input data and output data tensors.
- ▶ Alpha or beta pointer is `NULL`.
- ▶ LRN descriptor parameters are outside of their valid ranges.
- ▶ Any of the tensor strides are negative.

### **CUDNN\_STATUS\_UNSUPPORTED**

The function does not support the provided configuration, for example, any of the input and output tensor strides mismatch (for the same dimension) is a non-supported configuration.

## 3.2.34. `cudnnDropoutForward()`

This function performs forward dropout operation over `x` returning results in `y`. If `dropout` was used as a parameter to `cudnnSetDropoutDescriptor()`, the approximate dropout fraction of `x` values will be replaced by a 0, and the rest will be scaled by  $1/(1-\text{dropout})$ . This function should not be running concurrently with another `cudnnDropoutForward()` function using the same states.

```
cudnnStatus_t cudnnDropoutForward(
    cudnnHandle_t          handle,
    const cudnnDropoutDescriptor_t dropoutDesc,
    const cudnnTensorDescriptor_t xdesc,
    const void             *x,
    const cudnnTensorDescriptor_t ydesc,
    void                  *y,
    void                  *reserveSpace,
```

size\_t

reserveSpaceSizeInBytes)



**Note:**

- ▶ Better performance is obtained for fully packed tensors.
- ▶ This function should not be called during inference.

## Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**dropoutDesc**

*Input.* Previously created dropout descriptor object.

**xDesc**

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

**x**

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

**yDesc**

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

**y**

*Output.* Pointer to data of the tensor described by the `yDesc` descriptor.

**reserveSpace**

*Output.* Pointer to user-allocated GPU memory used by this function. It is expected that the contents of `reserveSpace` does not change between `cudaDropoutForward()` and [cudaDropoutBackward\(\)](#) calls.

**reserveSpaceSizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The call was successful.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The number of elements of input tensor and output tensors differ.
- ▶ The `datatype` of the input tensor and output tensors differs.
- ▶ The strides of the input tensor and output tensors differ and in-place operation is used (meaning, `x` and `y` pointers are equal).
- ▶ The provided `reserveSpaceSizeInBytes` is less than the value returned by [cudaDropoutGetReserveSpaceSize\(\)](#).
- ▶ [cudaSetDropoutDescriptor\(\)](#) has not been called on `dropoutDesc` with the `non-NULL` `states` argument.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

### 3.2.35. cudnnDropoutGetReserveSpaceSize ()

This function is used to query the amount of reserve needed to run dropout with the input dimensions given by `xDesc`. The same reserve space is expected to be passed to [cudnnDropoutForward\(\)](#) and [cudnnDropoutBackward\(\)](#), and its contents is expected to remain unchanged between [cudnnDropoutForward\(\)](#) and [cudnnDropoutBackward\(\)](#) calls.

```

cudnnStatus_t cudnnDropoutGetReserveSpaceSize(
    cudnnTensorDescriptor_t  xDesc,
    size_t                   *sizeInBytes)
    
```

#### Parameters

**xDesc**

*Input.* Handle to a previously initialized tensor descriptor, describing input to a dropout operation.

**sizeInBytes**

*Output.* Amount of GPU memory needed as reserve space to be able to run dropout with an input tensor descriptor specified by `xDesc`.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

### 3.2.36. cudnnDropoutGetStatesSize ()

This function is used to query the amount of space required to store the states of the random number generators used by the [cudnnDropoutForward\(\)](#) function.

```

cudnnStatus_t cudnnDropoutGetStatesSize(
    cudnnHandle_t      handle,
    size_t             *sizeInBytes)
    
```

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**sizeInBytes**

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

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

### 3.2.37. cudnnGetActivationDescriptor()

This function queries a previously initialized generic activation descriptor object.

```

cudnnStatus_t cudnnGetActivationDescriptor(
    const cudnnActivationDescriptor_t  activationDesc,
    cudnnActivationMode_t              *mode,
    cudnnNanPropagation_t              *reluNanOpt,
    double                              *coef)
    
```

#### Parameters

**activationDesc**

*Input.* Handle to a previously created activation descriptor.

**mode**

*Output.* Enumerant to specify the activation mode.

**reluNanOpt**

*Output.* Enumerant to specify the Nan propagation mode.

**coef**

*Output.* Floating point number to specify the clipping threshold when the activation mode is set to CUDNN\_ACTIVATION\_CLIPPED\_RELU or to specify the alpha coefficient when the activation mode is set to CUDNN\_ACTIVATION\_ELU.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was queried successfully.

### 3.2.38. cudnnGetActivationDescriptorSwishBeta()

This function queries the current beta parameter set for SWISH activation.

```

cudnnStatus_t
cudnnGetActivationDescriptorSwishBeta(cudnnActivationDescriptor_t
activationDesc, double* swish_beta)
    
```

#### Parameters

**activationDesc**

*Input.* Handle to a previously created activation descriptor.

**swish\_beta**

*Output.* Pointer to a double value that will receive the currently configured SWISH beta parameter.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The beta parameter was queried successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of activationDesc or swish\_beta were NULL.

### 3.2.39. cudnnGetAlgorithmDescriptor()

This function has been deprecated in cuDNN 8.0.

This function queries a previously initialized generic algorithm descriptor object.

```

cudnnStatus_t cudnnGetAlgorithmDescriptor(
    const cudnnAlgorithmDescriptor_t  algoDesc,
    cudnnAlgorithm_t                  *algorithm)

```

#### 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.

### 3.2.40. cudnnGetAlgorithmPerformance()

This function has been deprecated in cuDNN 8.0.

This function queries a previously initialized generic algorithm performance object.

```

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

```

#### 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.

### 3.2.41. cudnnGetAlgorithmSpaceSize()

This function has been deprecated in cuDNN 8.0.

This function queries for the amount of host memory needed to call [cudnnSaveAlgorithm\(\)](#), much like the *get workspace size* function query for the amount of device memory needed.

```

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

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**algoDesc**

*Input.* A previously created algorithm descriptor.

**algoSpaceSizeInBytes**

*Output.* Amount of host memory needed as a 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`.

### 3.2.42. cudnnGetCallback()

This function queries the internal states of cuDNN error reporting functionality.

```

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

#### 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.

## 3.2.43. 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.

```
size_t cudnnGetCudartVersion()
```

## 3.2.44. cudnnGetDropoutDescriptor()

This function queries the fields of a previously initialized dropout descriptor.

```
cudaStatus_t cudnnGetDropoutDescriptor(
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnHandle_t handle,
    float *dropout,
    void **states,
    unsigned long long *seed)
```

## Parameters

### dropoutDesc

*Input.* Previously initialized dropout descriptor.

### handle

*Input.* Handle to a previously created cuDNN context.

### dropout

*Output.* The probability with which the value from input is set to 0 during the dropout layer.

### states

*Output.* Pointer to user-allocated GPU memory that holds random number generator states.

### seed

*Output.* Seed used to initialize random number generator states.

## Returns

### CUDNN\_STATUS\_SUCCESS

The call was successful.

### CUDNN\_STATUS\_BAD\_PARAM

One or more of the arguments was an invalid pointer.

### 3.2.45. cudnnGetErrorString()

This function converts the cuDNN status code to a `NULL` terminated (ASCII) 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`.

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

#### Parameters

**status**

*Input.* cuDNN enumerant status code.

#### Returns

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

### 3.2.46. cudnnGetFilter4dDescriptor()

This function queries the parameters of the previously initialized `Filter4d` descriptor object.

```
cudaStatus_t cudnnGetFilter4dDescriptor(
    const cudaFilterDescriptor_t filterDesc,
    cudaDataType_t *dataType,
    cudaTensorFormat_t *format,
    int *k,
    int *C,
    int *h,
    int *w)
```

#### Parameters

**filterDesc**

*Input.* Handle to a previously created filter descriptor.

**dataType**

*Output.* Data type.

**format**

*Output.* Type of format.

**k**

*Output.* Number of output feature maps.

**C**

*Output.* Number of input feature maps.

**h**

*Output.* Height of each filter.

**w**

*Output.* Width of each filter.

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was set successfully.

## 3.2.47. cudnnGetFilterNdDescriptor()

This function queries a previously initialized `FilterNd` descriptor object.

```

cudnnStatus_t cudnnGetFilterNdDescriptor(
    const cudnnFilterDescriptor_t wDesc,
    int nbDimsRequested,
    cudnnDataType_t *dataType,
    cudnnTensorFormat_t *format,
    int *nbDims,
    int filterDimA[])
    
```

## 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 dimensions of at least `nbDimsRequested` that will be filled with the filter parameters from the provided filter descriptor.

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was set successfully.

### CUDNN\_STATUS\_BAD\_PARAM

The parameter `nbDimsRequested` is negative.

## 3.2.48. cudnnGetFilterSizeInBytes()

This function returns the size of the filter tensor in memory with respect to the given descriptor. It can be used to know the amount of GPU memory to be allocated to hold that filter tensor.

```

cudnnStatus_t
cudnnGetFilterSizeInBytes(const cudnnFilterDescriptor_t filterDesc, size_t *size) ;
    
```

## Parameters

**filterDesc**

*Input.* handle to a previously initialized filter descriptor.

**size**

*Output.* size in bytes needed to hold the tensor in GPU memory.

## Returns

**CUDNN\_STATUS\_SUCCESS**

filterDesc is valid.

**CUDNN\_STATUS\_BAD\_PARAM**

filterDesc is invalid.

### 3.2.49. cudnnGetLRNDescriptor()

This function retrieves values stored in the previously initialized LRN descriptor object.

```

cudnnStatus_t cudnnGetLRNDescriptor(
    cudnnLRNDescriptor_t    normDesc,
    unsigned                *lrnN,
    double                  *lrnAlpha,
    double                  *lrnBeta,
    double                  *lrnK)
    
```

## Parameters

**normDesc**

*Input.* Handle to a previously created LRN descriptor.

**lrnN, lrnAlpha, lrnBeta, lrnK**

*Output.* Pointers to receive values of parameters stored in the descriptor object. For more information, refer to [cudnnSetLRNDescriptor\(\)](#). Any of these pointers can be NULL (no value is returned for the corresponding parameter).

## Returns

**CUDNN\_STATUS\_SUCCESS**

Function completed successfully.

### 3.2.50. cudnnGetOpTensorDescriptor()

This function returns the configuration of the passed tensor pointwise math descriptor.

```

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

## 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.

## 3.2.51. cudnnGetPooling2dDescriptor()

This function queries a previously created Pooling2D descriptor object.

```

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)

```

## Parameters

### **poolingDesc**

*Input.* Handle to a previously created pooling descriptor.

### **mode**

*Output.* Enumerant to specify the pooling mode.

### **maxpoolingNanOpt**

*Output.* Enumerant to specify the Nan propagation mode.

### **windowHeight**

*Output.* Height of the pooling window.

### **windowWidth**

*Output.* Width of the pooling window.

**verticalPadding**

*Output.* Size of vertical padding.

**horizontalPadding**

*Output.* Size of horizontal padding.

**verticalStride**

*Output.* Pooling vertical stride.

**horizontalStride**

*Output.* Pooling horizontal stride.

Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

### 3.2.52. cudnnGetPooling2dForwardOutputDim()

This function provides the output dimensions of a tensor after Pooling2D has been applied.

```

cudnnStatus_t cudnnGetPooling2dForwardOutputDim(
    const cudnnPoolingDescriptor_t poolingDesc,
    const cudnnTensorDescriptor_t inputDesc,
    int *outN,
    int *outC,
    int *outH,
    int *outW)
    
```

Each dimension *h* and *w* of the output images is computed as follows:

```

outputDim = 1 + (inputDim + 2*padding - windowDim)/poolingStride;
    
```

Parameters

**poolingDesc**

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

**inputDesc**

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

**N**

*Output.* Number of images in the output.

**C**

*Output.* Number of channels in the output.

**H**

*Output.* Height of images in the output.

**W**

*Output.* Width of images in the output.

Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ poolingDesc has not been initialized.
- ▶ poolingDesc or inputDesc has an invalid number of dimensions (2 and 4 respectively are required).

### 3.2.53. cudnnGetPoolingNdDescriptor()

This function queries a previously initialized generic PoolingNd descriptor object.

```

cudnnStatus_t cudnnGetPoolingNdDescriptor(
const cudnnPoolingDescriptor_t poolingDesc,
int nbDimsRequested,
cudnnPoolingMode_t *mode,
cudnnNanPropagation_t *maxpoolingNanOpt,
int *nbDims,
int windowDimA[],
int paddingA[],
int strideA[])
    
```

#### 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**

*Output.* Enumerant to specify the Nan propagation mode.

**nbDims**

*Output.* Actual dimension of the pooling descriptor.

**windowDimA**

*Output.* Array of dimension of at least nbDimsRequested that will be filled with the window parameters from the provided pooling descriptor.

**paddingA**

*Output.* Array of dimension of at least nbDimsRequested that will be filled with the padding parameters from the provided pooling descriptor.

**strideA**

*Output.* Array of dimension at least nbDimsRequested that will be filled with the stride parameters from the provided pooling descriptor.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was queried successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The parameter nbDimsRequested is greater than CUDNN\_DIM\_MAX.

### 3.2.54. cudnnGetPoolingNdForwardOutputDim()

This function provides the output dimensions of a tensor after `PoolingNd` has been applied.

```
cudnnStatus_t cudnnGetPoolingNdForwardOutputDim(
    const cudnnPoolingDescriptor_t poolingDesc,
    const cudnnTensorDescriptor_t inputDesc,
    int nbDims,
    int outDimA[])
```

Each dimension of the  $(nbDims-2)$ -D images of the output tensor is computed as follows:

```
outputDim = 1 + (inputDim + 2*padding - windowDim)/poolingStride;
```

#### Parameters

##### **poolingDesc**

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

##### **inputDesc**

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

##### **nbDims**

*Input.* Number of dimensions in which pooling is to be applied.

##### **outDimA**

*Output.* Array of `nbDims` output dimensions.

#### Returns

##### **CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

##### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `poolingDesc` has not been initialized.
- ▶ The value of `nbDims` is inconsistent with the dimensionality of `poolingDesc` and `inputDesc`.

### 3.2.55. cudnnGetProperty()

This function writes a specific part of the cuDNN library version number into the provided host storage.

```
cudnnStatus_t cudnnGetProperty(
    libraryPropertyType type,
    int *value)
```



## Parameters

### type

*Input.* Enumerant type that instructs the function to report the numerical value of the cuDNN major version, minor version, or the patch level depending on whether `type` is set to `MAJOR_VERSION`, `MINOR_VERSION`, or `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.

## 3.2.56. cudnnGetReduceTensorDescriptor()

This function queries a previously initialized reduce tensor descriptor object.

```
cudnnStatus_t cudnnGetReduceTensorDescriptor(
    const cudnnReduceTensorDescriptor_t reduceTensorDesc,
    cudnnReduceTensorOp_t               *reduceTensorOp,
    cudnnDataType_t                     *reduceTensorCompType,
    cudnnNanPropagation_t               *reduceTensorNanOpt,
    cudnnReduceTensorIndices_t         *reduceTensorIndices,
    cudnnIndicesType_t                 *reduceTensorIndicesType)
```

## Parameters

### reduceTensorDesc

*Input.* Pointer to a previously initialized reduce tensor descriptor object.

### reduceTensorOp

*Output.* Enumerant to specify the reduced tensor operation.

### reduceTensorCompType

*Output.* Enumerant to specify the computation datatype of the reduction.

### reduceTensorNanOpt

*Output.* Enumerant to specify the Nan propagation mode.

### reduceTensorIndices

*Output.* Enumerant to specify the reduced tensor indices.

### reduceTensorIndicesType

*Output.* Enumerant to specify the reduced tensor indices type.

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was queried successfully.

### CUDNN\_STATUS\_BAD\_PARAM

`reduceTensorDesc` is NULL.

### 3.2.57. cudnnGetReductionIndicesSize()

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.

```

cudnnStatus_t cudnnGetReductionIndicesSize(
    cudnnHandle_t          handle,
    const cudnnReduceTensorDescriptor_t reduceDesc,
    const cudnnTensorDescriptor_t aDesc,
    const cudnnTensorDescriptor_t cDesc,
    size_t                 *sizeInBytes)
    
```

#### 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.

### 3.2.58. cudnnGetReductionWorkspaceSize()

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.

```

cudnnStatus_t cudnnGetReductionWorkspaceSize(
    cudnnHandle_t          handle,
    const cudnnReduceTensorDescriptor_t reduceDesc,
    const cudnnTensorDescriptor_t aDesc,
    const cudnnTensorDescriptor_t cDesc,
    size_t                 *sizeInBytes)
    
```

#### 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.

## 3.2.59. cudnnGetStream()

This function retrieves the user CUDA stream programmed in the cuDNN handle. When the user's CUDA stream is not set in the cuDNN handle, this function reports the null-stream.

```

cudnnStatus_t cudnnGetStream(
    cudnnHandle_t  handle,
    cudaStream_t  *streamId)
    
```

### 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.

## 3.2.60. cudnnGetTensor4dDescriptor()

This function queries the parameters of the previously initialized `Tensor4d` descriptor object.

```

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)
    
```

## Parameters

**tensorDesc**

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

**datatype**

*Output.* Data type.

**n**

*Output.* Number of images.

**c**

*Output.* Number of feature maps per image.

**h**

*Output.* Height of each feature map.

**w**

*Output.* Width of each feature map.

**nStride**

*Output.* Stride between two consecutive images.

**cStride**

*Output.* Stride between two consecutive feature maps.

**hStride**

*Output.* Stride between two consecutive rows.

**wStride**

*Output.* Stride between two consecutive columns.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The operation succeeded.

### 3.2.61. cudnnGetTensorNdDescriptor()

This function retrieves values stored in a previously initialized `TensorNd` descriptor object.

```

cudnnStatus_t cudnnGetTensorNdDescriptor(
    const cudnnTensorDescriptor_t  tensorDesc,
    int                             nbDimsRequested,
    cudnnDataType_t                *dataType,
    int                             *nbDims,
    int                             dimA[],
    int                             strideA[])
    
```

## 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 dimensions of at least `nbDimsRequested` that will be filled with the dimensions from the provided tensor descriptor.

**strideA**

*Output.* Array of dimensions of at least `nbDimsRequested` that will be filled with the strides from the provided tensor descriptor.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The results were returned successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

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

## 3.2.62. cudnnGetTensorSizeInBytes()

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.

```
cudnnStatus_t cudnnGetTensorSizeInBytes(
    const cudnnTensorDescriptor_t  tensorDesc,
    size_t                          *size)
```

### Parameters

**tensorDesc**

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

**size**

*Output.* Size in bytes needed to hold the tensor in GPU memory.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The results were returned successfully.

## 3.2.63. cudnnGetTensorTransformDescriptor()

This function returns the values stored in a previously initialized tensor transform descriptor.

```
cudnnStatus_t cudnnGetTensorTransformDescriptor(
```

```

cudnnTensorTransformDescriptor_t transformDesc,
uint32_t nbDimsRequested,
cudnnTensorFormat_t *destFormat,
int32_t padBeforeA[],
int32_t padAfterA[],
uint32_t foldA[],
cudnnFoldingDirection_t *direction);

```

## Parameters

### **transformDesc**

*Input.* A previously initialized tensor transform descriptor.

### **nbDimsRequested**

*Input.* The number of dimensions to consider. For more information, refer to [Tensor Descriptor](#).

### **destFormat**

*Output.* The transform format that will be returned.

### **padBeforeA[]**

*Output.* An array filled with the amount of padding to add before each dimension. The dimension of this `padBeforeA[]` parameter is equal to `nbDimsRequested`.

### **padAfterA[]**

*Output.* An array filled with the amount of padding to add after each dimension. The dimension of this `padBeforeA[]` parameter is equal to `nbDimsRequested`.

### **foldA[]**

*Output.* An array that was filled with the folding parameters for each spatial dimension. The dimension of this `foldA[]` array is `nbDimsRequested-2`.

### **direction**

*Output.* The setting that selects folding or unfolding. For more information, refer to [cudnnFoldingDirection\\_t](#).

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The results were obtained successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

If `transformDesc` is NULL or if `nbDimsRequested` is less than 3 or greater than `CUDNN_DIM_MAX`.

## 3.2.64. `cudnnGetVersion()`

This function returns the version number of the cuDNN library. It returns the `CUDNN_VERSION` defined 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 defined `CUDNN_VERSION` can be used to have the same application linked against different cuDNN versions using conditional compilation statements.

```

size_t cudnnGetVersion()

```


### 3.2.65. cudnnInitTransformDest()

This function initializes and returns a destination tensor descriptor `destDesc` for tensor transform operations. The initialization is done with the desired parameters described in the transform descriptor `cudnnTensorDescriptor_t`.

```

cudnnStatus_t cudnnInitTransformDest(
    const cudnnTensorTransformDescriptor_t transformDesc,
    const cudnnTensorDescriptor_t srcDesc,
    cudnnTensorDescriptor_t destDesc,
    size_t *destSizeInBytes);

```

 Note: The returned tensor descriptor will be packed.

#### Parameters

**transformDesc**

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

**srcDesc**

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

**destDesc**

*Output.* Handle of the tensor descriptor that will be initialized and returned.

**destSizeInBytes**

*Output.* A pointer to hold the size, in bytes, of the new tensor.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The tensor descriptor was initialized successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

If either `srcDesc` or `destDesc` is NULL, or if the tensor descriptor's `nbDims` is incorrect. For more information, refer to [Tensor Descriptor](#).

**CUDNN\_STATUS\_NOT\_SUPPORTED**

If the provided configuration is not 4D.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

Function failed to launch on the GPU.

### 3.2.66. cudnnLRNCrossChannelForward()

This function performs the forward LRN layer computation.

```

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)



Note: Supported formats are: `positive-strided`, NCHW and NHWC 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. NCHW layout is preferred for performance.

## 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:

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

For more information, refer to [Scaling Parameters](#).

### **xDesc, yDesc**

*Input.* Tensor descriptor objects for the input and output tensors.

### **x**

*Input.* Input tensor data pointer in device memory.

### **y**

*Output.* Output tensor data pointer in device memory.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the tensor pointers  $x$ ,  $y$  is `NULL`.
- ▶ Number of input tensor dimensions is 2 or less.
- ▶ LRN descriptor parameters are outside of their valid ranges.
- ▶ One of the tensor parameters is 5D but is not in NCDHW DHW-packed format.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration. Refer to the following 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.

### 3.2.67. cudnnNormalizationForwardInference ()

This function performs the forward normalization layer computation for the inference phase. Per-channel normalization layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```

cudnnStatus_t
cudnnNormalizationForwardInference(cudnnHandle_t handle,
                                   cudnnNormMode_t mode,
                                   cudnnNormOps_t normOps,
                                   cudnnNormAlgo_t algo,
                                   const void *alpha,
                                   const void *beta,
                                   const cudnnTensorDescriptor_t xDesc,
                                   const void *x,
                                   const cudnnTensorDescriptor_t normScaleBiasDesc,
                                   const void *normScale,
                                   const void *normBias,
                                   const cudnnTensorDescriptor_t normMeanVarDesc,
                                   const void *estimatedMean,
                                   const void *estimatedVariance,
                                   const cudnnTensorDescriptor_t zDesc,
                                   const void *z,
                                   cudnnActivationDescriptor_t activationDesc,
                                   const cudnnTensorDescriptor_t yDesc,
                                   void *y,
                                   double epsilon,
                                   int groupCnt);

```

Only 4D and 5D tensors are supported.

The input transformation performed by this function is defined as:

```

y = beta*y + alpha * [normBias + (normScale * (x-estimatedMean)/sqrt(epsilon +
estimatedVariance))]

```

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

For the training phase, refer to [cudnnNormalizationForwardTraining\(\)](#).

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

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

**mode**

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [cudnnNormMode\\_t](#).

**normOps**

*Input.* Mode of post-operative. Currently, `CUDNN_NORM_OPS_NORM_ACTIVATION` and `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` are not supported.

**algo**

*Input.* Algorithm to be performed. For more information, refer to [cudnnNormAlgo\\_t](#).

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**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**

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

**zDesc, \*z**

*Input.* Tensor descriptors and pointers in device memory for residual addition to the result of the normalization operation, prior to the activation. `zDesc` and `*z` are optional and are only used when `normOps` is `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION`, otherwise users may pass `NULL`. When in use, `z` should have exactly the same dimension as `x` and the final output `y`. For more information, refer to [cudnnTensorDescriptor\\_t](#).

Since `normOps` is only supported for `CUDNN_NORM_OPS_NORM`, we can set these to `NULL` for now.

**normScaleBiasDesc, normScale, normBias**

*Inputs.* Tensor descriptors and pointers in device memory for the normalization scale and bias parameters (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as beta and scale as gamma).

**normMeanVarDesc, estimatedMean, estimatedVariance**

*Inputs.* Mean and variance tensors and their tensor descriptors. The `estimatedMean` and `estimatedVariance` inputs, accumulated during the training phase from the [cudnnNormalizationForwardTraining\(\)](#) call, should be passed as inputs here.

**activationDesc**

*Input.* Descriptor for the activation operation. When the `normOps` input is set to either `CUDNN_NORM_OPS_NORM_ACTIVATION` or `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` then this activation is used, otherwise the user may pass `NULL`. Since `normOps` is only supported for `CUDNN_NORM_OPS_NORM`, we can set these to `NULL` for now.

**epsilon**

*Input.* Epsilon value used in the normalization formula. Its value should be equal to or greater than zero.

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

## Returns

### CUDNN\_STATUS\_SUCCESS

The computation was performed successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

A compute or data type other than what is supported was chosen, or an unknown algorithm type was chosen.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ One of the pointers `alpha`, `beta`, `x`, `y`, `normScale`, `normBias`, `estimatedMean`, and `estimatedInvVariance` is `NULL`.
- ▶ The number of `xDesc` or `yDesc` tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- ▶ `normScaleBiasDesc` and `normMeanVarDesc` dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for per-channel, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- ▶ `epsilon` value is less than zero.
- ▶ Dimensions or data types mismatch for `xDesc` and `yDesc`.

### 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.

## 3.2.68. cudnnOpsInferVersionCheck ()

This function is the first of a series of corresponding functions that check for consistent library versions among DLL files for different modules.

```
cudnnStatus_t cudnnOpsInferVersionCheck(void)
```

## Returns

### CUDNN\_STATUS\_SUCCESS

The version of this DLL file is consistent with cuDNN DLLs on which it depends.

### CUDNN\_STATUS\_VERSION\_MISMATCH

The version of this DLL file does not match that of a cuDNN DLLs on which it depends.

## 3.2.69. cudnnOpTensor ()

This function implements the equation  $C = op(\alpha_1[0] * A, \alpha_2[0] * B) + \beta[0] * C$ , given the tensors `A`, `B`, and `C` and the scaling factors `alpha1`, `alpha2`, and `beta`. The `op` to use is indicated by the descriptor `cudnnOpTensorDescriptor_t`, meaning,

the type of `opTensorDesc`. Currently-supported ops are listed by the [`cudaOpTensorOp\_t`](#) enum.

```

cudaStatus_t cudaOpTensor(
    cudaHandle_t          handle,
    const cudaOpTensorDescriptor_t opTensorDesc,
    const void            *alpha1,
    const cudaTensorDescriptor_t aDesc,
    const void            *A,
    const void            *alpha2,
    const cudaTensorDescriptor_t bDesc,
    const void            *B,
    const void            *beta,
    const cudaTensorDescriptor_t cDesc,
    void                  *C)
    
```

The following restrictions on the input and destination tensors apply:

- ▶ Each dimension of the input tensor `A` must match the corresponding dimension of the destination tensor `C`, and each dimension of the input tensor `B` must match the corresponding dimension of the destination tensor `C` or must be equal to 1. In the latter case, the same value from the input tensor `B` for those dimensions will be used to blend into the `C` tensor.
- ▶ The data types of the input tensors `A` and `B`, and the destination tensor `C`, must satisfy [Table 13](#).

Table 13. Supported Datatypes for `cudaOpTensor()`

<b>opTensorCompType in opTensorDesc</b>	<b>A</b>	<b>B</b>	<b>c (destination)</b>
FLOAT	FLOAT	FLOAT	FLOAT
FLOAT	INT8	INT8	FLOAT
FLOAT	HALF	HALF	FLOAT
FLOAT	BFLOAT16	BFLOAT16	FLOAT
DOUBLE	DOUBLE	DOUBLE	DOUBLE
FLOAT	FLOAT	FLOAT	HALF
FLOAT	HALF	HALF	HALF
FLOAT	INT8	INT8	INT8
FLOAT	FLOAT	FLOAT	INT8
FLOAT	FLOAT	FLOAT	BFLOAT16
FLOAT	BFLOAT16	BFLOAT16	BFLOAT16

**Note:** `CUDNN_TENSOR_NCHW_VECT_C` is not supported as input tensor format. All tensors up to dimension five (5) are supported. This routine does not support tensor formats beyond these dimensions.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### opTensorDesc

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

### alpha1, alpha2, beta

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

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

For more information, refer to [Scaling Parameters](#).

### aDesc, bDesc, cDesc

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

### A, B

*Input.* Pointer to data of the tensors described by the aDesc and bDesc descriptors, respectively.

### C

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The function executed successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration. Refer to the following examples of non-supported configurations:

- ▶ The dimensions of the bias tensor and the output tensor dimensions are above 5.
- ▶ opTensorCompType is not set as stated above.

### CUDNN\_STATUS\_BAD\_PARAM

The data type of the destination tensor c is unrecognized, or the restrictions on the input and destination tensors, stated above, are not met.

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

## 3.2.70. cudnnPoolingForward()

This function computes pooling of input values (meaning, the maximum or average of several adjacent values) to produce an output with smaller height and/or width.

```
cudaStatus_t cudnnPoolingForward(
    cudaHandle_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)
```

All tensor formats are supported, best performance is expected when using HW-packed tensors. Only 2 and 3 spatial dimensions are allowed. Vectorized tensors are only supported if they have 2 spatial dimensions.

The dimensions of the output tensor `yDesc` can be smaller or bigger than the dimensions advised by the routine [cudnnGetPooling2dForwardOutputDim\(\)](#) or [cudnnGetPoolingNdForwardOutputDim\(\)](#).

For average pooling, the compute type is `float` even for integer input and output data type. Output round is nearest-even and clamp to the most negative or most positive value of type if out of range.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **poolingDesc**

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

### **alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

### **xDesc**

*Input.* Handle to the previously initialized input tensor descriptor. Must be of type `float`, `double`, `half`, `int8`, `int8x4`, `int8x32`, or `bf16`. For more information, refer to [cudnnDataType\\_t](#).

### **x**

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

### **yDesc**

*Input.* Handle to the previously initialized output tensor descriptor. Must be of type `float`, `double`, `half`, `int8`, `int8x4`, `int8x32`, or `bf16`. For more information, refer to [cudnnDataType\\_t](#).

### **y**

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

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions `n`, `c` of the input tensor and output tensors differ.

- ▶ The `datatype` of the input tensor and output tensors differs.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

### 3.2.71. `cudaQueryRuntimeError()`

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.

```

cudaStatus_t cudaQueryRuntimeError(
    cudaHandle_t          handle,
    cudaStatus_t          *rstatus,
    cudaErrQueryMode_t    mode,
    cudaRuntimeTag_t      *tag)
    
```

When the `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` mode is selected in `cudaBatchNormalizationForwardTraining()` or `cudaBatchNormalizationBackward()`, the algorithm may encounter numerical overflows where `CUDNN_BATCHNORM_SPATIAL` performs just fine albeit at a slower speed. The user can invoke `cudaQueryRuntimeError()` to make sure numerical overflows did not occur during the kernel execution. Those issues are reported by the kernel that performs computations.

`cudaQueryRuntimeError()` can be used in polling and blocking software control flows. There are two polling modes (`CUDNN_ERRQUERY_RAWCODE` and `CUDNN_ERRQUERY_NONBLOCKING`) and one blocking mode `CUDNN_ERRQUERY_BLOCKING`.


`CUDNN_ERRQUERY_RAWCODE` reads the error storage location regardless of the kernel completion status. The kernel might not even start 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 are completed. The `cudaQueryRuntimeError()` function will return immediately and report `CUDNN_STATUS_RUNTIME_IN_PROGRESS` in `rstatus` if some tasks in the user stream are pending. Otherwise, the function will copy the remote kernel error code to `rstatus`.

In the blocking mode (`CUDNN_ERRQUERY_BLOCKING`), the function waits for all tasks to drain in the user stream before reporting the remote kernel error code. The blocking flavor can be further adjusted by calling `cudaSetDeviceFlags` with the `cudaDeviceScheduleSpin`, `cudaDeviceScheduleYield`, or `cudaDeviceScheduleBlockingSync` flag.

`CUDNN_ERRQUERY_NONBLOCKING` and `CUDNN_ERRQUERY_BLOCKING` modes should not be used when the user stream is changed in the cuDNN handle, meaning, `cudaSetStream()` is invoked between functions that report runtime kernel errors and the `cudaQueryRuntimeError()` 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 `cudaQueryRuntimeError()`.

 **Note:** The `cudaQueryRuntimeError()` function should be used in conjunction with [cudaBatchNormalizationForwardTraining\(\)](#) and [cudaBatchNormalizationBackward\(\)](#) when the `cudaBatchNormMode_t` argument is `CUDNN_BATCHNORM_SPATIAL_PERSISTENT`.

## Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**rstatus**

*Output.* Pointer to the user's error code storage.

**mode**

*Input.* Remote error query mode.

**tag**

*Input/Output.* Currently, this argument should be `NULL`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

No errors detected (`rstatus` holds a valid value).

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid input argument.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

A stream blocking synchronization or a non-blocking stream query failed.

**CUDNN\_STATUS\_MAPPING\_ERROR**

The device cannot access zero-copy memory to report kernel errors.

## 3.2.72. cudaReduceTensor()

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.

```

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



Each dimension of the output tensor `c` must match the corresponding dimension of the input tensor `A` or must be equal to 1. The dimensions equal to 1 indicate the dimensions of `A` to be reduced.

The implementation will generate indices for the min and max ops only, as indicated by the `cudnnReduceTensorIndices_t` enum of the `reduceTensorDesc`. Requesting indices for the other reduction ops results in an error. The data type of the indices is indicated by the `cudnnIndicesType_t` enum; currently only the 32-bit (unsigned int) type is supported.

The indices returned by the implementation are not absolute indices but relative to the dimensions being reduced. The indices are also flattened, meaning, not coordinate tuples.

The data types of the tensors `A` and `C` must match if of type double. In this case, `alpha` and `beta` and the computation enum of `reduceTensorDesc` are all assumed to be of type double.

The `HALF` and `INT8` data types may be mixed with the `FLOAT` data types. In these cases, the computation enum of `reduceTensorDesc` is required to be of type `FLOAT`.



Note: Up to dimension 8, all tensor formats are supported. Beyond those dimensions, this routine is not supported.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **reduceTensorDesc**

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

### **indices**

*Output.* Handle to a previously allocated space for writing indices.

### **indicesSizeInBytes**

*Input.* Size of the above previously allocated space.

### **workspace**

*Input.* Handle to a previously allocated space for the reduction implementation.

### **workspaceSizeInBytes**

*Input.* Size of the above previously allocated space.

### **alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

### **aDesc, cDesc**

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

### **A**

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

C

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The function executed successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

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

- ▶ The dimensions of the input tensor and the output tensor are above 8.
- ▶ `reduceTensorCompType` is not set as stated above.

### CUDNN\_STATUS\_BAD\_PARAM

The corresponding dimensions of the input and output tensors all match, or the conditions in the above paragraphs are unmet.

### CUDNN\_INVALID\_VALUE

The allocations for the indices or workspace are insufficient.

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

## 3.2.73. `cudnnRestoreAlgorithm()`

This function has been deprecated in cuDNN 8.0.

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.

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

## 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 a 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.

## 3.2.74. cudnnRestoreDropoutDescriptor()

This function restores a dropout descriptor to a previously saved-off state.

```

cudnnStatus_t cudnnRestoreDropoutDescriptor(
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnHandle_t             handle,
    float                     dropout,
    void                      *states,
    size_t                    stateSizeInBytes,
    unsigned long long        seed)
    
```

## 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 calls to [cudnnSetDropoutDescriptor\(\)](#) that initialized states buffer. Using a different seed from this has no effect. A change of seed, and subsequent update to random number generator states can be achieved by calling [cudnnSetDropoutDescriptor\(\)](#).

## Returns

### CUDNN\_STATUS\_SUCCESS

The call was successful.

**CUDNN\_STATUS\_INVALID\_VALUE**

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

### 3.2.75. `cudaSaveAlgorithm()`

This function has been deprecated in cuDNN 8.0.

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.

```

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

```

#### 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 a 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.

### 3.2.76. `cudaScaleTensor()`

This function scales all the elements of a tensor by a given factor.

```

cudaStatus_t cudaScaleTensor(
    cudaHandle_t      handle,
    const cudaTensorDescriptor_t yDesc,
    void*             *y,
    const void        *alpha)

```

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**yDesc**

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

**y**

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

**alpha**

*Input.* Pointer in the host memory to a single value that all elements of the tensor will be scaled with. For more information, refer to [Scaling Parameters](#).

## 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.

## 3.2.77. `cudaSetActivationDescriptor()`

This function initializes a previously created generic activation descriptor object.

```

cudaStatus_t cudaSetActivationDescriptor(
    cudaActivationDescriptor_t      activationDesc,
    cudaActivationMode_t           mode,
    cudaNanPropagation_t          reluNanOpt,
    double                          coef)
    
```

## Parameters

**activationDesc**

*Input/Output.* Handle to a previously created activation descriptor.

**mode**

*Input.* Enumerant to specify the activation mode.

**reluNanOpt**

*Input.* Enumerant to specify the `Nan` propagation mode.

**coef**

*Input.* Floating point number. When the activation mode (refer to [cudaActivationMode\\_t](#)) is set to `CUDNN_ACTIVATION_CLIPPED_RELU`, this input specifies the clipping threshold; and when the activation mode is set to `CUDNN_ACTIVATION_RELU`, this input specifies the upper bound.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

mode `OR` `reluNanOpt` has an invalid enumerant value.

### 3.2.78. `cudnnSetActivationDescriptorSwishBeta()`

This function sets the beta parameter of the SWISH activation function to `swish_beta`.

```
cudnnStatus_t cudnnSetActivationDescriptorSwishBeta(cudnnActivationDescriptor_t
activationDesc, double swish_beta)
```

#### Parameters

**activationDesc**

*Input/Output.* Handle to a previously created activation descriptor.

**swish\_beta**

*Input.* The value to set the SWISH activations' beta parameter to.

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The value was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

The activation descriptor is a `NULL` pointer.

### 3.2.79. `cudnnSetAlgorithmDescriptor()`

This function has been deprecated in cuDNN 8.0.

This function initializes a previously created generic algorithm descriptor object.

```
cudnnStatus_t cudnnSetAlgorithmDescriptor(
cudnnAlgorithmDescriptor_t algorithmDesc,
cudnnAlgorithm_t algorithm)
```

#### 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.

### 3.2.80. `cudnnSetAlgorithmPerformance()`

This function has been deprecated in cuDNN 8.0.

This function initializes a previously created generic algorithm performance object.

```
cudnnStatus_t cudnnSetAlgorithmPerformance(
cudnnAlgorithmPerformance_t algoPerf,
```

```

    cudnnAlgorithmDescriptor_t    algoDesc,
    cudnnStatus_t                status,
    float                          time,
    size_t                         memory)

```

## Parameters

### **algoPerf**

*Input/Output.* Handle to a previously created algorithm performance object.

### **algoDesc**

*Input.* The algorithm descriptor which the performance results describe.

### **status**

*Input.* The cuDNN status returned from running the `algoDesc` algorithm.

### **time**

*Input.* The GPU time spent running the `algoDesc` algorithm.

### **memory**

*Input.* The GPU memory needed to run the `algoDesc` algorithm.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

`mode` or `reluNanOpt` has an invalid enumerate value.

## 3.2.81. cudnnSetCallback()

This function sets the internal states of cuDNN error reporting functionality.

```

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

```

## Parameters

### **mask**

*Input.* An unsigned integer. The four least significant bits (LSBs) of this unsigned integer are used for switching on and off the different levels of error reporting messages. This applies for both the default callbacks, and for the customized callbacks. The bit position is in correspondence with the enum of `cudnnSeverity_t`. The user may utilize the predefined macros `CUDNN_SEV_ERROR_EN`, `CUDNN_SEV_WARNING_EN`, and `CUDNN_SEV_INFO_EN` to form the bit mask. When a bit is set to 1, the corresponding message channel is enabled.

For example, when bit 3 is set to 1, the API logging is enabled. Currently, only the log output of level `CUDNN_SEV_INFO` is functional; the others are not yet implemented. When used for turning on and off the logging with the default callback, the user may pass `NULL` to `udata` and `fptr`. In addition, the environment variable `CUDNN_LOGDEST_DBG` must be set. For more information, refer to [Deprecation Policy](#).

- ▶ 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 cudnnDebug\_t is defined in the header file. It provides the metadata, such as time, time since start, stream ID, process and thread ID, that the user may choose to print or store in their customized callback.
- ▶ The variable msg is the logging message generated by cuDNN. Each line of this message is terminated by \0, and the end of the message is terminated by \0\0. Users may select what is necessary to show in the log, and may reformat the string.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

### 3.2.82. cudnnSetDropoutDescriptor ()

This function initializes a previously created dropout descriptor object. If the states argument is equal to NULL, then the random number generator states won't be initialized, and only the dropout value will be set. The user is expected not to change the memory pointed at by states for the duration of the computation.

```
cudnnStatus_t cudnnSetDropoutDescriptor (
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnHandle_t handle,
    float dropout,
    void *states,
    size_t stateSizeInBytes,
    unsigned long long seed)
```

When the states argument is not NULL, a cuRAND initialization kernel is invoked by cudnnSetDropoutDescriptor(). This kernel requires a substantial amount of



GPU memory for the stack. Memory is released when the kernel finishes. The `CUDNN_STATUS_ALLOC_FAILED` status is returned when no sufficient free memory is available for the GPU stack.

## Parameters

### **dropoutDesc**

*Input/Output.* Previously created dropout descriptor object.

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **dropout**

*Input.* The probability with which the value from input is set to zero during the dropout layer.

### **states**

*Output.* Pointer to user-allocated GPU memory that will hold random number generator states.

### **stateSizeInBytes**

*Input.* Specifies the size in bytes of the provided memory for the states.

### **seed**

*Input.* Seed used to initialize random number generator states.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The call was successful.

### **CUDNN\_STATUS\_INVALID\_VALUE**

The `sizeInBytes` argument is less than the value returned by [`cudaDropoutGetStatesSize\(\)`](#).

### **CUDNN\_STATUS\_ALLOC\_FAILED**

The function failed to temporarily extend the GPU stack.

### **CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

### **CUDNN\_STATUS\_INTERNAL\_ERROR**

Internally used CUDA functions returned an error status.

## 3.2.83. `cudaSetFilter4dDescriptor()`

This function initializes a previously created filter descriptor object into a 4D filter. The layout of the filters must be contiguous in memory.

```
cudaStatus_t cudaSetFilter4dDescriptor(
    cudaFilterDescriptor_t  filterDesc,
    cudaDataType_t         dataType,
    cudaTensorFormat_t     format,
    int                    k,
    int                    c,
    int                    h,
    int                    w)
```

Tensor format `CUDNN_TENSOR_NHWC` has limited support in [`cudaConvolutionForward\(\)`](#), [`cudaConvolutionBackwardData\(\)`](#), and [`cudaConvolutionBackwardFilter\(\)`](#).

## Parameters

### **filterDesc**

*Input/Output.* Handle to a previously created filter descriptor.

### **dataType**

*Input.* Data type.

### **format**

*Input.* Type of the filter layout format. If this input is set to `CUDNN_TENSOR_NCHW`, which is one of the enumerant values allowed by [`cudaTensorFormat\_t`](#) descriptor, then the layout of the filter is in the form of `KCRS`, where:

- ▶ `K` represents the number of output feature maps
- ▶ `C` is the number of input feature maps
- ▶ `R` is the number of rows per filter
- ▶ `S` is the number of columns per filter

If this input is set to `CUDNN_TENSOR_NHWC`, then the layout of the filter is in the form of `KRSC`. For more information, refer to [`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.

## 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.

## 3.2.84. `cudaSetFilterNdDescriptor()`

This function initializes a previously created filter descriptor object. The layout of the filters must be contiguous in memory.

```
cudaStatus_t cudaSetFilterNdDescriptor(
    cudaFilterDescriptor_t filterDesc,
    cudaDataType_t         dataType,
    cudaTensorFormat_t     format,
    int                    nbDims,
```

```
const int filterDimA[])
```

The tensor format CUDNN\_TENSOR\_NHWC has limited support in [cudnnConvolutionForward\(\)](#), [cudnnConvolutionBackwardData\(\)](#), and [cudnnConvolutionBackwardFilter\(\)](#).

## Parameters

### filterDesc

*Input/Output.* Handle to a previously created filter descriptor.

### datatype

*Input.* Data type.

### format

*Input.*Type of the filter layout format. If this input is set to CUDNN\_TENSOR\_NCHW, which is one of the enumerant values allowed by [cudnnTensorFormat\\_t](#) descriptor, then the layout of the filter is as follows:

- ▶ For N=4, a 4D filter descriptor, the filter layout is in the form of KCRS:
  - ▶ K represents the number of output feature maps
  - ▶ C is the number of input feature maps
  - ▶ R is the number of rows per filter
  - ▶ S is the number of columns per filter
- ▶ For N=3, a 3D filter descriptor, the number S (number of columns per filter) is omitted.
- ▶ For N=5 and greater, the layout of the higher dimensions immediately follows RS.

On the other hand, if this input is set to CUDNN\_TENSOR\_NHWC, then the layout of the filter is as follows:

- ▶ For N=4, a 4D filter descriptor, the filter layout is in the form of KRSC.
- ▶ For N=3, a 3D filter descriptor, the number S (number of columns per filter) is omitted and the layout of C immediately follows R.
- ▶ For N=5 and greater, the layout of the higher dimensions are inserted between S and C. For more information, refer to [cudnnTensorFormat\\_t](#).

### nbDims

*Input.* Dimension of the filter.

### filterDimA

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the elements of the array `filterDimA` is negative or `dataType` or `format` has an invalid enumerant value.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The parameter `nbDims` exceeds `CUDNN_DIM_MAX`.

### 3.2.85. `cudaSetLRNDescriptor()`

This function initializes a previously created LRN descriptor object.

```
cudaStatus_t cudaSetLRNDescriptor(
    cudaLRNDescriptor_t    normDesc,
    unsigned               lrnN,
    double                 lrnAlpha,
    double                 lrnBeta,
    double                 lrnK)
```



**Note:**

- ▶ 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. The 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 the `DivisiveNormalization` layer, the window has the same extent 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 `(window width) ^ #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 the normalization formula. By default, this value is set to `2.0`.

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was set successfully.

### CUDNN\_STATUS\_BAD\_PARAM

One of the input parameters was out of valid range as described above.

## 3.2.86. cudnnSetOpTensorDescriptor()

This function initializes a tensor pointwise math descriptor.

```

cudnnStatus_t cudnnSetOpTensorDescriptor(
    cudnnOpTensorDescriptor_t  opTensorDesc,
    cudnnOpTensorOp_t          opTensorOp,
    cudnnDataType_t            opTensorCompType,
    cudnnNanPropagation_t      opTensorNanOpt)
    
```

## 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 the input parameters passed is invalid.

## 3.2.87. cudnnSetPooling2dDescriptor()

This function initializes a previously created generic pooling descriptor object into a 2D description.

```

cudnnStatus_t cudnnSetPooling2dDescriptor(
    cudnnPoolingDescriptor_t  poolingDesc,
    cudnnPoolingMode_t        mode,
    cudnnNanPropagation_t     maxpoolingNanOpt,
    int                        windowHeight,
    int                        windowWidth,
    int                        verticalPadding,
    int                        horizontalPadding,
    int                        verticalStride,
    int                        horizontalStride)
    
```

## Parameters

**poolingDesc**

*Input/Output.* Handle to a previously created pooling descriptor.

**mode**

*Input.* Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Input.* Enumerant to specify the Nan propagation mode.

**windowHeight**

*Input.* Height of the pooling window.

**windowWidth**

*Input.* Width of the pooling window.

**verticalPadding**

*Input.* Size of vertical padding.

**horizontalPadding**

*Input.* Size of horizontal padding.

**verticalStride**

*Input.* Pooling vertical stride.

**horizontalStride**

*Input.* Pooling horizontal stride.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the parameters `windowHeight`, `windowWidth`, `verticalStride`, `horizontalStride` is negative or `mode` or `maxpoolingNanOpt` has an invalid enumerate value.

### 3.2.88. cudnnSetPoolingNdDescriptor()

This function initializes a previously created generic pooling descriptor object.

```

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[])
    
```

## Parameters

**poolingDesc**

*Input/Output.* Handle to a previously created pooling descriptor.

**mode**

*Input.* Enumerant to specify the pooling mode.

**maxpoolingNanOpt**

*Input.* Enumerant to specify the Nan propagation mode.

**nbDims**

*Input.* Dimension of the pooling operation. Must be greater than zero.

**windowDimA**

*Input.* Array of dimension `nbDims` containing the window size for each dimension. The value of array elements must be greater than zero.

**paddingA**

*Input.* Array of dimension `nbDims` containing the padding size for each dimension. Negative padding is allowed.

**strideA**

*Input.* Array of dimension `nbDims` containing the striding size for each dimension. The value of array elements must be greater than zero (meaning, negative striding size is not allowed).

### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was initialized successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

If `(nbDims > CUDNN_DIM_MAX-2)`.

**CUDNN\_STATUS\_BAD\_PARAM**

Either `nbDims`, or at least one of the elements of the arrays `windowDimA` or `strideA` is negative, or `mode` or `maxpoolingNanOpt` has an invalid enumerate value.

## 3.2.89. cudnnSetReduceTensorDescriptor()

This function initializes a previously created reduce tensor descriptor object.

```

cudnnStatus_t cudnnSetReduceTensorDescriptor(
    cudnnReduceTensorDescriptor_t    reduceTensorDesc,
    cudnnReduceTensorOp_t           reduceTensorOp,
    cudnnDataType_t                 reduceTensorCompType,
    cudnnNanPropagation_t           reduceTensorNanOpt,
    cudnnReduceTensorIndices_t      reduceTensorIndices,
    cudnnIndicesType_t              reduceTensorIndicesType)
    
```

### 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 reduced 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).

### 3.2.90. cudnnSetSpatialTransformerNdDescriptor()

This function initializes a previously created generic spatial transformer descriptor object.

```

cudnnStatus_t cudnnSetSpatialTransformerNdDescriptor(
    cudnnSpatialTransformerDescriptor_t    stDesc,
    cudnnSamplerType_t                    samplerType,
    cudnnDataType_t                       dataType,
    const int                              nbDims,
    const int                              dimA[])
    
```

**Parameters**

**stDesc**

*Input/Output.* Previously created spatial transformer descriptor object.

**samplerType**

*Input.* Enumerant to specify the sampler type.

**dataType**

*Input.* Data type.

**nbDims**

*Input.* Dimension of the transformed tensor.

**dimA**

*Input.* Array of dimension nbDims containing the size of the transformed tensor for every dimension.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The call was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:



- ▶ Either `stDesc` or `dimA` is `NULL`.
- ▶ Either `dataType` or `samplerType` has an invalid enumerant value.

### 3.2.91. `cudaSetStream()`

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.

```
cudaStatus_t cudaSetStream(
    cudaHandle_t   handle,
    cudaStream_t   streamId)
```

With CUDA 11.x or later, internal streams have the same priority as the stream set by the last call to this function. In CUDA graph capture mode, CUDA 11.8 or later is required in order for the stream priorities to match.

#### 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.

### 3.2.92. `cudaSetTensor()`

This function sets all the elements of a tensor to a given value.

```
cudaStatus_t cudaSetTensor(
    cudaHandle_t   handle,
    const cudaTensorDescriptor_t yDesc,
    void           *y,
    const void     *valuePtr)
```

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**yDesc**

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

**y**

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

**valuePtr**

*Input.* Pointer in host memory to a single value. All elements of the `y` tensor will be set to `value[0]`. The data type of the element in `value[0]` has to match the data type of tensor `y`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

One of the provided pointers is `NIL`.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 3.2.93. cudnnSetTensor4dDescriptor()

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.

```

cudnnStatus_t cudnnSetTensor4dDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                      n,
    int                      c,
    int                      h,
    int                      w)
    
```

The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type `datatype`.

## Parameters

**tensorDesc**

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

**format**

*Input.* Type of format.

**datatype**

*Input.* Data type.

**n**

*Input.* Number of images.

**c**

*Input.* Number of feature maps per image.

**h**  
*Input.* Height of each feature map.

**w**  
*Input.* Width of each feature map.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the parameters `n`, `c`, `h`, `w` was negative or `format` has an invalid enumerant value or `dataType` has an invalid enumerant value.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The total size of the tensor descriptor exceeds the maximum limit of 2 Giga-elements.

### 3.2.94. `cudaSetTensor4dDescriptorEx()`

This function initializes a previously created generic tensor descriptor object into a 4D tensor, similarly to `cudaSetTensor4dDescriptor()` 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.

```

cudaStatus_t cudaSetTensor4dDescriptorEx(
    cudaTensorDescriptor_t  tensorDesc,
    cudaDataType_t         dataType,
    int                    n,
    int                    c,
    int                    h,
    int                    w,
    int                    nStride,
    int                    cStride,
    int                    hStride,
    int                    wStride)
    
```

At present, some cuDNN routines have limited support for strides. Those routines will return `CUDNN_STATUS_NOT_SUPPORTED` if a 4D tensor object with an unsupported stride is used. [cudaTransformTensor\(\)](#) can be used to convert the data to a supported layout.

The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type `dataType`.

### Parameters

**tensorDesc**

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

**dataType**

*Input.* Data type.

**n**

*Input.* Number of images.

**c**

*Input.* Number of feature maps per image.

**h**

*Input.* Height of each feature map.

**w**

*Input.* Width of each feature map.

**nStride**

*Input.* Stride between two consecutive images.

**cStride**

*Input.* Stride between two consecutive feature maps.

**hStride**

*Input.* Stride between two consecutive rows.

**wStride**

*Input.* Stride between two consecutive columns.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the parameters `n`, `c`, `h`, `w` or `nStride`, `cStride`, `hStride`, `wStride` is negative or `dataType` has an invalid enumerant value.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The total size of the tensor descriptor exceeds the maximum limit of 2 Giga-elements.

## 3.2.95. cudnnSetTensorNdDescriptor()

This function initializes a previously created generic tensor descriptor object.

```

cudnnStatus_t cudnnSetTensorNdDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnDataType_t         dataType,
    int                      nbDims,
    const int                dimA[],
    const int                strideA[])
    
```

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 `cuda.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.

 Note: Do not use 2 dimensions. Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. For more information, refer to [cudnnGetRNNLinLayerBiasParams\(\)](#).

**dimA**

*Input.* Array of dimension `nbDims` that contain the size of the tensor for every dimension. The size along unused dimensions should be set to 1. By convention, the ordering of dimensions in the array follows the format - [N, C, D, H, W], with W occupying the smallest index in the array.

**strideA**

*Input.* Array of dimension `nbDims` that contain the stride of the tensor for every dimension. By convention, the ordering of the strides in the array follows the format - [Nstride, Cstride, Dstride, Hstride, Wstride], with Wstride occupying the smallest index in the array.

## 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.

## 3.2.96. cudnnSetTensorNdDescriptorEx()

This function initializes an Nd tensor descriptor.

```

cudnnStatus_t cudnnSetTensorNdDescriptorEx(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                      nbDims,
    const int                dimA[])
    
```

## 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.

Note: Do not use 2 dimensions. Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. For more information, refer to [cudnnGetRNNLinLayerBiasParams\(\)](#).

**dimA***Input.* Array containing the size of each dimension.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

Tensor descriptor was not allocated properly; or input parameters are not set correctly.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

Dimension size requested is larger than maximum dimension size supported.

## 3.2.97. cudnnSetTensorTransformDescriptor()

This function initializes a tensor transform descriptor that was previously created using the [cudnnCreateTensorTransformDescriptor\(\)](#) function.

```
cudnnStatus_t cudnnSetTensorTransformDescriptor(
    cudnnTensorTransformDescriptor_t transformDesc,
    const uint32_t nbDims,
    const cudnnTensorFormat_t destFormat,
    const int32_t padBeforeA[],
    const int32_t padAfterA[],
    const uint32_t foldA[],
    const cudnnFoldingDirection_t direction);
```

## Parameters

**transformDesc***Output.* The tensor transform descriptor to be initialized.**nbDims***Input.* The dimensionality of the transform operands. Must be greater than 2. For more information, refer to [Tensor Descriptor](#).**destFormat***Input.* The desired destination format.**padBeforeA[]***Input.* An array that contains the amount of padding that should be added before each dimension. Set to `NULL` for no padding.**padAfterA[]***Input.* An array that contains the amount of padding that should be added after each dimension. Set to `NULL` for no padding.

**foldA[]**

*Input.* An array that contains the folding parameters for each spatial dimension (dimensions 2 and up). Set to `NULL` for no folding.

**direction**

*Input.* Selects folding or unfolding. This input has no effect when folding parameters are all  $\leq 1$ . For more information, refer to [cudaFoldingDirection\\_t](#).

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function was launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

The parameter `transformDesc` is `NULL`, or if `direction` is invalid, or `nbDims` is  $\leq 2$ .

**CUDNN\_STATUS\_NOT\_SUPPORTED**

If the dimension size requested is larger than maximum dimension size supported (meaning, one of the `nbDims` is larger than `CUDNN_DIM_MAX`), or if `destFormat` is something other than `NCHW` or `NHWC`.

### 3.2.98. **cudaSoftmaxForward()**

This routine computes the softmax function.

```

cudaStatus_t cudaSoftmaxForward(
    cudaHandle_t          handle,
    cudaSoftmaxAlgorithm_t algorithm,
    cudaSoftmaxMode_t    mode,
    const void           *alpha,
    const cudaTensorDescriptor_t xDesc,
    const void           *x,
    const void           *beta,
    const cudaTensorDescriptor_t yDesc,
    void                 *y)
    
```

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.

#### Data Types

This function supports the following data types:

- ▶ `CUDNN_DATA_FLOAT`
- ▶ `CUDNN_DATA_DOUBLE`
- ▶ `CUDNN_DATA_HALF`
- ▶ `CUDNN_DATA_BFLOAT16`
- ▶ `CUDNN_DATA_INT8`

#### Parameters

**handle**

*Input.* Handle to a previously created `cuDNN` context.

**algorithm**

*Input.* Enumerant to specify the softmax algorithm.

**mode**

*Input.* Enumerant to specify the softmax mode.

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc**

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

**x**

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

**yDesc**

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

**y**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions `n`, `c`, `h`, `w` of the input tensor and output tensors differ.
- ▶ The `datatype` of the input tensor and output tensors differ.
- ▶ The parameters `algorithm` or `mode` have an invalid enumerant value.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 3.2.99. cudnnSpatialTfGridGeneratorForward()

This function generates a grid of coordinates in the input tensor corresponding to each pixel from the output tensor.

```
cudaStatus_t cudnnSpatialTfGridGeneratorForward(
    cudaHandle_t          handle,
    const cudnnSpatialTransformerDescriptor_t stDesc,
    const void            *theta,
    void                  *grid)
```

Only 2D transformation is supported.



## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### stDesc

*Input.* Previously created spatial transformer descriptor object.

### theta

*Input.* Affine transformation matrix. It should be of size  $n*2*3$  for a 2d transformation, where  $n$  is the number of images specified in `stDesc`.

### grid

*Output.* A grid of coordinates. It is of size  $n*h*w*2$  for a 2d transformation, where  $n$ ,  $h$ ,  $w$  is specified in `stDesc`. In the 4th dimension, the first coordinate is  $x$ , and the second coordinate is  $y$ .

## Returns

### CUDNN\_STATUS\_SUCCESS

The call was successful.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ `handle` is NULL.
- ▶ One of the parameters `grid` or `theta` is NULL.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration. Refer to the following examples of non-supported configurations:

- ▶ The dimension of the transformed tensor specified in `stDesc`  $> 4$ .

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

## 3.2.100. cudnnSpatialTfSamplerForward()

This function performs a sampler operation and generates the output tensor using the grid given by the grid generator.

```

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)
    
```

Only 2D transformation is supported.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### stDesc

*Input.* Previously created spatial transformer descriptor object.

### alpha, beta

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

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

For more information, refer to [Scaling Parameters](#).

### 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`.

## Returns

### CUDNN\_STATUS\_SUCCESS

The call was successful.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ `handle` is NULL.
- ▶ One of the parameters `x`, `y` or `grid` is NULL.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration. Refer to the following examples of non-supported configurations:

- ▶ The dimension of transformed tensor > 4.

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

### 3.2.101. cudnnTransformFilter()

This function converts the filter between different formats, data types, or dimensions based on the described transformation. It can be used to convert a filter with an unsupported layout format to a filter with a supported layout format.

```

cudnnStatus_t cudnnTransformFilter(
    cudnnHandle_t handle,
    const cudnnTensorTransformDescriptor_t transDesc,

    const void *alpha,
    const cudnnFilterDescriptor_t srcDesc,
    const void *srcData,
    const void *beta,
    const cudnnFilterDescriptor_t destDesc,
    void *destData);

```

This function copies the scaled data from the input filter `srcDesc` to the output tensor `destDesc` with a different layout. If the filter descriptors of `srcDesc` and `destDesc` have different dimensions, they must be consistent with folding and padding amount and order specified in `transDesc`.

The `srcDesc` and `destDesc` tensors must not overlap in any way (that is, tensors cannot be transformed in place).



Note: When performing a folding transform or a zero-padding transform, the scaling factors (`alpha`, `beta`) should be set to (1, 0). However, unfolding transforms support any (`alpha`, `beta`) values. This function is thread safe.

#### Parameters

##### handle

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

##### transDesc

*Input.* A descriptor containing the details of the requested filter transformation. For more information, refer to [cudnnTensorTransformDescriptor\\_t](#).

##### alpha, beta

*Input.* Pointers, in the host memory, to the scaling factors used to scale the data in the input tensor `srcDesc`. `beta` is used to scale the destination tensor, while `alpha` is used to scale the source tensor. For more information, refer to [Scaling Parameters](#).

The `beta` scaling value is not honored in the folding and zero-padding cases. Unfolding supports any (`alpha`, `beta`).

##### srcDesc, destDesc

*Input.* Handles to the previously initiated filter descriptors. `srcDesc` and `destDesc` must not overlap. For more information, refer to [cudnnTensorDescriptor\\_t](#).

##### srcData

*Input.* Pointers, in the host memory, to the data of the tensor described by `srcDesc`.

**destData**

*Output.* Pointers, in the host memory, to the data of the tensor described by destDesc.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

A parameter is uninitialized or initialized incorrectly, or the number of dimensions is different between srcDesc and destDesc.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration. Also, in the folding and padding paths, any value other than A=1 and B=0 will result in a CUDNN\_STATUS\_NOT\_SUPPORTED.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

### 3.2.102. cudnnTransformTensor ()

This function copies the scaled data from one tensor to another tensor with a different layout. Those descriptors need to have the same dimensions but not necessarily the same strides. The input and output tensors must not overlap in any way (meaning, tensors cannot be transformed in place). This function can be used to convert a tensor with an unsupported format to a supported one.

```

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)
    
```

**Parameters**

**handle**

*Input.* Handle to a previously created cuDNN context.

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc**

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudnnTensorDescriptor\\_t](#).

**x**

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

**yDesc**

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**y**

*Output.* Pointer to data of the tensor described by the `yDesc` descriptor.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

The dimensions `n`, `c`, `h`, `w` or the `dataType` of the two tensor descriptors are different.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 3.2.103. cudaTransformTensorEx()

This function converts the tensor layouts between different formats. It can be used to convert a tensor with an unsupported layout format to a tensor with a supported layout format.

```
cudaStatus_t cudaTransformTensorEx(
    cudaHandle_t handle,
    const cudaTensorTransformDescriptor_t transDesc,

    const void *alpha,
    const cudaTensorDescriptor_t srcDesc,
    const void *srcData,
    const void *beta,
    const cudaTensorDescriptor_t destDesc,
    void *destData);
```

This function copies the scaled data from the input tensor `srcDesc` to the output tensor `destDesc` with a different layout. The tensor descriptors of `srcDesc` and `destDesc` should have the same dimensions but need not have the same strides.

The `srcDesc` and `destDesc` tensors must not overlap in any way (that is, tensors cannot be transformed in place).



**Note:** When performing a folding transform or a zero-padding transform, the scaling factors (`alpha`, `beta`) should be set to (1, 0). However, unfolding transforms support any (`alpha`, `beta`) values. This function is thread safe.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

### **transDesc**

*Input.* A descriptor containing the details of the requested tensor transformation. For more information, refer to [cudnnTensorTransformDescriptor\\_t](#).

### **alpha, beta**

*Input.* Pointers, in the host memory, to the scaling factors used to scale the data in the input tensor `srcDesc`. `beta` is used to scale the destination tensor, while `alpha` is used to scale the source tensor. For more information, refer to [Scaling Parameters](#).

The beta scaling value is not honored in the folding and zero-padding cases. Unfolding supports any (`alpha`, `beta`).

### **srcDesc, destDesc**

*Input.* Handles to the previously initiated tensor descriptors. `srcDesc` and `destDesc` must not overlap. For more information, refer to [cudnnTensorDescriptor\\_t](#).

### **srcData**

*Input.* Pointers, in the host memory, to the data of the tensor described by `srcDesc`.

### **destData**

*Output.* Pointers, in the host memory, to the data of the tensor described by `destDesc`.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The function was launched successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

A parameter is uninitialized or initialized incorrectly, or the number of dimensions is different between `srcDesc` and `destDesc`.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

Function does not support the provided configuration. Also, in the folding and padding paths, any value other than `A=1` and `B=0` will result in a `CUDNN_STATUS_NOT_SUPPORTED`.

### **CUDNN\_STATUS\_EXECUTION\_FAILED**

Function failed to launch on the GPU.

---

# Chapter 4. `cuda_ops_train.so` Library

This entity contains common training routines and algorithms, such as batch normalization, softmax, dropout, and so on. The `cuda_ops_train` library depends on `cuda_ops_infer`.

## 4.1. API Functions

These are the API functions in the `cuda_ops_train.so` library.

### 4.1.1. `cudaActivationBackward()`

This routine computes the gradient of a neuron activation function.

```
cudaStatus_t cudaActivationBackward(
    cudaHandle_t          handle,
    cudaActivationDescriptor_t  activationDesc,
    const void           *alpha,
    const cudaTensorDescriptor_t  yDesc,
    const void           *y,
    const cudaTensorDescriptor_t  dyDesc,
    const void           *dy,
    const cudaTensorDescriptor_t  xDesc,
    const void           *x,
    const void           *beta,
    const cudaTensorDescriptor_t  dxDesc,
    void                 *dx)
```

In-place operation is allowed for this routine; meaning `dy` and `dx` pointers may be equal. However, this requires the corresponding tensor descriptors to be identical (particularly, the strides of the input and output must match for an in-place operation to be allowed).

All tensor formats are supported for 4 and 5 dimensions, however, the best performance is obtained when the strides of `yDesc` and `xDesc` are equal and HW-packed. For more than 5 dimensions the tensors must have their spatial dimensions packed.

### Parameters

#### **handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudaHandle\\_t](#).

**activationDesc**

*Input.* Activation descriptor. For more information, refer to [cudaActivationDescriptor\\_t](#).

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**yDesc**

*Input.* Handle to the previously initialized input tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**y**

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

**dyDesc**

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

**dy**

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

**xDesc**

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

**x**

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

**dxDesc**

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

**dx**

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

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

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

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration. Refer to the following examples of non-supported configurations:

- ▶ The dimensions `n`, `c`, `h`, and `w` of the input tensor and output tensor differ.
- ▶ The `datatype` of the input tensor and output tensor differs.



- ▶ The strides `nStride`, `cStride`, `hStride`, and `wStride` of the input tensor and the input differential tensor differ.
- ▶ The strides `nStride`, `cStride`, `hStride`, and `wStride` of the output tensor and the output differential tensor differ.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 4.1.2. `cudaBatchNormalizationBackward()`

This function performs the backward batch normalization layer computation. This layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```

cudaStatus_t cudaBatchNormalizationBackward(
    cudaHandle_t          handle,
    cudaBatchNormMode_t  mode,
    const void            *alphaDataDiff,
    const void            *betaDataDiff,
    const void            *alphaParamDiff,
    const void            *betaParamDiff,
    const cudaTensorDescriptor_t  xDesc,
    const void            *x,
    const cudaTensorDescriptor_t  dyDesc,
    const void            *dy,
    const cudaTensorDescriptor_t  dxDesc,
    void                  *dx,
    const cudaTensorDescriptor_t  bnScaleBiasDiffDesc,
    const void            *bnScale,
    void                  *resultBnScaleDiff,
    void                  *resultBnBiasDiff,
    double                epsilon,
    const void            *savedMean,
    const void            *savedInvVariance)
    
```

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`, and `dx`.

For more information, refer to [cudaDeriveBNTensorDescriptor\(\)](#) for the secondary tensor descriptor generation for the parameters used in this function.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudaHandle\\_t](#).

**mode**

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [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:

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

For more information, refer to [Scaling Parameters](#).

**\*alphaParamDiff, \*betaParamDiff**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc, dxDesc, dyDesc**

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

**\*x**

*Inputs.* 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/Outputs.* 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`, and `savedInvVariance`. The dimensions for this tensor descriptor are dependent on normalization mode. For more information, refer to [cudaDeriveBNTensorDescriptor\(\)](#).



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

**\*bnScale**

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



Note: The `bnBias` parameter is not needed for this layer's computation.

**resultBnScaleDiff, resultBnBiasDiff**


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

**epsilon**

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

**\*savedMean, \*savedInvVariance**

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

 Note: 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.

### Supported configurations

This function supports the following combinations of data types for various descriptors.

Table 14. Supported Configurations for `cudaBatchNormalizationBackward()`

Data Type Configurations	xDesc	bnScaleBiasMean	alpha, beta	yDesc
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_BFLOAT16

### 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, and resultBnBiasDiff is NULL.
- ▶ The number of xDesc, 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`, and `dxDesc`.

### 4.1.3. `cudaBatchNormalizationBackwardEx()`

This function is an extension of the [`cudaBatchNormalizationBackward\(\)`](#) for performing the backward batch normalization layer computation with a fast NHWC semi-persistent kernel.

```

cudaStatus_t cudaBatchNormalizationBackwardEx (
    cudaHandle_t          handle,
    cudaBatchNormMode_t  mode,
    cudaBatchNormOps_t   bnOps,
    const void           *alphaDataDiff,
    const void           *betaDataDiff,
    const void           *alphaParamDiff,
    const void           *betaParamDiff,
    const cudaTensorDescriptor_t xDesc,
    const void           *xData,
    const cudaTensorDescriptor_t yDesc,
    const void           *yData,
    const cudaTensorDescriptor_t dyDesc,
    const void           *dyData,
    const cudaTensorDescriptor_t dzDesc,
    void                *dzData,
    const cudaTensorDescriptor_t dxDesc,
    void                *dxData,
    const cudaTensorDescriptor_t dBnScaleBiasDesc,
    const void           *bnScaleData,
    const void           *bnBiasData,
    void                *dBnScaleData,
    void                *dBnBiasData,
    double              epsilon,
    const void           *savedMean,
    const void           *savedInvVariance,
    const cudaActivationDescriptor_t activationDesc,
    void                *workspace,
    size_t              workspaceSizeInBytes
    void                *reserveSpace
    size_t              reserveSpaceSizeInBytes);

```

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

- ▶ All tensors, namely, `x`, `y`, `dz`, `dy` and `dx` must be NHWC-fully packed, and must be of the type `CUDNN_DATA_HALF`.
- ▶ The input parameter `mode` must be set to `CUDNN_BATCHNORM_SPATIAL_PERSISTENT`.
- ▶ `workspace` is not NULL.
- ▶ Before cuDNN version 8.2.0, the tensor `c` dimension should always be a multiple of 4. After 8.2.0, the tensor `c` dimension should be a multiple of 4 only when `bnOps` is `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION`.
- ▶ `workspaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetBatchNormalizationBackwardExWorkspaceSize\(\)`](#).
- ▶ `reserveSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetBatchNormalizationTrainingExReserveSpaceSize\(\)`](#).

- The content in `reserveSpace` stored by [`cudnnBatchNormalizationForwardTrainingEx\(\)`](#) must be preserved.

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

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

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

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

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

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

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [`cudnnHandle\_t`](#).

### **mode**

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [`cudnnBatchNormMode\_t`](#).

### **bnOps**

*Input.* Mode of operation. Currently, `CUDNN_BATCHNORM_OPS_BN_ACTIVATION` and `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION` are only supported in the NHWC layout. For more information, refer to [`cudnnBatchNormOps\_t`](#). This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

### **\*alphaDataDiff, \*betaDataDiff**

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

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

For more information, refer to [Scaling Parameters](#).

### **\*alphaParamDiff, \*betaParamDiff**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc, \*x, yDesc, \*yData, dyDesc, \*dyData**

*Inputs.* Tensor descriptors and pointers in the device memory for the layer's  $x$  data, backpropagated gradient input  $dy$ , the original forward output  $y$  data.  $yDesc$  and  $yData$  are not needed if  $bnOps$  is set to `CUDNN_BATCHNORM_OPS_BN`, users may pass `NULL`. For more information, refer to [cudaTensorDescriptor\\_t](#).

**dzDesc, dxDesc**

*Inputs.* Tensor descriptors and pointers in the device memory for the computed gradient output  $dz$ , and  $dx$ .  $dzDesc$  is not needed when  $bnOps$  is `CUDNN_BATCHNORM_OPS_BN` or `CUDNN_BATCHNORM_OPS_BN_ACTIVATION`, users may pass `NULL`. For more information, refer to [cudaTensorDescriptor\\_t](#).


**\*dzData, \*dxData**

*Outputs.* Tensor descriptors and pointers in the device memory for the computed gradient output  $dz$ , and  $dx$ .  $*dzData$  is not needed when  $bnOps$  is `CUDNN_BATCHNORM_OPS_BN` or `CUDNN_BATCHNORM_OPS_BN_ACTIVATION`, users may pass `NULL`. For more information, refer to [cudaTensorDescriptor\\_t](#).

**dBnScaleBiasDesc**

*Input.* Shared tensor descriptor for the following six tensors:  $bnScaleData$ ,  $bnBiasData$ ,  $dBnScaleData$ ,  $dBnBiasData$ ,  $savedMean$ , and  $savedInvVariance$ . For more information, refer to [cudaDeriveBNTensorDescriptor\(\)](#).

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



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

For more information, refer to [cudaTensorDescriptor\\_t](#).

**\*bnScaleData**

*Input.* Pointer in the device memory for the batch normalization scale parameter (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, the quantity scale is referred to as  $\gamma$ ).

**\*bnBiasData**

*Input.* Pointers in the device memory for the batch normalization bias parameter (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as  $\beta$ ). This parameter is used only when activation should be performed.

**\*dBnScaleData, \*dBnBiasData**

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

**epsilon**

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

**\*savedMean, \*savedInvVariance**

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

**activationDesc**

*Input.* 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, otherwise the user may pass `NULL`.

**workspace**

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

**workSpaceSizeInBytes**

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

**\*reserveSpace**

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

**reserveSpaceSizeInBytes**

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

### Supported configurations

This function supports the following combinations of data types for various descriptors.

Table 15. Supported Configurations for `cudnnBatchNormalizationBackwardEx()`

Data Type Configurations	xDesc	bnScaleBiasMean	alpha, beta	yDesc
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_BFLOAT16

### 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`, and `resultBnBiasDiff` is NULL.
- ▶ The number of `xDesc`, `yDesc`, or `dxDesc` tensor descriptor dimensions is not within the range of [4, 5] (only 4D and 5D tensors are supported).
- ▶ `dBnScaleBiasDesc` dimensions not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for spatial, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- ▶ 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`, or `dxDesc`.

### 4.1.4. cudnnBatchNormalizationForwardTraining()

This function performs the forward batch normalization layer computation for the training phase. This layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```

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)
    
```

Only 4D and 5D tensors are supported.

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

For the inference phase, use `cudnnBatchNormalizationForwardInference`.

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

Refer to [cudnnDeriveBNTensorDescriptor\(\)](#) for the secondary tensor descriptor generation for the parameters used in this function.



## Parameters

### handle

Handle to a previously created cuDNN library descriptor. For more information, refer to [cudaHandle\\_t](#).

### mode

Mode of operation (spatial or per-activation). For more information, refer to [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:

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

For more information, refer to [Scaling Parameters](#).

### xDesc, yDesc

Tensor descriptors and pointers in device memory for the layer's  $x$  and  $y$  data. For more information, refer to [cudaTensorDescriptor\\_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 [cudaDeriveBNTensorDescriptor\(\)](#). 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 [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as beta and scale as gamma). Note that  $bnBias$  parameter can replace the previous layer's bias parameter for improved efficiency.

### exponentialAverageFactor

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

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

Use a  $factor=1/(1+n)$  at  $N$ -th call to the function to get the Cumulative Moving Average (CMA) behavior, for example:

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

For example:

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

**resultRunningMean, resultRunningVariance**

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

**epsilon**

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

**resultSaveMean, resultSaveInvVariance**

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

Supported configurations

This function supports the following combinations of data types for various descriptors.

Table 16. Supported Configurations for cudnnBatchNormalizationForwardTraining()

Data Type Configurations	xDesc	bnScaleBiasMean	alpha, beta	yDesc
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_BFLOAT16

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

## 4.1.5. cudnnBatchNormalizationForwardTrainingEx ()

This function is an extension of the [cudnnBatchNormalizationForwardTraining\(\)](#) for performing the forward batch normalization layer computation.

```

cudnnStatus_t cudnnBatchNormalizationForwardTrainingEx(
    cudnnHandle_t             handle,
    cudnnBatchNormMode_t     mode,
    cudnnBatchNormOps_t      bnOps,
    const void                *alpha,
    const void                *beta,
    const cudnnTensorDescriptor_t xDesc,
    const void                *xData,
    const cudnnTensorDescriptor_t zDesc,
    const void                *zData,
    const cudnnTensorDescriptor_t yDesc,
    void                      *yData,
    const cudnnTensorDescriptor_t bnScaleBiasMeanVarDesc,
    const void                *bnScaleData,
    const void                *bnBiasData,
    double                    exponentialAverageFactor,
    void                      *resultRunningMeanData,
    void                      *resultRunningVarianceData,
    double                    epsilon,
    void                      *saveMean,
    void                      *saveInvVariance,
    const cudnnActivationDescriptor_t activationDesc,
    void                      *workspace,
    size_t                    workspaceSizeInBytes
    void                      *reserveSpace
    size_t                    reserveSpaceSizeInBytes);

```

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

- ▶ All tensors, namely, `x`, `y`, `dz`, `dy` and `dx` must be NHWC-fully packed and must be of the type `CUDNN_DATA_HALF`.
- ▶ The input parameter `mode` must be set to `CUDNN_BATCHNORM_SPATIAL_PERSISTENT`.
- ▶ `workspace` is not `NULL`.
- ▶ Before cuDNN version 8.2.0, the tensor `c` dimension should always be a multiple of 4. After 8.2.0, the tensor `c` dimension should be a multiple of 4 only when `bnOps` is `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION`.
- ▶ `workSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetBatchNormalizationForwardTrainingExWorkspaceSize\(\)`](#).
- ▶ `reserveSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetBatchNormalizationTrainingExReserveSpaceSize\(\)`](#).
- ▶ The content in `reserveSpace` stored by [`cudaBatchNormalizationForwardTrainingEx\(\)`](#) must be preserved.

If `workspace` is `NULL` and `workSpaceSizeInBytes` of zero is passed in, this API will function exactly like the non-extended function [`cudaBatchNormalizationForwardTraining\(\)`](#).

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`, and `dx`.

## Parameters

### **handle**

Handle to a previously created cuDNN library descriptor. For more information, refer to [`cudaHandle\_t`](#).

### **mode**

Mode of operation (spatial or per-activation). For more information, refer to [`cudaBatchNormMode\_t`](#).

### **bnOps**

*Input.* Mode of operation for the fast NHWC kernel. For more information, refer to [`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.

**\*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:

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

For more information, refer to [Scaling Parameters](#).

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

Tensor descriptors and pointers in device memory for the layer's input  $x$  and output  $y$ , and for the optional  $z$  tensor input for residual addition to the result of the batch normalization operation, prior to the activation. The optional  $zDesc$  and  $*zData$  descriptors are only used when `bnOps` is `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION`, otherwise users may pass `NULL`. When in use,  $z$  should have exactly the same dimension as  $x$  and the final output  $y$ . For more information, refer to [cudnnTensorDescriptor\\_t](#).

**bnScaleBiasMeanVarDesc**

Shared tensor descriptor `desc` for the secondary tensor that was derived by [cudnnDeriveBNTensorDescriptor\(\)](#). The dimensions for this tensor descriptor are dependent on the normalization mode.

**\*bnScaleData, \*bnBiasData**

*Inputs.* Pointers in device memory for the batch normalization scale and bias parameters (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as beta and scale as gamma). Note that `bnBiasData` parameter can replace the previous layer's bias parameter for improved efficiency.

**exponentialAverageFactor**

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

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

Use a `factor=1/(1+n)` at N-th call to the function to get the Cumulative Moving Average (CMA) behavior, for example:

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

For example:

$$\begin{aligned} \text{CMA}[n+1] &= (n * \text{CMA}[n] + \text{x}[n+1]) / (n+1) \\ &= ((n+1) * \text{CMA}[n] - \text{CMA}[n]) / (n+1) + \text{x}[n+1] / (n+1) \\ &= \text{CMA}[n] * (1 - 1 / (n+1)) + \text{x}[n+1] * 1 / (n+1) \\ &= \text{CMA}[n] * (1 - \text{factor}) + \text{x}[n+1] * \text{factor} \end{aligned}$$

**\*resultRunningMeanData, \*resultRunningVarianceData**

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

**epsilon**

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

**\*saveMean, \*saveInvVariance**

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

**activationDesc**

*Input.* The tensor descriptor for the activation operation. When the bnOps input is set to either CUDNN\_BATCHNORM\_OPS\_BN\_ACTIVATION or CUDNN\_BATCHNORM\_OPS\_BN\_ADD\_ACTIVATION then this activation is used, otherwise user may pass NULL.

**\*workspace, workSpaceSizeInBytes**

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

**\*reserveSpace**

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

**reserveSpaceSizeInBytes**

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

Supported configurations

This function supports the following combinations of data types for various descriptors.

Table 17. Supported Configurations for cudnnBatchNormalizationForwardTrainingEx()

Data Type Configurations	xDesc	bnScaleBiasMean	alpha, beta	yDesc
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_BFLOAT16

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

## 4.1.6. cudnnDivisiveNormalizationBackward()

This function performs the backward `DivisiveNormalization` layer computation.

```

cudnnStatus_t cudnnDivisiveNormalizationBackward(
    cudnnHandle_t          handle,
    cudnnLRNDescriptor_t   normDesc,
    cudnnDivNormMode_t     mode,
    const void             *alpha,
    const cudnnTensorDescriptor_t xDesc,
    const void             *x,
    const void             *means,
    const void             *dy,
    void                  *temp,
    void                  *temp2,
    const void             *beta,
    const cudnnTensorDescriptor_t dxDesc,
    void                  *dx,
    void                  *dMeans)

```

Supported tensor formats are NCHW for 4D and NCDHW for 5D with any non-overlapping non-negative strides. Only 4D and 5D tensors are supported.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor.

**normDesc**

*Input.* Handle to a previously initialized LRN parameter descriptor (this descriptor is used for both LRN and `DivisiveNormalization` layers).

**mode**

*Input.* `DivisiveNormalization` layer mode of operation. Currently only `CUDNN_DIVNORM_PRECOMPUTED_MEANS` is implemented. Normalization is performed using the means input tensor that is expected to be precomputed by the user.

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc, x, means**

*Input.* Tensor descriptor and pointers in device memory for the layer's `x` and means data. Note that the `means` tensor is expected to be precomputed by the user. It can also contain any valid values (not required to be actual `means`, and can be for instance a result of a convolution with a Gaussian kernel).

**dy**

*Input.* Tensor pointer in device memory for the layer's `dy` cumulative loss differential data (error backpropagation).

**temp, temp2**

*Workspace.* Temporary tensors in device memory. These are used for computing intermediate values during the backward pass. These tensors do not have to be preserved from forward to backward pass. Both use `xDesc` as a descriptor.

**dxDesc**

*Input.* Tensor descriptor for `dx` and `dMeans`.

**dx, dMeans**

*Output.* Tensor pointers (in device memory) for the layers resulting in cumulative gradients `dx` and `dMeans` ( $dLoss/dx$  and  $dLoss/dMeans$ ). Both share the same descriptor.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the tensor pointers `x`, `dx`, `temp`, `temp2`, and `dy` is `NULL`.
- ▶ Number of any of the input or output tensor dimensions is not within the `[4, 5]` range.
- ▶ Either `alpha` or `beta` pointer is `NULL`.
- ▶ A mismatch in dimensions between `xDesc` and `dxDesc`.



- ▶ LRN descriptor parameters are outside of their valid ranges.
- ▶ Any of the tensor strides is negative.

**CUDNN\_STATUS\_UNSUPPORTED**

The function does not support the provided configuration, for example, any of the input and output tensor strides mismatch (for the same dimension) is a non-supported configuration.

### 4.1.7. cudnnDropoutBackward()

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.

```

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)
    
```

Better performance is obtained for fully packed tensors.

#### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**dropoutDesc**

*Input.* Previously created dropout descriptor object.

**dyDesc**

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

**dy**

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

**dxDesc**

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

**dx**

*Output.* Pointer to data of the tensor described by the `dxDesc` descriptor.

**reserveSpace**

*Input.* Pointer to user-allocated GPU memory used by this function. It is expected that `reserveSpace` was populated during a call to `cudnnDropoutForward` and has not been changed.

**reserveSpaceSizeInBytes**

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The call was successful.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ The number of elements of input tensor and output tensors differ.
- ▶ The `datatype` of the input tensor and output tensors differs.
- ▶ The strides of the input tensor and output tensors differ and in-place operation is used (i.e., `x` and `y` pointers are equal).
- ▶ The provided `reserveSpaceSizeInBytes` is less 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.

## 4.1.8. `cudaDnnGetBatchNormalizationBackwardExWorkspaceSize`

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

```

cudaDnnStatus_t cudaDnnGetBatchNormalizationBackwardExWorkspaceSize(
    cudaDnnHandle_t          handle,
    cudaDnnBatchNormMode_t   mode,
    cudaDnnBatchNormOps_t    bnOps,
    const cudaDnnTensorDescriptor_t xDesc,
    const cudaDnnTensorDescriptor_t yDesc,
    const cudaDnnTensorDescriptor_t dyDesc,
    const cudaDnnTensorDescriptor_t dzDesc,
    const cudaDnnTensorDescriptor_t dxDesc,
    const cudaDnnTensorDescriptor_t dBnScaleBiasDesc,
    const cudaDnnActivationDescriptor_t activationDesc,
    size_t                   *sizeInBytes);

```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudaDnnHandle\\_t](#).

### mode

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [cudaDnnBatchNormMode\\_t](#).

**bnOps**

*Input.* Mode of operation for the fast NHWC kernel. For more information, refer to [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, yDesc, dyDesc, dzDesc, dxDesc**

Tensor descriptors and pointers in the device memory for the layer's  $x$  data, back propagated differential  $dy$  (inputs), the optional  $y$  input data, the optional  $dz$  output, and the  $dx$  output, which is the resulting differential with respect to  $x$ . For more information, refer to [cudaTensorDescriptor\\_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 [cudaDeriveBNTensorDescriptor\(\)](#). The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be `float` for FP16 and FP32 input tensors, and `double` for FP64 input tensors.

**activationDesc**

*Input.* 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, otherwise user may pass `NULL`.

**\*sizeInBytes**

*Output.* Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the [cudaGetBatchNormalizationForwardTrainingExWorkspaceSize\(\)](#) function with the specified `bnOps` input setting.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ Number of `xDesc`, `yDesc` or `dxDesc` tensor descriptor dimensions is not within the range of [4, 5] (only 4D and 5D tensors are supported).
- ▶ `dBnScaleBiasDesc` dimensions not `1xCx1x1` for 4D and `1xCx1x1x1` for 5D for spatial, and are not `1xCxHxW` for 4D and `1xCxDxHxW` for 5D for per-activation mode.
- ▶ Dimensions or data types mismatch for any pair of `xDesc`, `dyDesc`, or `dxDesc`.

## 4.1.9. cudnnGetBatchNormalizationForwardTrainingExW

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

```

cudnnStatus_t cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize(
    cudnnHandle_t          handle,
    cudnnBatchNormMode_t  mode,
    cudnnBatchNormOps_t   bnOps,
    const cudnnTensorDescriptor_t xDesc,
    const cudnnTensorDescriptor_t zDesc,
    const cudnnTensorDescriptor_t yDesc,
    const cudnnTensorDescriptor_t bnScaleBiasMeanVarDesc,
    const cudnnActivationDescriptor_t activationDesc,
    size_t                 *sizeInBytes);

```

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

**mode**

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [cudnnBatchNormMode\\_t](#).

**bnOps**

*Input.* Mode of operation for the fast NHWC kernel. For more information, refer to [cudnnBatchNormOps\\_t](#). This input can be used to set this function to perform either only the batch normalization, or batch normalization followed by activation, or batch normalization followed by element-wise addition and then activation.

**xDesc, zDesc, yDesc**

Tensor descriptors and pointers in the device memory for the layer's `x` data, the optional `z` input data, and the `y` output. `zDesc` is only needed when `bnOps` is `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION`, otherwise the user may pass `NULL`. For more information, refer to [cudnnTensorDescriptor\\_t](#).

**bnScaleBiasMeanVarDesc**

*Input.* Shared tensor descriptor for the following six tensors: `bnScaleData`, `bnBiasData`, `dBnScaleData`, `dBnBiasData`, `savedMean`, and `savedInvVariance`. This is the shared tensor descriptor `desc` for the secondary tensor that was derived by [cudnnDeriveBNTensorDescriptor\(\)](#). The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be `float` for FP16 and FP32 input tensors, and `double` for FP64 input tensors.

**activationDesc**

*Input.* 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, otherwise the user may pass `NULL`.

**\*sizeInBytes**

*Output.* Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the `cudaGetBatchNormalizationForwardTrainingExWorkspaceSize()` function with the specified `bnOps` input setting.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ Number of `xDesc`, `yDesc` or `dxDesc` tensor descriptor dimensions is not within the range of [4, 5] (only 4D and 5D tensors are supported).
- ▶ `dBnScaleBiasDesc` dimensions not `1xCx1x1` for 4D and `1xCx1x1x1` for 5D for spatial, and are not `1xCxHxW` for 4D and `1xCxDxHxW` for 5D for per-activation mode.
- ▶ Dimensions or data types mismatch for `xDesc` or `yDesc`.

**4.1.10. cudaGetBatchNormalizationTrainingExReserveSpaceSize**

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.

```

cudaStatus_t cudaGetBatchNormalizationTrainingExReserveSpaceSize(
    cudaHandle_t          handle,
    cudaBatchNormMode_t   mode,
    cudaBatchNormOps_t    bnOps,
    const cudaActivationDescriptor_t activationDesc,
    const cudaTensorDescriptor_t xDesc,
    size_t                *sizeInBytes);
    
```

**Parameters**

**handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudaHandle\\_t](#).

**mode**

*Input.* Mode of operation (spatial or per-activation). For more information, refer to [cudaBatchNormMode\\_t](#).

**bnOps**

*Input.* Mode of operation for the fast NHWC kernel. For more information, refer to [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**

Tensor descriptors for the layer's  $x$  data. For more information, refer to [cudnnTensorDescriptor\\_t](#).

**activationDesc**

*Input.* 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, otherwise user may pass `NULL`.

**\*sizeInBytes**

*Output.* Amount of GPU memory reserved.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The `xDesc` tensor descriptor dimension is not within the `[4, 5]` range (only 4D and 5D tensors are supported).

**4.1.11. cudnnGetNormalizationBackwardWorkspaceSize()**

This function returns the amount of GPU memory workspace the user should allocate to be able to call [cudnnNormalizationBackward\(\)](#) function for the specified `normOps` and `algo` input setting. The workspace allocated will then be passed to the function [cudnnNormalizationBackward\(\)](#).

```

cudnnStatus_t
cudnnGetNormalizationBackwardWorkspaceSize(cudnnHandle_t handle,
                                           cudnnNormMode_t mode,
                                           cudnnNormOps_t normOps,
                                           cudnnNormAlgo_t algo,
                                           const cudnnTensorDescriptor_t xDesc,
                                           const cudnnTensorDescriptor_t yDesc,
                                           const cudnnTensorDescriptor_t dyDesc,
                                           const cudnnTensorDescriptor_t dzDesc,
                                           const cudnnTensorDescriptor_t dxDesc,
                                           const cudnnTensorDescriptor_t
dNormScaleBiasDesc,
                                           const cudnnActivationDescriptor_t
activationDesc,
                                           const cudnnTensorDescriptor_t
normMeanVarDesc,
                                           size_t *sizeInBytes,
                                           int groupCnt);
    
```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

### mode

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [cudnnNormMode\\_t](#).

### normOps

*Input.* Mode of post-operative. Currently CUDNN\_NORM\_OPS\_NORM\_ACTIVATION and CUDNN\_NORM\_OPS\_NORM\_ADD\_ACTIVATION are only supported in the NHWC layout. For more information, refer to [cudnnNormOps\\_t](#). This input can be used to set this function to perform either only the normalization, or normalization followed by activation, or normalization followed by element-wise addition and then activation.

### algo

*Input.* Algorithm to be performed. For more information, refer to [cudnnNormAlgo\\_t](#).

### xDesc, yDesc, dyDesc, dzDesc, dxDesc

Tensor descriptors and pointers in the device memory for the layer's  $x$  data, back propagated differential  $dy$  (inputs), the optional  $y$  input data, the optional  $dz$  output, and the  $dx$  output, which is the resulting differential with respect to  $x$ . For more information, refer to [cudnnTensorDescriptor\\_t](#).

### dNormScaleBiasDesc

*Input.* Shared tensor descriptor for the following four tensors: normScaleData, normBiasData, dNormScaleData, dNormBiasData. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be float for FP16 and FP32 input tensors, and double for FP64 input tensors.

### activationDesc

*Input.* Descriptor for the activation operation. When the normOps input is set to either CUDNN\_NORM\_OPS\_NORM\_ACTIVATION or CUDNN\_NORM\_OPS\_NORM\_ADD\_ACTIVATION, then this activation is used, otherwise the user may pass NULL.

### normMeanVarDesc

*Input.* Shared tensor descriptor for the following tensors: savedMean and savedInvVariance. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be float for FP16 and FP32 input tensors, and double for FP64 input tensors.

### \*sizeInBytes

*Output.* Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the [cudnnGetNormalizationForwardTrainingWorkspaceSize\(\)](#) function with the specified normOps input setting.

### groupCnt

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

## Returns

### CUDNN\_STATUS\_SUCCESS

The computation was performed successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ Number of `xDesc`, `yDesc` or `dxDesc` tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- ▶ `dNormScaleBiasDesc` dimensions not `1xCx1x1` for 4D and `1xCx1x1x1` for 5D for per-channel, 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`, or `dxDesc`.

## 4.1.12. cudnnGetNormalizationForwardTrainingWorkspace

This function returns the amount of GPU memory workspace the user should allocate to be able to call [cudnnNormalizationForwardTraining\(\)](#) function for the specified `normOps` and `algo` input setting. The workspace allocated should then be passed by the user to the function [cudnnNormalizationForwardTraining\(\)](#).

```

cudnnStatus_t
cudnnGetNormalizationForwardTrainingWorkspaceSize(cudnnHandle_t handle,
                                                  cudnnNormMode_t mode,
                                                  cudnnNormOps_t normOps,
                                                  cudnnNormAlgo_t algo,
                                                  const cudnnTensorDescriptor_t xDesc,
                                                  const cudnnTensorDescriptor_t
zDesc,
                                                  const cudnnTensorDescriptor_t
yDesc,
                                                  const cudnnTensorDescriptor_t
normScaleBiasDesc,
                                                  const cudnnActivationDescriptor_t
activationDesc,
                                                  const cudnnTensorDescriptor_t
normMeanVarDesc,
                                                  size_t *sizeInBytes,
                                                  int groupCnt);

```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

### mode

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [cudnnNormMode\\_t](#).



**normOps**

*Input.* Mode of post-operative. Currently `CUDNN_NORM_OPS_NORM_ACTIVATION` and `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` are only supported in the NHWC layout. For more information, refer to [cudnnNormOps\\_t](#). This input can be used to set this function to perform either only the normalization, or normalization followed by activation, or normalization followed by element-wise addition and then activation.

**algo**

*Input.* Algorithm to be performed. For more information, refer to [cudnnNormAlgo\\_t](#).

**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. `zDesc` is only needed when `normOps` is `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION`, otherwise the user may pass `NULL`. For more information, refer to [cudnnTensorDescriptor\\_t](#).

**normScaleBiasDesc**

*Input.* Shared tensor descriptor for the following tensors: `normScaleData` and `normBiasData`. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be float for FP16 and FP32 input tensors, and double for FP64 input tensors.

**activationDesc**

*Input.* Descriptor for the activation operation. When the `normOps` input is set to either `CUDNN_NORM_OPS_NORM_ACTIVATION` or `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION`, then this activation is used, otherwise the user may pass `NULL`.

**normMeanVarDesc**

*Input.* Shared tensor descriptor for the following tensors: `savedMean` and `savedInvVariance`. The dimensions for this tensor descriptor are dependent on normalization mode. Note that the data type of this tensor descriptor must be float for FP16 and FP32 input tensors, and double for FP64 input tensors.

**\*sizeInBytes**

*Output.* Amount of GPU memory required for the workspace, as determined by this function, to be able to execute the [cudnnGetNormalizationForwardTrainingWorkspaceSize\(\)](#) function with the specified `normOps` input setting.

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

**Returns****CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ Number of `xDesc`, `yDesc` or `zDesc` tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- ▶ `normScaleBiasDesc` dimensions not `1xCx1x1` for 4D and `1xCx1x1x1` for 5D for per-channel, and are not `1xCxHxW` for 4D and `1xCxDxHxW` for 5D for per-activation mode.
- ▶ Dimensions or data types mismatch for `xDesc` or `yDesc`.

### 4.1.13. cudnnGetNormalizationTrainingReserveSpaceSize

This function returns the amount of reserve GPU memory workspace the user should allocate for the normalization operation, for the specified `normOps` 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.

```

cudnnStatus_t
cudnnGetNormalizationTrainingReserveSpaceSize(cudnnHandle_t handle,
                                              cudnnNormMode_t mode,
                                              cudnnNormOps_t normOps,
                                              cudnnNormAlgo_t algo,
                                              const cudnnActivationDescriptor_t
activationDesc,
                                              const cudnnTensorDescriptor_t xDesc,
                                              size_t *sizeInBytes,
                                              int groupCnt);

```

#### Parameters

##### handle

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [cudnnHandle\\_t](#).

##### mode

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [cudnnNormMode\\_t](#).

##### normOps

*Input.* Mode of post-operative. Currently `CUDNN_NORM_OPS_NORM_ACTIVATION` and `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` are only supported in the NHWC layout. For more information, refer to [cudnnNormOps\\_t](#). This input can be used to set this function to perform either only the normalization, or normalization followed by activation, or normalization followed by element-wise addition and then activation.

##### algo

*Input.* Algorithm to be performed. For more information, refer to [cudnnNormAlgo\\_t](#).

##### xDesc

Tensor descriptors for the layer's `x` data. For more information, refer to [cudnnTensorDescriptor\\_t](#).

##### activationDesc

*Input.* Descriptor for the activation operation. When the `normOps` input is set to either `CUDNN_NORM_OPS_NORM_ACTIVATION` or `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` then this activation is used, otherwise the user may pass `NULL`.

**\*sizeInBytes**

*Output.* Amount of GPU memory reserved.

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

## 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).

### 4.1.14. `cudaLrncrossChannelBackward()`

This function performs the backward LRN layer computation.

```

cudaStatus_t cudaLrncrossChannelBackward(
    cudaHandle_t          handle,
    cudaLRNDescriptor_t   normDesc,
    cudaLRNMode_t         lrnMode,
    const void            *alpha,
    const cudaTensorDescriptor_t yDesc,
    const void            *y,
    const cudaTensorDescriptor_t dyDesc,
    const void            *dy,
    const cudaTensorDescriptor_t xDesc,
    const void            *x,
    const void            *beta,
    const cudaTensorDescriptor_t dxDesc,
    void                  *dx)
    
```

Supported formats are: positive-strided, NCHW and NHWC 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. NCHW layout is preferred for performance.

## Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor.

**normDesc**

*Input.* Handle to a previously initialized LRN parameter descriptor.

**lrnMode**

*Input.* LRN layer mode of operation. Currently, only `CUDNN_LRN_CROSS_CHANNEL_DIM1` is implemented. Normalization is performed along the tensor's `dimA[1]`.

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**yDesc, y**

*Input.* Tensor descriptor and pointer in device memory for the layer's  $y$  data.

**dyDesc, dy**

*Input.* Tensor descriptor and pointer in device memory for the layer's input cumulative loss differential data  $dy$  (including error backpropagation).

**xDesc, x**

*Input.* Tensor descriptor and pointer in device memory for the layer's  $x$  data. Note that these values are not modified during backpropagation.

**dxDesc, dx**

*Output.* Tensor descriptor and pointer in device memory for the layer's resulting cumulative loss differential data  $dx$  (including error backpropagation).

## Returns

**CUDNN\_STATUS\_SUCCESS**

The computation was performed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the tensor pointers  $x$ ,  $y$  is NULL.
- ▶ Number of input tensor dimensions is 2 or less.
- ▶ LRN descriptor parameters are outside of their valid ranges.
- ▶ One of the 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  $x$ ,  $y$ ,  $dx$ , or  $dy$ .
- ▶ Any tensor parameters strides are negative.

## 4.1.15. cudnnNormalizationBackward()

This function performs backward normalization layer computation that is specified by mode. Per-channel normalization layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```
cudnnStatus_t  
cudnnNormalizationBackward(cudnnHandle_t handle,
```

```

        cudnnNormMode_t mode,
        cudnnNormOps_t normOps,
        cudnnNormAlgo_t algo,
        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 dNormScaleBiasDesc,
        const void *normScaleData,
        const void *normBiasData,
        void *dNormScaleData,
        void *dNormBiasData,
        double epsilon,
        const cudnnTensorDescriptor_t normMeanVarDesc,
        const void *savedMean,
        const void *savedInvVariance,
        cudnnActivationDescriptor_t activationDesc,
        void *workSpace,
        size_t workSpaceSizeInBytes,
        void *reserveSpace,
        size_t reserveSpaceSizeInBytes,
int groupCnt)

```

Only 4D and 5D tensors are supported.

The `epsilon` value has to be the same during training, backpropagation, and inference. 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 function can accept a `*workspace` pointer to the GPU workspace, and `workSpaceSizeInBytes`, the size of the workspace, from the user.

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

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

Higher performance for `CUDNN_NORM_PER_CHANNEL` mode can be obtained when the following conditions are true:

- ▶ All tensors, namely, `x`, `y`, `dz`, `dy`, and `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_NORM_PER_CHANNEL`.
- ▶ The input parameter `algo` must be set to `CUDNN_NORM_ALGO_PERSIST`.
- ▶ Workspace is not `NULL`.

- ▶ `workSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetNormalizationBackwardWorkspaceSize\(\)`](#).
- ▶ `reserveSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetNormalizationTrainingReserveSpaceSize\(\)`](#).
- ▶ The content in `reserveSpace` stored by [`cudaNormalizationForwardTraining\(\)`](#) must be preserved.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [`cudaHandle\_t`](#).

### **mode**

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [`cudaNormMode\_t`](#).

### **normOps**

*Input.* Mode of post-operative. Currently `CUDNN_NORM_OPS_NORM_ACTIVATION` and `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` are only supported in the NHWC layout. For more information, refer to [`cudaNormOps\_t`](#). This input can be used to set this function to perform either only the normalization, or normalization followed by activation, or normalization followed by element-wise addition and then activation.

### **algo**

*Input.* Algorithm to be performed. For more information, refer to [`cudaNormAlgo\_t`](#).

### **\*alphaDataDiff, \*betaDataDiff**

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

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

For more information, refer to [Scaling Parameters](#).

### **\*alphaParamDiff, \*betaParamDiff**

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

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

For more information, refer to [Scaling Parameters](#).

### **xDesc, \*xData, yDesc, \*yData, dyDesc, \*dyData**

*Inputs.* Tensor descriptors and pointers in the device memory for the layer's x data, backpropagated gradient input `dy`, the original forward output `y` data. `yDesc` and `yData` are not needed if `normOps` is set to `CUDNN_NORM_OPS_NORM`, users may pass `NULL`. For more information, refer to [`cudaTensorDescriptor\_t`](#).

**dzDesc, dxDesc**


*Inputs.* Tensor descriptors and pointers in the device memory for the computed gradient output `dz` and `dx`. `dzDesc` is not needed when `normOps` is `CUDNN_NORM_OPS_NORM` or `CUDNN_NORM_OPS_NORM_ACTIVATION`, users may pass `NULL`. For more information, refer to [cudaTensorDescriptor\\_t](#).

**\*dzData, \*dxData**

*Outputs.* Tensor descriptors and pointers in the device memory for the computed gradient output `dz` and `dx`. `*dzData` is not needed when `normOps` is `CUDNN_NORM_OPS_NORM` or `CUDNN_NORM_OPS_NORM_ACTIVATION`, users may pass `NULL`. For more information, refer to [cudaTensorDescriptor\\_t](#).

**dNormScaleBiasDesc**

*Input.* Shared tensor descriptor for the following six tensors: `normScaleData`, `normBiasData`, `dNormScaleData`, and `dNormBiasData`. The dimensions for this tensor descriptor are dependent on normalization mode.

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

For more information, refer to [cudaTensorDescriptor\\_t](#).

**\*normScaleData**

*Input.* Pointer in the device memory for the normalization scale parameter (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, the quantity scale is referred to as gamma).

**\*normBiasData**

*Input.* Pointers in the device memory for the normalization bias parameter (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as beta). This parameter is used only when activation should be performed.

**\*dNormScaleData, \*dNormBiasData**


*Outputs.* Pointers in the device memory for the gradients of `normScaleData` and `normBiasData`, respectively.

**epsilon**

*Input.* Epsilon value used in normalization formula. Its value should be equal to or greater than zero. The same epsilon value should be used in forward and backward functions.

**normMeanVarDesc**

*Input.* Shared tensor descriptor for the following tensors: `savedMean` and `savedInvVariance`. The dimensions for this tensor descriptor are dependent on normalization mode.

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

For more information, refer to [cudnnTensorDescriptor\\_t](#).

**\*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 `normScaleData`, `normBiasData` 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.* Descriptor for the activation operation. When the `normOps` input is set to either `CUDNN_NORM_OPS_NORM_ACTIVATION` or `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` then this activation is used, otherwise the user may pass `NULL`.

**workspace**

*Input.* Pointer to the GPU workspace.

**workspaceSizeInBytes**

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

**\*reserveSpace**

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

**reserveSpaceSizeInBytes**

*Input.* The size of the `reserveSpace`. It must be equal or larger than the amount required by [cudnnGetNormalizationTrainingReserveSpaceSize\(\)](#).

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

## 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`, `xData`, `dyData`, `dxData`, `normScaleData`, `dNormScaleData`, and `dNormBiasData` is `NULL`.
- ▶ The number of `xDesc`, `yDesc`, or `dxDesc` tensor descriptor dimensions is not within the range of [4,5] (only 4D and 5D tensors are supported).
- ▶ `dNormScaleBiasDesc` dimensions not `1xCx1x1` for 4D and `1xCx1x1x1` for 5D for per-channel, 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 zero.



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

### 4.1.16. cudnnNormalizationForwardTraining()

This function performs the forward normalization layer computation for the training phase. Depending on mode, different normalization operations will be performed. Per-channel layer is based on the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

```

cudnnStatus_t
cudnnNormalizationForwardTraining(cudnnHandle_t handle,
    cudnnNormMode_t mode,
    cudnnNormOps_t normOps,
    cudnnNormAlgo_t algo,
    const void *alpha,
    const void *beta,
    const cudnnTensorDescriptor_t xDesc,
    const void *xData,
    const cudnnTensorDescriptor_t normScaleBiasDesc,
    const void *normScale,
    const void *normBias,
    double exponentialAverageFactor,
    const cudnnTensorDescriptor_t normMeanVarDesc,
    void *resultRunningMean,
    void *resultRunningVariance,
    double epsilon,
    void *resultSaveMean,
    void *resultSaveInvVariance,
    cudnnActivationDescriptor_t activationDesc,
    const cudnnTensorDescriptor_t zDesc,
    const void *zData,
    const cudnnTensorDescriptor_t yDesc,
    void *yData,
    void *workspace,
    size_t workSpaceSizeInBytes,
    void *reserveSpace,
    size_t reserveSpaceSizeInBytes,
    int groupCnt);
    
```

Only 4D and 5D tensors are supported.

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

For the inference phase, refer to [cudnnNormalizationForwardInference\(\)](#).

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

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

- ▶ All tensors, namely, `xData`, `yData` 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_NORM_PER_CHANNEL`.
- ▶ The input parameter `algo` must be set to `CUDNN_NORM_ALGO_PERSIST`.
- ▶ `workspace` is not `NULL`.

- ▶ `workspaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetNormalizationForwardTrainingWorkspaceSize\(\)`](#).
- ▶ `reserveSpaceSizeInBytes` is equal to or larger than the amount required by [`cudaGetNormalizationTrainingReserveSpaceSize\(\)`](#).
- ▶ The content in `reserveSpace` stored by [`cudaNormalizationForwardTraining\(\)`](#) must be preserved.

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 `normOps` input can be used to set this function to perform either only the normalization, or normalization followed by activation, or 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 `xData`, `yData`.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor. For more information, refer to [`cudaHandle\_t`](#).

### **mode**

*Input.* Mode of operation (per-channel or per-activation). For more information, refer to [`cudaNormMode\_t`](#).

### **normOps**

*Input.* Mode of post-operative. Currently `CUDNN_NORM_OPS_NORM_ACTIVATION` and `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` are only supported in the NHWC layout. For more information, refer to [`cudaNormOps\_t`](#). This input can be used to set this function to perform either only the normalization, or normalization followed by activation, or normalization followed by element-wise addition and then activation.

### **algo**

*Input.* Algorithm to be performed. For more information, refer to [`cudaNormAlgo\_t`](#).

### **\*alpha, \*beta**

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

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

For more information, refer to [Scaling Parameters](#).

### **xDesc, yDesc**

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

**\*xData**

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

**\*yData**

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

**zDesc, \*zData**

*Input.* Tensor descriptors and pointers in device memory for residual addition to the result of the normalization operation, prior to the activation. `zDesc` and `*zData` are optional and are only used when `normOps` is `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION`, otherwise the user may pass `NULL`. When in use, `z` should have exactly the same dimension as `xData` and the final output `yData`. For more information, refer to [cudaTensorDescriptor\\_t](#).

**normScaleBiasDesc, normScale, normBias**

*Inputs.* Tensor descriptors and pointers in device memory for the normalization scale and bias parameters (in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper, bias is referred to as beta and scale as gamma). The dimensions for the tensor descriptor are dependent on the normalization mode.

**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 the Cumulative Moving Average (CMA) behavior, for example:

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

For example:

$$\begin{aligned} \text{CMA}[n+1] &= (n * \text{CMA}[n] + \text{x}[n+1]) / (n+1) \\ &= ((n+1) * \text{CMA}[n] - \text{CMA}[n]) / (n+1) + \text{x}[n+1] / (n+1) \\ &= \text{CMA}[n] * (1 - 1 / (n+1)) + \text{x}[n+1] * 1 / (n+1) \\ &= \text{CMA}[n] * (1 - \text{factor}) + \text{x}[n+1] * \text{factor} \end{aligned}$$

**normMeanVarDesc**

*Inputs.* Tensor descriptor used for following tensors: `resultRunningMean`, `resultRunningVariance`, `resultSaveMean`, `resultSaveInvVariance`.

**\*resultRunningMean, \*resultRunningVariance**

*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 `resultRunningVariance` (or passed as an input in inference mode) is the sample variance and is the moving average of `variance[x]` where the variance is computed either over batch or spatial+batch dimensions depending on the mode. If these pointers are not `NULL`, the tensors should be initialized to some reasonable values or to 0.

**epsilon**

*Input.* Epsilon value used in the normalization formula. Its value should be equal to or greater than zero.

**\*resultSaveMean, \*resultSaveInvVariance**

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

**activationDesc**

*Input.* The tensor descriptor for the activation operation. When the `normOps` input is set to either `CUDNN_NORM_OPS_NORM_ACTIVATION` or `CUDNN_NORM_OPS_NORM_ADD_ACTIVATION` then this activation is used, otherwise the user may pass `NULL`.

**\*workspace, workspaceSizeInBytes**

*Inputs.* `*workspace` is a pointer to the GPU workspace, and `workspaceSizeInBytes` is the size of the workspace. When `*workspace` is not `NULL` and `*workspaceSizeInBytes` is large enough, and the tensor layout is `NHWC` and the data type configuration is supported, then this function will trigger a semi-persistent `NHWC` kernel for 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 `cudaGetNormalizationTrainingReserveSpaceSize()`.

**groupCnt**

*Input.* The number of grouped convolutions. Currently, only 1 is supported.

### Supported configurations

This function supports the following combinations of data types for various descriptors.

Table 18. Supported Configurations for `cudaNormalizationForwardTraining()`

Data Type Configurations	xDesc, yDesc, zDesc	normScaleBiasDesc, normMeanVarDesc
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
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT

## 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`, `xData`, `yData`, `normScale`, and `normBias` is `NULL`.
- ▶ The number of `xDesc` or `yDesc` tensor descriptor dimensions is not within the [4,5] range (only 4D and 5D tensors are supported).
- ▶ `normScaleBiasDesc` dimensions are not 1xCx1x1 for 4D and 1xCx1x1x1 for 5D for per-channel mode, and are not 1xCxHxW for 4D and 1xCxDxHxW for 5D for per-activation mode.
- ▶ Exactly one of `resultSaveMean`, `resultSaveInvVariance` pointers are `NULL`.
- ▶ Exactly one of `resultRunningMean`, `resultRunningInvVariance` pointers are `NULL`.
- ▶ `epsilon` value is less than zero.
- ▶ Dimensions or data types mismatch for `xDesc` or `yDesc`.

## 4.1.17. cudnnOpsTrainVersionCheck()

This function checks whether the version of the `OpsTrain` subset of the library is consistent with the other sub-libraries.

```
cudaStatus_t cudnnOpsTrainVersionCheck(void)
```

## Returns

### CUDNN\_STATUS\_SUCCESS

The version is consistent with other sub-libraries.

### CUDNN\_STATUS\_VERSION\_MISMATCH

The version of `OpsTrain` is not consistent with other sub-libraries. Users should check the installation and make sure all sub-component versions are consistent.

## 4.1.18. cudnnPoolingBackward()

This function computes the gradient of a pooling operation.

```
cudaStatus_t cudnnPoolingBackward(
    cudaHandle_t          handle,
    const cudaPoolingDescriptor_t poolingDesc,
    const void           *alpha,
    const cudaTensorDescriptor_t yDesc,
    const void           *y,
    const cudaTensorDescriptor_t dyDesc,
    const void           *dy,
    const cudaTensorDescriptor_t xDesc,
    const void           *xData,
    const void           *beta,
```

```
const cudnnTensorDescriptor_t dxDesc,
void *dx)
```

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.



Note: Tensor vectorization is not supported for any tensor descriptor arguments in this function. Best performance is expected when using HW-packed tensors. Only 2 and 3 spatial dimensions are supported.

`cudnnPoolingBackward()` allows both `x` and `y` data pointers (together with the related tensor descriptor handles) to be `NULL` for avg-pooling. This could save memory footprint and bandwidth.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### poolingDesc

*Input.* Handle to the previously initialized pooling descriptor.

### alpha, beta

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

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

For more information, refer to [Scaling Parameters](#).

### yDesc

*Input.* Handle to the previously initialized input tensor descriptor. Can be `NULL` for average pooling.

### y

*Input.* Data pointer to GPU memory associated with the tensor descriptor `yDesc`. Can be `NULL` for average pooling.

### dyDesc

*Input.* Handle to the previously initialized input differential tensor descriptor. Must be of type `FLOAT`, `DOUBLE`, `HALF`, or `BFLOAT16`. For more information, refer to [cudnnDataType\\_t](#).

### dy

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

### xDesc

*Input.* Handle to the previously initialized output tensor descriptor. Can be `NULL` for average pooling.

### x

*Input.* Data pointer to GPU memory associated with the output tensor descriptor `xDesc`. Can be `NULL` for average pooling.

**dxDesc**

*Input.* Handle to the previously initialized output differential tensor descriptor. Must be of type `FLOAT`, `DOUBLE`, `HALF`, or `BFLOAT16`. For more information, refer to [cudaDataType\\_t](#).

**dx**

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

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions `n`, `c`, `h`, `w` of the `yDesc` and `dyDesc` tensors differ.
- ▶ The strides `nStride`, `cStride`, `hStride`, `wStride` of the `yDesc` and `dyDesc` tensors differ.
- ▶ The dimensions `n`, `c`, `h`, `w` of the `dxDesc` and `dxDesc` tensors differ.
- ▶ The strides `nStride`, `cStride`, `hStride`, `wStride` of the `xDesc` and `dxDesc` tensors differ.
- ▶ The datatype of the four tensors differ.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

- ▶ The `wStride` of input tensor or output tensor is not 1.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**4.1.19. cudnnSoftmaxBackward()**

This routine computes the gradient of the softmax function.

```

cudaStatus_t cudnnSoftmaxBackward(
    cudaHandle_t          handle,
    cudaSoftmaxAlgorithm_t algorithm,
    cudaSoftmaxMode_t     mode,
    const void            *alpha,
    const cudaTensorDescriptor_t yDesc,
    const void            *yData,
    const cudaTensorDescriptor_t dyDesc,
    const void            *dy,
    const void            *beta,
    const cudaTensorDescriptor_t dxDesc,
    void                  *dx)
    
```

In-place operation is allowed for this routine; meaning, `dy` and `dx` pointers may be equal. However, this requires `dyDesc` and `dxDesc` descriptors to be identical (particularly, the strides of the input and output must match for in-place operation to be allowed).

All tensor formats are supported for all modes and algorithms with 4 and 5D tensors. Performance is expected to be highest with NCHW fully-packed tensors. For more than 5 dimensions tensors must be packed in their spatial dimensions.

## Data Types

This function supports the following data types:

- ▶ CUDNN\_DATA\_FLOAT
- ▶ CUDNN\_DATA\_DOUBLE
- ▶ CUDNN\_DATA\_HALF
- ▶ CUDNN\_DATA\_BFLOAT16

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### algorithm

*Input.* Enumerant to specify the softmax algorithm.

### mode

*Input.* Enumerant to specify the softmax mode.

### alpha, beta

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

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

For more information, refer to [Scaling Parameters](#).

### yDesc

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

### y

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

### dyDesc

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

### dy

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

### dxDesc

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

### dx

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The function launched successfully.



**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions `n`, `c`, `h`, `w` of the `yDesc`, `dyDesc` and `dxDesc` tensors differ.
- ▶ The strides `nStride`, `cStride`, `hStride`, `wStride` of the `yDesc` and `dyDesc` tensors differ.
- ▶ The `datatype` of the three tensors differs.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 4.1.20. cudnnSpatialTfGridGeneratorBackward()

This function computes the gradient of a grid generation operation.

```

cudnnStatus_t cudnnSpatialTfGridGeneratorBackward(
    cudnnHandle_t          handle,
    const cudnnSpatialTransformerDescriptor_t stDesc,
    const void             *dgrid,
    void                   *dtheta)
    
```

Only 2d transformation is supported.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**stDesc**

*Input.* Previously created spatial transformer descriptor object.

**dgrid**

*Input.* Data pointer to GPU memory contains the input differential data.

**dtheta**

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

### Returns

**CUDNN\_STATUS\_SUCCESS**

The call was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is NULL.
- ▶ One of the parameters `dgrid` or `dtheta` is NULL.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

- ▶ The dimension of the transformed tensor specified in `stDesc > 4`.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 4.1.21. cudnnSpatialTfSamplerBackward()

This function computes the gradient of a sampling operation.

```

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)
    
```

Only 2d transformation is supported.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**stDesc**

*Input.* Previously created spatial transformer descriptor object.

**alpha, beta**

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

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

For more information, refer to [Scaling Parameters](#).

**xDesc**

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

**x**

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

**dxDesc**

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

**dx**

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

**alphaDgrid, betaDgrid**

*Input.* Pointers to scaling factors (in host memory) used to blend the gradient outputs `dgrid` with prior value in the destination pointer as follows:

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

For more information, refer to [Scaling Parameters](#).

**dyDesc**

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

**dy**

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

**grid**

*Input.* A grid of coordinates generated by [cudnnSpatialTfGridGeneratorForward\(\)](#).

**dgrid**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The call was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is NULL.
- ▶ One of the parameters `x`, `dx`, `y`, `dy`, `grid`, `dgrid` is NULL.
- ▶ The dimension of `dy` differs from those specified in `stDesc`.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

- ▶ The dimension of transformed tensor > 4.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

---

# Chapter 5. `cuda_cnn_infer.so` Library

This entity contains all routines related to convolutional neural networks needed at inference time. The `cuda_cnn_infer` library depends on `cuda_ops_infer`.

For the backend data and descriptor types, refer to the [cuDNN Backend API](#) section.

## 5.1. Data Type References

These are the data type references in the `cuda_cnn_infer.so` library.

### 5.1.1. Pointer To Opaque Struct Types

These are the pointers to the opaque struct types in the `cuda_cnn_infer.so` library.

#### 5.1.1.1. `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.

### 5.1.2. Struct Types

These are the struct types in the `cuda_cnn_infer.so` library.

#### 5.1.2.1. `cudaConvolutionBwdDataAlgoPerf_t`

`cudaConvolutionBwdDataAlgoPerf_t` is a structure containing performance results returned by [`cudaFindConvolutionBackwardDataAlgorithm\(\)`](#) or heuristic results returned by [`cudaGetConvolutionBackwardDataAlgorithm\_v7\(\)`](#).

#### Data Members

**`cudaConvolutionBwdDataAlgo_t algo`**

The algorithm runs to obtain the associated performance metrics.

**cudaStatus\_t status**

If any error occurs during the workspace allocation or timing of [cudaConvolutionBackwardData\(\)](#), this status will represent that error. Otherwise, this status will be the return status of [cudaConvolutionBackwardData\(\)](#).

- ▶ CUDNN\_STATUS\_ALLOC\_FAILED if any error occurred during workspace allocation or if the provided workspace is insufficient.
- ▶ CUDNN\_STATUS\_INTERNAL\_ERROR if any error occurred during timing calculations or workspace deallocation.
- ▶ Otherwise, this will be the return status of [cudaConvolutionBackwardData\(\)](#).

**float time**

The execution time of [cudaConvolutionBackwardData\(\)](#) (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.

### 5.1.2.2. 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 runs 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 the provided workspace is insufficient.
- ▶ CUDNN\_STATUS\_INTERNAL\_ERROR if any error occurred during timing calculations or workspace deallocation.
- ▶ Otherwise, this will be the return status of [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.

## 5.1.3. Enumeration Types

These are the enumeration types in the `cuda_cnn_infer.so` library.

### 5.1.3.1. `cudaConvolutionBwdDataAlgo_t`

`cudaConvolutionBwdDataAlgo_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 products without actually explicitly forming the matrix that holds the input tensor data. The sum is done using the atomic add 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 forming the matrix that holds the input tensor data. The results are deterministic.

**CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_FFT**

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

**CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_FFT\_TILING**

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

**CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_WINOGRAD**

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

**CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_WINOGRAD\_NONFUSED**

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

### 5.1.3.2. `cudnnConvolutionBwdFilterAlgo_t`

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

#### Values

##### **CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_0**

This algorithm expresses the convolution as a sum of matrix products without actually explicitly forming the matrix that holds the input tensor data. The sum is done using the atomic add 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 forming the matrix that holds the input tensor data. The results are deterministic.

##### **CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_FFT**

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

##### **CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_3**

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

##### **CUDNN\_CONVOLUTION\_BWD\_FILTER\_WINOGRAD\_NONFUSED**

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

##### **CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_FFT\_TILING**

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

### 5.1.3.3. `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 forming the matrix that holds the input tensor data.

##### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_IMPLICIT\_PRECOMP\_GEMM**

This algorithm expresses convolution as a matrix product without actually explicitly forming the matrix that holds the input tensor data, but still needs some memory

workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data.

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_GEMM**

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

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_DIRECT**

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

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_FFT**

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

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_FFT\_TILING**

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

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD**

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

#### **CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD\_NONFUSED**

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

### 5.1.3.4. `cudnnConvolutionMode_t`

`cudnnConvolutionMode_t` is an enumerated type used by [`cudnnSetConvolution2dDescriptor\(\)`](#) 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.

### 5.1.3.5. `cudnnReorderType_t`

`cudnnReorderType_t` is an enumerated type to set the convolution reordering type. The reordering type can be set by [`cudnnSetConvolutionReorderType\(\)`](#) and its status can be read by [`cudnnGetConvolutionReorderType\(\)`](#).

```
typedef enum {
    CUDNN_DEFAULT_REORDER = 0,
    CUDNN_NO_REORDER      = 1,
```



```
} cudnnReorderType_t;
```

## 5.2. API Functions

These are the API functions in the `cuda_cnn_infer.so` library.

### 5.2.1. `cudaCnnInferVersionCheck()`

This function checks whether the version of the `CnnInfer` subset of the library is consistent with the other sub-libraries.

```
cudaStatus_t cudaCnnInferVersionCheck(void)
```

#### Returns

##### **CUDNN\_STATUS\_SUCCESS**

The version is consistent with other sub-libraries.

##### **CUDNN\_STATUS\_VERSION\_MISMATCH**

The version of `CnnInfer` is not consistent with other sub-libraries. Users should check the installation and make sure all sub-component versions are consistent.

### 5.2.2. `cudaConvolutionBackwardData()`

This function computes the convolution data 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 `dx`. Scaling factors `alpha` and `beta` can be used to scale the computed result or accumulate with the current `dx`.

```
cudaStatus_t cudaConvolutionBackwardData(
    cudaHandle_t          handle,
    const void           *alpha,
    const cudaFilterDescriptor_t wDesc,
    const void           *w,
    const cudaTensorDescriptor_t dyDesc,
    const void           *dy,
    const cudaConvolutionDescriptor_t convDesc,
    cudaConvolutionBwdDataAlgo_t algo,
    void                 *workSpace,
    size_t               workSpaceSizeInBytes,
    const void           *beta,
    const cudaTensorDescriptor_t dxDesc,
    void                 *dx)
```

#### Parameters

##### **handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [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
```

For more information, refer to [Scaling Parameters](#).

**wDesc**

*Input.* Handle to a previously initialized filter descriptor. For more information, refer to [cudaFilterDescriptor\\_t](#).

**w**

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

**dyDesc**

*Input.* Handle to the previously initialized input differential tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**dy**

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

**convDesc**

*Input.* Previously initialized convolution descriptor. For more information, refer to [cudaConvolutionDescriptor\\_t](#).

**algo**

*Input.* Enumerant that specifies which backward data convolution algorithm should be used to compute the results. For more information, refer to [cudaConvolutionBwdDataAlgo\\_t](#).

**workspace**

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

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**dxDesc**

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

**dx**

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

### Supported configurations


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

Table 19. Supported Configurations for `cudaConvolutionBackwardData()`

Data Type Configurations	wDesc, dyDesc and dxDesc Data Type	convDesc Data Type
TRUE_HALF_CONFIG (only supported on architectures	CUDNN_DATA_HALF	CUDNN_DATA_HALF

Data Type Configurations	wDesc, dyDesc and dxDesc Data Type	convDesc Data Type
with true FP16 support, meaning, compute capability 5.3 and later)		
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE

### Supported algorithms

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

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

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

- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_0 (**\_ALGO\_0**)
- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_1 (**\_ALGO\_1**)
- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_FFT (**\_FFT**)
- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_FFT\_TILING (**\_FFT\_TILING**)
- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_WINOGRAD (**\_WINOGRAD**)
- ▶ CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_WINOGRAD\_NONFUSED (**\_WINOGRAD\_NONFUSED**)
- ▶ CUDNN\_TENSOR\_NCHW (**\_NCHW**)
- ▶ CUDNN\_TENSOR\_NHWC (**\_NHWC**)
- ▶ CUDNN\_TENSOR\_NCHW\_VECT\_C (**\_NCHW\_VECT\_C**)

Table 20. Supported Algorithms for `cudaConvolutionBackwardData()` 2D Convolutions: `wDesc: _NHWC`

Filter descriptor <code>wDesc: _NHWC</code> (refer to <a href="#">cudaTensorFormat_t</a> )					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
<code>_ALGO_0</code> <code>_ALGO_1</code>		NHWC HWC-packed	NHWC HWC-packed	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG	

Table 21. Supported Algorithms for `cudaConvolutionBackwardData()` 2D Convolutions: `wDesc: _NCHW`

Filter descriptor <code>wDesc: _NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
<code>_ALGO_0</code>	No	NCHW CHW-packed	All except <code>_NCHW_VECT_C</code> .	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: greater than 0 for all dimensions convDesc Group Count Support: Greater than 0
<code>_ALGO_1</code>	Yes	NCHW CHW-packed	All except <code>_NCHW_VECT_C</code> .	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: greater than 0 for all dimensions convDesc Group Count Support: Greater than 0

Filter descriptor `wDesc`: `_NCHW`.

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

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

Filter descriptor <code>wDesc: _NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
					stride can be 1 or 2 <code>wDesc</code> filter height must be greater than <code>convDesc</code> zero-padding height <code>wDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width
<code>_WINOGRAD</code>	Yes	NCHW CHW-packed	All except <code>_NCHW_VECT_C</code> .	PSEUDO_HALF_CONFIG FLOAT_CONFIG	Dilation: 1 for all dimensions <code>convDesc</code> Group Count Support: Greater than 0 <code>convDesc</code> vertical and horizontal filter stride must equal 1 <code>wDesc</code> filter height must be 3 <code>wDesc</code> filter width must be 3
<code>_WINOGRAD_NONP</code>	Yes	NCHW CHW-packed	All except <code>_NCHW_VECT_C</code> .	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG	Dilation: 1 for all dimensions

Filter descriptor $wDesc$ : <code>_NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for $dyDesc$	Tensor Formats Supported for $dxDesc$	Data Type Configuration Supported	Important
				<p><code>FLOAT_CONFIG</code></p> <p><code>convDesc</code></p> <p>Group Count Support: Greater than 0</p> <p><code>convDesc</code> vertical and horizontal filter stride must equal 1</p> <p><math>wDesc</math> filter (height, width) must be (3,3) or (5,5)</p> <p>If <math>wDesc</math> filter (height, width) is (5,5) then the data type config <code>TRUE_HALF_CONFIG</code> is not supported</p>	

Table 22. Supported Algorithms for `cudaConvolutionBackwardData()` 3D Convolutions:  $wDesc$ : `_NCHW`

Filter descriptor $wDesc$ : <code>_NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for $dyDesc$	Tensor Formats Supported for $dxDesc$	Data Type Configuration Supported	Important
<code>_ALGO_0</code>	Yes	NCDHW CDHW-packed	All except <code>_NCDHW_VECT_C</code>	<p><code>PSEUDO_HALF_CONFIG</code></p> <p><code>PSEUDO_BFLOAT16_CONFIG</code></p> <p><code>FLOAT_CONFIG</code></p>	<p>Dilation: greater than 0 for all dimensions</p>



Filter descriptor $wDesc$ : <code>_NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for $dyDesc$	Tensor Formats Supported for $dxDesc$	Data Type Configuration Supported	Important
				DOUBLE_CONFIG	convDesc Group Count Support: Greater than 0
<code>_ALGO_1</code>	Yes	NCDHW CDHW-packed	NCDHW CDHW-packed	TRUE_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0
<code>_FFT_TILING</code>	Yes	NCDHW CDHW-packed	NCDHW DHW-packed	PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0  $wDesc$ filter height must equal 16 or less  $wDesc$ filter width must equal 16 or less  $wDesc$ filter depth must equal 16 or less  convDesc must have all filter strides equal to 1  $wDesc$ filter height must

Filter descriptor <code>wDesc: _NCHW</code> .					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
					be greater than <code>convDesc</code> zero-padding height <code>wDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width <code>wDesc</code> filter depth must be greater than <code>convDesc</code> zero-padding width

Table 23. Supported Algorithms for `cudaConvolutionBackwardData()` 3D Convolutions: `wDesc: _NHWC`

Filter descriptor <code>wDesc: _NHWC</code>					
Algo Name (3D Convolutions)	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
<code>_ALGO_1</code>	Yes	NDHWC DHWC-packed	NDHWC DHWC-packed	<code>TRUE_HALF_CONFIG</code> <code>PSEUDO_HALF_CONFIG</code> <code>PESUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code>	Dilation: Greater than 0 for all dimensions <code>convDesc</code> 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, dyDesc, wDesc, convDesc, dxDesc, dy, w, dx, alpha, and beta
- ▶ wDesc and dyDesc have a non-matching number of dimensions
- ▶ wDesc and dxDesc have a non-matching number of dimensions
- ▶ wDesc has fewer than three number of dimensions
- ▶ wDesc, dxDesc, and dyDesc have a non-matching data type.
- ▶ wDesc and dxDesc have a non-matching number of input feature maps per image (or group in case of grouped convolutions).
- ▶ dyDesc spatial sizes do not match with the expected size as determined by `cudaGetConvolutionNdForwardOutputDim`

### CUDNN\_STATUS\_NOT\_SUPPORTED

At least one of the following conditions are met:

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

### CUDNN\_STATUS\_MAPPING\_ERROR

An error occurs during the texture binding of texture object creation associated with the filter data or the input differential tensor data.

### CUDNN\_STATUS\_EXECUTION\_FAILED

The function failed to launch on the GPU.

## 5.2.3. `cudaConvolutionBiasActivationForward()`

This function applies a bias and then an activation to the convolutions or cross-correlations of `cudaConvolutionForward()`, returning results in `y`. The full computation follows the equation  $y = \text{act}(\alpha_1 * \text{conv}(x) + \alpha_2 * z + \text{bias})$ .

```

cudaStatus_t cudaConvolutionBiasActivationForward(
    cudaHandle_t          handle,
    const void            *alpha1,
    const cudaTensorDescriptor_t xDesc,
    const void            *x,
    const cudaFilterDescriptor_t wDesc,
    const void            *w,
    const cudaConvolutionDescriptor_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)

```

The routine [cudnnGetConvolution2dForwardOutputDim\(\)](#) or [cudnnGetConvolutionNdForwardOutputDim\(\)](#) can be used to determine the proper dimensions of the output tensor descriptor `yDesc` with respect to `xDesc`, `convDesc`, and `wDesc`.

Only the `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algo is enabled with `CUDNN_ACTIVATION_IDENTITY`. In other words, in the [cudnnActivationDescriptor\\_t](#) structure of the input `activationDesc`, if the mode of the [cudnnActivationMode\\_t](#) field is set to the enum value `CUDNN_ACTIVATION_IDENTITY`, then the input [cudnnConvolutionFwdAlgo\\_t](#) of this function [cudnnConvolutionBiasActivationForward\(\)](#) must be set to the enum value `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`. For more information, refer to [cudnnSetActivationDescriptor\(\)](#).

Device pointer `z` and `y` may be pointing to the same buffer, however, `x` cannot point to the same buffer as `z` or `y`.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

### alpha1, alpha2

*Input.* Pointers to scaling factors (in host memory) used to blend the computation result of convolution with `z` and bias as follows:

```
y = act ( alpha1 * conv(x) + alpha2 * z + bias )
```

For more information, refer to [Scaling Parameters](#).

### xDesc

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudnnTensorDescriptor\\_t](#).

### x

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

### wDesc

*Input.* Handle to a previously initialized filter descriptor. For more information, refer to [cudnnFilterDescriptor\\_t](#).

### w

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

**convDesc**

*Input.* Previously initialized convolution descriptor. For more information, refer to [cudaConvolutionDescriptor\\_t](#).

**algo**

*Input.* Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, refer to [cudaConvolutionFwdAlgo\\_t](#).

**workSpace**

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

**workSpaceSizeInBytes**

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

**zDesc**

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

**z**

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

**biasDesc**

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

**bias**

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

**activationDesc**

*Input.* Handle to a previously initialized activation descriptor. For more information, refer to [cudaActivationDescriptor\\_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 [cudaConvolutionForward\(\)](#). The following table specifies the supported combinations of data types for `x`, `y`, `z`, `bias`, and `alpha1/alpha2`.

Table 24. Supported Combinations of Data Types (X = CUDNN\_DATA) for `cudaConvolutionBiasActivationForward()`

<b>x</b>	<b>w</b>	<b>convDesc</b>	<b>y and z</b>	<b>bias</b>	<b>alpha1/ alpha2</b>
X_DOUBLE	X_DOUBLE	X_DOUBLE	X_DOUBLE	X_DOUBLE	X_DOUBLE
X_FLOAT	X_FLOAT	X_FLOAT	X_FLOAT	X_FLOAT	X_FLOAT
X_HALF	X_HALF	X_FLOAT	X_HALF	X_HALF	X_FLOAT

<b>x</b>	<b>w</b>	<b>convDesc</b>	<b>y and z</b>	<b>bias</b>	<b>alpha1/ alpha2</b>
X_BFLOAT16	X_BFLOAT16	X_FLOAT	X_BFLOAT16	X_BFLOAT16	X_FLOAT
X_INT8	X_INT8	X_INT32	X_INT8	X_FLOAT	X_FLOAT
X_INT8	X_INT8	X_INT32	X_FLOAT	X_FLOAT	X_FLOAT
X_INT8x4	X_INT8x4	X_INT32	X_INT8x4	X_FLOAT	X_FLOAT
X_INT8x4	X_INT8x4	X_INT32	X_FLOAT	X_FLOAT	X_FLOAT
X_UINT8	X_INT8	X_INT32	X_INT8	X_FLOAT	X_FLOAT
X_UINT8	X_INT8	X_INT32	X_FLOAT	X_FLOAT	X_FLOAT
X_UINT8x4	X_INT8x4	X_INT32	X_INT8x4	X_FLOAT	X_FLOAT
X_UINT8x4	X_INT8x4	X_INT32	X_FLOAT	X_FLOAT	X_FLOAT
X_INT8x32	X_INT8x32	X_INT32	X_INT8x32	X_FLOAT	X_FLOAT

## Returns

In addition to the error values listed by the documentation of [cudaConvolutionForward\(\)](#), the possible error values returned by this function and their meanings are listed below.

### **CUDNN\_STATUS\_SUCCESS**

The operation was launched successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ At least one of the following is NULL: handle, xDesc, wDesc, convDesc, yDesc, zDesc, biasDesc, activationDesc, xData, wData, yData, zData, bias, alpha1, and alpha2.
- ▶ The number of dimensions of xDesc, wDesc, yDesc, and zDesc is not equal to the array length of convDesc + 2.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

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

- ▶ The mode of activationDesc is not 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 first dimension of biasDesc is not equal to one.
- ▶ The second dimension of biasDesc and the first dimension of filterDesc are not equal.
- ▶ The data type of biasDesc does not correspond to the data type of yDesc as listed in the above data types table.

- ▶ `zDesc` and `destDesc` do not match.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 5.2.4. `cudaConvolutionForward()`

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.

```

cudaStatus_t cudaConvolutionForward(
    cudaHandle_t          handle,
    const void*          *alpha,
    const cudaTensorDescriptor_t xDesc,
    const void*          *x,
    const cudaFilterDescriptor_t wDesc,
    const void*          *w,
    const cudaConvolutionDescriptor_t convDesc,
    cudaConvolutionFwdAlgo_t algo,
    void*                *workSpace,
    size_t               workSpaceSizeInBytes,
    const void*          *beta,
    const cudaTensorDescriptor_t yDesc,
    void*                *y)
    
```

The routine [cudaGetConvolution2dForwardOutputDim\(\)](#) or [cudaGetConvolutionNdForwardOutputDim\(\)](#) can be used to determine the proper dimensions of the output tensor descriptor `yDesc` with respect to `xDesc`, `convDesc`, and `wDesc`.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [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
```

For more information, refer to [Scaling Parameters](#).

**xDesc**

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**x**

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

**wDesc**

*Input.* Handle to a previously initialized filter descriptor. For more information, refer to [cudaFilterDescriptor\\_t](#).

**w**

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

**convDesc**

*Input.* Previously initialized convolution descriptor. For more information, refer to [cudaConvolutionDescriptor\\_t](#).

**algo**

*Input.* Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, refer to [cudaConvolutionFwdAlgo\\_t](#).

**workSpace**

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

**workSpaceSizeInBytes**

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

**yDesc**

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

**y**

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

### Supported configurations

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

Table 25. Supported Configurations for `cudaConvolutionForward()`

Data Type Configurations	<code>xDesc</code> and <code>wDesc</code>	<code>convDesc</code>	<code>yDesc</code>
<code>TRUE_HALF_CONFIG</code> (only supported on architectures with true FP16 support, meaning, compute capability 5.3 and later)	<code>CUDNN_DATA_HALF</code>	<code>CUDNN_DATA_HALF</code>	<code>CUDNN_DATA_HALF</code>
<code>PSEUDO_HALF_CONFIG</code>	<code>CUDNN_DATA_HALF</code>	<code>CUDNN_DATA_FLOAT</code>	<code>CUDNN_DATA_HALF</code>
<code>PSEUDO_BFLOAT16_CONFIG</code> (only support on architecture with bfloat16 support, meaning, compute capability 8.0 and later)	<code>CUDNN_DATA_BFLOAT16</code>	<code>CUDNN_DATA_FLOAT</code>	<code>CUDNN_DATA_BFLOAT16</code>



<b>Data Type Configurations</b>	<b>xDesc and wDesc</b>	<b>convDesc</b>	<b>yDesc</b>
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, meaning, compute capability 6.1 and later)	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_INT8
INT8_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
INT8x4_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_INT8x4
INT8x4_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
UINT8_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	xDesc: CUDNN_DATA_UINT8  wDesc: CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_INT8
UINT8x4_CONFIG (only supported	xDesc: CUDNN_DATA_UINT8x4	CUDNN_DATA_INT32	CUDNN_DATA_INT8x4

Data Type Configurations	xDesc and wDesc	convDesc	yDesc
on architectures with DP4A support, meaning, compute capability 6.1 and later)	wDesc: CUDNN_DATA_INT8x4		
UINT8_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	xDesc: CUDNN_DATA_UINT8  wDesc: CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
UINT8x4_EXT_CONFIG (only supported on architectures with DP4A support, meaning, compute capability 6.1 and later)	xDesc: CUDNN_DATA_UINT8x4  wDesc: CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
INT8x32_CONFIG (only supported on architectures with IMMA support, meaning compute capability 7.5 and later)	CUDNN_DATA_INT8x32	CUDNN_DATA_INT32	CUDNN_DATA_INT8x32

### Supported algorithms



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

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

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

- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_IMPLICIT\_GEMM (**\_IMPLICIT\_GEMM**)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_IMPLICIT\_PRECOMP\_GEMM (**\_IMPLICIT\_PRECOMP\_GEMM**)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_GEMM (**\_GEMM**)

- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_DIRECT (\_DIRECT)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_FFT (\_FFT)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_FFT\_TILING (\_FFT\_TILING)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD (\_WINOGRAD)
- ▶ CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD\_NONFUSED (\_WINOGRAD\_NONFUSED)
- ▶ CUDNN\_TENSOR\_NCHW (\_NCHW)
- ▶ CUDNN\_TENSOR\_NHWC (\_NHWC)
- ▶ CUDNN\_TENSOR\_NCHW\_VECT\_C (\_NCHW\_VECT\_C)

Table 26. Supported Algorithms for cudnnConvolutionForward() 2D Convolutions: wDesc: \_NCHW

Filter descriptor wDesc: _NCHW (refer to <a href="#">cudnnTensorFormat_t</a> )				
convDesc Group count support: Greater than 0, for all algos.				
Algo Name	Tensor Formats Supported for xDesc	Tensor Formats Supported for yDesc	Data Type Configurations Supported	Important
_IMPLICIT_GEMM	All except _NCHW_VECT_C.	All except _NCHW_VECT_C.	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: Greater than 0 for all dimensions
_IMPLICIT_PRECOMPUTED	All except _NCHW_VECT_C.	All except _NCHW_VECT_C.	TRUE_HALF_CONFIG PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: 1 for all dimensions
_GEMM	All except _NCHW_VECT_C.	All except _NCHW_VECT_C.	PSEUDO_HALF_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: 1 for all dimensions
_FFT	NCHW HW-packed	NCHW HW-packed	PSEUDO_HALF_CONFIG FLOAT_CONFIG	Dilation: 1 for all dimensions  xDesc feature map height + 2 * convDesc zero-

Filter descriptor <code>wDesc</code> : <code>_NCHW</code> (refer to <a href="#"><code>cudaTensorFormat_t</code></a> )				
<code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name	Tensor Formats Supported for <code>xDesc</code>	Tensor Formats Supported for <code>yDesc</code>	Data Type Configurations Supported	Important
				padding height must equal 256 or less  <code>xDesc</code> feature map width + 2 * <code>convDesc</code> zero-padding width must equal 256 or less  <code>convDesc</code> vertical and horizontal filter stride must equal 1  <code>wDesc</code> filter height must be greater than <code>convDesc</code> zero-padding height  <code>wDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width
<code>_FFT_TILING</code>			<code>PSEUDO_HALF_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code> is also supported when the task can be handled by 1D FFT, meaning, one of the filter dimensions,	Dilation: 1 for all dimensions  When neither of <code>wDesc</code> filter dimension is 1, the filter width and height must not be larger than 32  When either of <code>wDesc</code> filter

Filter descriptor <code>wDesc: _NCHW</code> (refer to <a href="#"><code>cudaTensorFormat_t</code></a> )				
<code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name	Tensor Formats Supported for <code>xDesc</code>	Tensor Formats Supported for <code>yDesc</code>	Data Type Configurations Supported	Important
			width or height is 1.	dimension is 1, the largest filter dimension should not exceed 256  <code>convDesc</code> vertical and horizontal filter stride must equal 1 when either the filter width or filter height is 1, otherwise the stride can be a 1 or 2  <code>wDesc</code> filter height must be greater than <code>convDesc</code> zero-padding height  <code>wDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width
<code>_WINOGRAD</code>	All except <code>_NCHW_VECT</code>	All except <code>_NCHW_VECT</code>	<code>PSEUDO_HALF_CONFIG</code> <code>C_FLOAT_CONFIG</code>	Dilation: 1 for all dimensions  <code>convDesc</code> vertical and horizontal filter stride must equal 1  <code>wDesc</code> filter height must be 3  <code>wDesc</code> filter width must be 3

Filter descriptor wDesc: <code>_NCHW</code> (refer to <code>cudaTensorFormat_t</code> )				
convDesc Group count support: Greater than 0, for all algos.				
Algo Name	Tensor Formats Supported for xDesc	Tensor Formats Supported for yDesc	Data Type Configurations Supported	Important
<code>_WINOGRAD_NONFUSED</code>			<code>TRUE_HALF_CONFIG</code> <code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code>	Dilation: 1 for all dimensions convDesc vertical and horizontal filter stride must equal 1 wDesc filter (height, width) must be (3,3) or (5,5) If wDesc filter (height, width) is (5,5), then data type config <code>TRUE_HALF_CONFIG</code> is not supported.
<code>_DIRECT</code>	Currently not implemented in cuDNN.			

Table 27. Supported Algorithms for `cudaConvolutionForward()` 2D Convolutions: wDesc: `_NCHWC`

Filter descriptor wDesc: <code>_NCHWC</code>				
convDesc Group count support: Greater than 0.				
Algo Name	xDesc	yDesc	Data Type Configurations Supported	Important
<code>_IMPLICIT_GEMM</code> <code>_IMPLICIT_PRECOMP_GEMM</code>	<code>_NCHW_VECT_C</code>	<code>_NCHW_VECT_C</code>	<code>INT8x4_CONFIG</code> <code>UINT8x4_CONFIG</code>	Dilation: 1 for all dimensions
<code>_IMPLICIT_PRECOMP_GEMM</code>	<code>_NCHW_VECT_C</code>	<code>_NCHW_VECT_C</code>	<code>INT8x32_CONFIG</code>	Dilation: 1 for all dimensions Requires compute

Filter descriptor wDesc: <code>_NCHWC</code>				
convDesc Group count support: Greater than 0.				
Algo Name	xDesc	yDesc	Data Type Configurations Supported	Important
				capability 7.2 or above.

Table 28. Supported Algorithms for `cudaConvolutionForward()` 2D Convolutions: wDesc: `_NHWC`

Filter descriptor wDesc: <code>_NHWC</code>				
convDesc Group count support: Greater than 0.				
Algo Name	xDesc	yDesc	Data Type Configurations Supported	Important
<code>_IMPLICIT_GEMM</code> <code>_IMPLICIT_PRECOMP_GEMM</code>	NHWC fully-packed	NHWC fully-packed	<code>INT8_CONFIG</code> <code>INT8_EXT_CONFIG</code> <code>UINT8_CONFIG</code> <code>UINT8_EXT_CONFIG</code>	Dilation: 1 for all dimensions  Input and output feature maps must be a multiple of 4. Output features maps can be non-multiple in the case of <code>INT8_EXT_CONFIG</code> or <code>UINT8_EXT_CONFIG</code> .
<code>_IMPLICIT_GEMM</code> <code>_IMPLICIT_PRECOMP_GEMM</code>	NHWC HWC-packed.	NHWC HWC-packed.  NCHW CHW-packed	<code>TRUE_HALF_CONFIG</code> <code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code>	

Table 29. Supported Algorithms for `cudaConvolutionForward()` 3D Convolutions: `wDesc: _NCHW`

Filter descriptor <code>wDesc: _NCHW</code>				
<code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name	<code>xDesc</code>	<code>yDesc</code>	Data Type Configurations Supported	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 0 for all dimensions
<code>_IMPLICIT_PRECOMP_GEMM</code>			<code>PSEUDO_BFLOAT16_CONFIG</code>	
<code>_FFT_TILING</code>	NCDHW DHW-packed	NCDHW DHW-packed	<code>DOUBLE_CONFIG</code>	Dilation: Greater than 0 for all dimensions  Dilation: 1 for all dimensions  <code>wDesc</code> filter height must equal 16 or less  <code>wDesc</code> filter width must equal 16 or less  <code>wDesc</code> filter depth must equal 16 or less  <code>convDesc</code> must have all filter strides equal to 1  <code>wDesc</code> filter height must be greater than <code>convDesc</code> zero-padding height  <code>wDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width



Filter descriptor <code>wDesc</code> : <code>_NCHW</code>				
<code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name	<code>xDesc</code>	<code>yDesc</code>	Data Type Configurations Supported	Important
				<code>wDesc</code> filter depth must be greater than <code>convDesc</code> zero-padding depth

Table 30. Supported Algorithms for `cudaConvolutionForward()` 3D Convolutions: `wDesc`: `_NHWC`

Filter descriptor <code>wDesc</code> : <code>_NHWC</code>				
<code>convDesc</code> Group count support: Greater than 0, for all algos.				
Algo Name	<code>xDesc</code>	<code>yDesc</code>	Data Type Configurations Supported	Important
<code>_IMPLICIT_PRECOMPUTED</code>	<code>NHWC</code> DHWC-packed	<code>NHWC</code> DHWC-packed	<code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code>	Dilation: Greater than 0 for all dimensions

Note: 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`, and `beta`
- ▶ `xDesc` and `yDesc` have a non-matching number of dimensions
- ▶ `xDesc` and `wDesc` have a non-matching number of dimensions
- ▶ `xDesc` has fewer than three number of dimensions
- ▶ `xDesc`'s number of dimensions is not equal to `convDesc` array length + 2

- ▶ `xDesc` and `wDesc` have a non-matching number of input feature maps per image (or group in case of grouped convolutions)
- ▶ `yDesc` or `wDesc` indicate an output channel count that isn't a multiple of group count (if group count has been set in `convDesc`).
- ▶ `xDesc`, `wDesc`, and `yDesc` have a non-matching data type
- ▶ For some spatial dimension, `wDesc` has a spatial size that is larger than the input spatial size (including zero-padding size)

**CUDNN\_STATUS\_NOT\_SUPPORTED**

At least one of the following conditions are met:

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

**CUDNN\_STATUS\_MAPPING\_ERROR**

An error occurs during the texture object creation associated with the filter data.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 5.2.5. `cudaCreateConvolutionDescriptor()`

This function creates a convolution descriptor object by allocating the memory needed to hold its opaque structure. For more information, refer to [`cudaConvolutionDescriptor\_t`](#).

```
cudaStatus_t cudaCreateConvolutionDescriptor(
    cudaConvolutionDescriptor_t *convDesc)
```

### Returns

**CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

## 5.2.6. `cudaDestroyConvolutionDescriptor()`

This function destroys a previously created convolution descriptor object.

```
cudaStatus_t cudaDestroyConvolutionDescriptor(
    cudaConvolutionDescriptor_t convDesc)
```

## Returns

### CUDNN\_STATUS\_SUCCESS

The descriptor was destroyed successfully.

## 5.2.7. cudnnFindConvolutionBackwardDataAlgorithm()

This function attempts all algorithms available for [cudnnConvolutionBackwardData\(\)](#). It will attempt both the provided `convDesc` `mathType` and `CUDNN_DEFAULT_MATH` (assuming the two differ).

```

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)
    
```

Algorithms without the `CUDNN_TENSOR_OP_MATH` availability will only be tried with `CUDNN_DEFAULT_MATH`, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of [cudnnConvolutionBwdDataAlgoPerf\\_t](#). These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API [cudnnGetConvolutionBackwardDataAlgorithmMaxCount\(\)](#).



### Note:

- ▶ This function is host blocking.
- ▶ It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### wDesc

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

### dyDesc

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

### convDesc

*Input.* Previously initialized convolution descriptor.

### dxDesc

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

### requestedAlgoCount

*Input.* The maximum number of elements to be stored in `perfResults`.

**returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

**perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is not allocated properly.
- ▶ `wDesc`, `dyDesc`, or `dxDesc` is not allocated properly.
- ▶ `wDesc`, `dyDesc`, or `dxDesc` has fewer than 1 dimension.
- ▶ Either `returnedCount` or `perfResults` is NIL.
- ▶ `requestedCount` is less than 1.

**CUDNN\_STATUS\_ALLOC\_FAILED**

This function was unable to allocate memory to store sample input, filters and output.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

At least one of the following conditions are met:

- ▶ The function was unable to allocate necessary timing objects.
- ▶ The function was unable to deallocate necessary timing objects.
- ▶ The function was unable to deallocate sample input, filters and output.

**5.2.8. cudnnFindConvolutionBackwardDataAlgorithmEx (**

This function attempts all algorithms available for `cudnnConvolutionBackwardData()`. It will attempt both the provided `convDescv` `mathType` and `CUDNN_DEFAULT_MATH` (assuming the two differ).

```

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)
    
```

Algorithms without the `CUDNN_TENSOR_OP_MATH` availability will only be tried with `CUDNN_DEFAULT_MATH`, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of `cudaConvolutionBwdDataAlgoPerf_t`. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudaGetConvolutionBackwardDataAlgorithmMaxCount()`.

 Note: 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 is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a `workSpace` of 0 bytes.

**workSpaceSizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ handle is not allocated properly.
- ▶ wDesc, dyDesc, or dxDesc is not allocated properly.
- ▶ wDesc, dyDesc, or dxDesc has fewer than 1 dimension.
- ▶ w, dy, or dx is NIL.
- ▶ Either returnedCount or perfResults is NIL.
- ▶ requestedCount is less than 1.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

At least one of the following conditions are met:

- ▶ The function was unable to allocate necessary timing objects.
- ▶ The function was unable to deallocate necessary timing objects.
- ▶ The function was unable to deallocate sample input, filters and output.

## 5.2.9. cudnnFindConvolutionForwardAlgorithm()


This function attempts all algorithms available for [cudnnConvolutionForward\(\)](#). It will attempt both the provided convDesc mathType and CUDNN\_DEFAULT\_MATH (assuming the two differ).

```

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)
    
```

Algorithms without the CUDNN\_TENSOR\_OP\_MATH availability will only be tried with CUDNN\_DEFAULT\_MATH, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of [cudnnConvolutionFwdAlgoPerf\\_t](#). These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API [cudnnGetConvolutionForwardAlgorithmMaxCount\(\)](#).

 **Note:**

- ▶ This function is host blocking.
- ▶ It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **xDesc**

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

### **wDesc**

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

### **convDesc**

*Input.* Previously initialized convolution descriptor.

### **yDesc**

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

### **requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

### **returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

### **perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The query was successful.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is not allocated properly.
- ▶ `xDesc`, `wDesc`, or `yDesc` are 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.

## 5.2.10. cudnnFindConvolutionForwardAlgorithmEx()

This function attempts all algorithms available for [cudnnConvolutionForward\(\)](#). It will attempt both the provided `convDesc` `mathType` and `CUDNN_DEFAULT_MATH` (assuming the two differ).

```

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)
    
```

Algorithms without the `CUDNN_TENSOR_OP_MATH` availability will only be tried with `CUDNN_DEFAULT_MATH`, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of [cudnnConvolutionFwdAlgoPerf\\_t](#). These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API [cudnnGetConvolutionForwardAlgorithmMaxCount\(\)](#).



Note: 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 is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a `workSpace` of 0 bytes.

**workSpaceSizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is not allocated properly.
- ▶ `xDesc`, `wDesc`, or `yDesc` are 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.

## 5.2.11. cudnnGetConvolution2dDescriptor()

This function queries a previously initialized 2D convolution descriptor object.

```

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)

```

## Parameters

**convDesc**

*Input.* Handle to a previously created convolution descriptor.

**pad\_h**

*Output.* Zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

**pad\_w**

*Output.* Zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

**u**

*Output.* Vertical filter stride.

**v**

*Output.* Horizontal filter stride.

**dilation\_h**

*Output.* Filter height dilation.

**dilation\_w**

*Output.* Filter width dilation.

**mode**

*Output.* Convolution mode.

**computeType**

*Output.* Compute precision.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The operation was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

The parameter `convDesc` is NIL.

### 5.2.12. `cudaGetConvolution2dForwardOutputDim()`

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.

```

cudaStatus_t cudaGetConvolution2dForwardOutputDim(
    const cudaConvolutionDescriptor_t convDesc,
    const cudaTensorDescriptor_t inputTensorDesc,
    const cudaFilterDescriptor_t filterDesc,
    int *n,
    int *c,
    int *h,
    int *w)

```

Each dimension `h` and `w` of the output images is computed as follows:

```

outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) ) /
convolutionStride;

```



Note: The dimensions provided by this routine must be strictly respected when calling [cudnnConvolutionForward\(\)](#) or [cudnnConvolutionBackwardBias\(\)](#). Providing a smaller or larger output tensor is not supported by the convolution routines.

## Parameters

**convDesc**

*Input.* Handle to a previously created convolution descriptor.

**inputTensorDesc**

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

**filterDesc**

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

**n**

*Output.* Number of output images.

**c**

*Output.* Number of output feature maps per image.

**h**

*Output.* Height of each output feature map.

**w**

*Output.* Width of each output feature map.

## Returns

**CUDNN\_STATUS\_BAD\_PARAM**

One or more of the descriptors has not been created correctly or there is a mismatch between the feature maps of `inputTensorDesc` and `filterDesc`.

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

## 5.2.13. cudnnGetConvolutionBackwardDataAlgorithmMaxC

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.

```

cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithmMaxCount(
    cudnnHandle_t handle,
    int *count)
    
```

## 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.

## 5.2.14. cudnnGetConvolutionBackwardDataAlgorithm\_v7()

This function serves as a heuristic for obtaining the best suited algorithm for [cudnnConvolutionBackwardData\(\)](#) for the given layer specifications. This function will return all algorithms (including CUDNN\_TENSOR\_OP\_MATH and CUDNN\_DEFAULT\_MATH versions of algorithms where CUDNN\_TENSOR\_OP\_MATH may be available) sorted by expected (based on internal heuristic) relative performance with the fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, use [cudnnFindConvolutionBackwardDataAlgorithm\(\)](#). The total number of resulting algorithms can be queried through the returnedAlgoCount variable.

```

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)
    
```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### wDesc

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

### dyDesc

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

### convDesc

*Input.* Previously initialized convolution descriptor.

### dxDesc

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

### requestedAlgoCount

*Input.* The maximum number of elements to be stored in perfResults.

### returnedAlgoCount

*Output.* The number of output elements stored in perfResults.

### perfResults

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

## Returns

### CUDNN\_STATUS\_SUCCESS

The query was successful.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ One of the parameters `handle`, `wDesc`, `dyDesc`, `convDesc`, `dxDesc`, `perfResults`, or `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 filters are different.
- ▶ `requestedAlgoCount` is less than or equal to 0.

## 5.2.15. cudnnGetConvolutionBackwardDataWorkspaceSize

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call `cudnnConvolutionBackwardData()` with the specified algorithm. The workspace allocated will then be passed to the routine `cudnnConvolutionBackwardData()`. The specified algorithm can be the result of the call to `cudnnGetConvolutionBackwardDataAlgorithm_v7()` 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.

```

cudnnStatus_t cudnnGetConvolutionBackwardDataWorkspaceSize(
    cudnnHandle_t          handle,
    const cudnnFilterDescriptor_t    wDesc,
    const cudnnTensorDescriptor_t    dyDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t    dxDesc,
    cudnnConvolutionBwdDataAlgo_t    algo,
    size_t                 *sizeInBytes)
    
```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### wDesc

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

### dyDesc

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

### convDesc

*Input.* Previously initialized convolution descriptor.

### dxDesc

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

### algo

*Input.* Enumerant that specifies the chosen convolution algorithm.

**sizeInBytes**

*Output.* Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified `algo`.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The numbers of feature maps of the input tensor and output tensor differ.
- ▶ The `dataType` of the two tensor descriptors or the filter are different.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

**5.2.16. cudnnGetConvolutionForwardAlgorithmMaxCount()**

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.

```

cudnnStatus_t cudnnGetConvolutionForwardAlgorithmMaxCount(
    cudnnHandle_t handle,
    int *count)
    
```

**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.

**5.2.17. cudnnGetConvolutionForwardAlgorithm\_v7()**

This function serves as a heuristic for obtaining the best suited algorithm for `cudnnConvolutionForward()` for the given layer specifications. This function will return all algorithms (including `CUDNN_TENSOR_OP_MATH` and `CUDNN_DEFAULT_MATH` versions of algorithms where `CUDNN_TENSOR_OP_MATH` may be available) sorted by expected (based on internal heuristic) relative performance with the fastest being

index 0 of `perfResults`. For an exhaustive search for the fastest algorithm, use `cudaFindConvolutionForwardAlgorithm()`. The total number of resulting algorithms can be queried through the `returnedAlgoCount` variable.

```

cudaStatus_t cudaGetConvolutionForwardAlgorithm_v7(
    cudaHandle_t          handle,
    const cudaTensorDescriptor_t  xDesc,
    const cudaFilterDescriptor_t  wDesc,
    const cudaConvolutionDescriptor_t  convDesc,
    const cudaTensorDescriptor_t  yDesc,
    const int             requestedAlgoCount,
    int                  *returnedAlgoCount,
    cudaConvolutionFwdAlgoPerf_t  *perfResults)

```

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **xDesc**

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

### **wDesc**

*Input.* Handle to a previously initialized convolution filter descriptor.

### **convDesc**

*Input.* Previously initialized convolution descriptor.

### **yDesc**

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

### **requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

### **returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

### **perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The query was successful.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the parameters `handle`, `xDesc`, `wDesc`, `convDesc`, `yDesc`, `perfResults`, or `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.

## 5.2.18. cudnnGetConvolutionForwardWorkspaceSize()

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_v7()` 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.

```

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)

```

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**xDesc**

*Input.* Handle to the previously initialized  $x$  tensor descriptor.

**wDesc**

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

**convDesc**

*Input.* Previously initialized convolution descriptor.

**yDesc**

*Input.* Handle to the previously initialized  $y$  tensor descriptor.

**algo**

*Input.* Enumerant that specifies the chosen convolution algorithm.

**sizeInBytes**

*Output.* Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified `algo`.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the parameters `handle`, `xDesc`, `wDesc`, `convDesc`, or `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.

### 5.2.19. `cudaGetConvolutionGroupCount()`

This function returns the group count specified in the given convolution descriptor.

```
cudaStatus_t cudaGetConvolutionGroupCount(
    cudaConvolutionDescriptor_t convDesc,
    int *groupCount)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The group count was returned successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid convolution descriptor was provided.

### 5.2.20. `cudaGetConvolutionMathType()`

This function returns the math type specified in a given convolution descriptor.

```
cudaStatus_t cudaGetConvolutionMathType(
    cudaConvolutionDescriptor_t convDesc,
    cudaMathType_t *mathType)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The math type was returned successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid convolution descriptor was provided.

### 5.2.21. `cudaGetConvolutionNdDescriptor()`

This function queries a previously initialized convolution descriptor object.

```
cudaStatus_t cudaGetConvolutionNdDescriptor(
    const cudaConvolutionDescriptor_t convDesc,
    int arrayLengthRequested,
    int *arrayLength,
    int padA[],
    int filterStrideA[],
    int dilationA[],
    cudaConvolutionMode_t *mode,
    cudaDataType_t *dataType)
```

#### 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 successful.

**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`.

## 5.2.22. `cudaGetConvolutionNdForwardOutputDim()`

This function returns the dimensions of the resulting `Nd` 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.

```

cudaStatus_t cudaGetConvolutionNdForwardOutputDim(
    const cudaConvolutionDescriptor_t convDesc,
    const cudaTensorDescriptor_t inputTensorDesc,
    const cudaFilterDescriptor_t filterDesc,
    int nbDims,
    int tensorOutputDimA[])
    
```

Each dimension of the `(nbDims-2)-D` images of the output tensor is computed as follows:

```

// ...
    
```

```
outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) ) /
convolutionStride;
```



Note: 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.

## 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 exited successfully.

## 5.2.23. cudnnGetConvolutionReorderType()

This function retrieves the convolution reorder type from the given convolution descriptor.

```
cudnnStatus_t cudnnGetConvolutionReorderType(
```

```

    cudnnConvolutionDescriptor_t convDesc,
    cudnnReorderType_t *reorderType);

```

## Parameters

### convDesc

*Input.* The convolution descriptor from which the reorder type should be retrieved.

### reorderType

*Output.* The retrieved reorder type. For more information, refer to [cudnnReorderType\\_t](#).

## Returns

### CUDNN\_STATUS\_BAD\_PARAM

One of the inputs to this function is not valid.

### CUDNN\_STATUS\_SUCCESS

The reorder type is retrieved successfully.

## 5.2.24. cudnnGetFoldedConvBackwardDataDescriptors ()

This function calculates folding descriptors for backward data gradients. It takes as input the data descriptors along with the convolution descriptor and computes the folded data descriptors and the folding transform descriptors. These can then be used to do the actual folding transform.

```

cudnnStatus_t
cudnnGetFoldedConvBackwardDataDescriptors(const cudnnHandle_t handle,
                                          const cudnnFilterDescriptor_t filterDesc,
                                          const cudnnTensorDescriptor_t diffDesc,
                                          const cudnnConvolutionDescriptor_t
convDesc,
                                          const cudnnTensorDescriptor_t gradDesc,
                                          const cudnnTensorFormat_t transformFormat,
                                          cudnnFilterDescriptor_t foldedFilterDesc,
                                          cudnnTensorDescriptor_t paddedDiffDesc,
                                          cudnnConvolutionDescriptor_t
foldedConvDesc,
                                          cudnnTensorDescriptor_t foldedGradDesc,
                                          cudnnTensorTransformDescriptor_t
filterFoldTransDesc,
                                          cudnnTensorTransformDescriptor_t
diffPadTransDesc,
                                          cudnnTensorTransformDescriptor_t
gradFoldTransDesc,
                                          cudnnTensorTransformDescriptor_t
gradUnfoldTransDesc) ;

```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### filterDesc

*Input.* Filter descriptor before folding.

### diffDesc

*Input.* Diff descriptor before folding.

**convDesc**

*Input.* Convolution descriptor before folding.

**gradDesc**

*Input.* Gradient descriptor before folding.

**transformFormat**

*Input.* Transform format for folding.

**foldedFilterDesc**

*Output.* Folded filter descriptor.

**paddedDiffDesc**

*Output.* Padded Diff descriptor.

**foldedConvDesc**

*Output.* Folded convolution descriptor.

**foldedGradDesc**

*Output.* Folded gradient descriptor.

**filterFoldTransDesc**

*Output.* Folding transform descriptor for filter.

**diffPadTransDesc**

*Output.* Folding transform descriptor for Desc.

**gradFoldTransDesc**

*Output.* Folding transform descriptor for gradient.

**gradUnfoldTransDesc**

*Output.* Unfolding transform descriptor for folded gradient.

## Returns

**CUDNN\_STATUS\_SUCCESS**

Folded descriptors were computed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

If any of the input parameters is NULL or if the input tensor has more than 4 dimensions.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

Computing the folded descriptors failed.

## 5.2.25. cudnnIm2Col()

This function constructs the A matrix necessary to perform a forward pass of GEMM convolution.

```

cudnnStatus_t cudnnIm2Col(
    cudnnHandle_t          handle,
    cudnnTensorDescriptor_t srcDesc,
    const void            *srcData,
    cudnnFilterDescriptor_t filterDesc,
    cudnnConvolutionDescriptor_t convDesc,
    void                  *colBuffer)
    
```

This A matrix has a height of `batch_size*y_height*y_width` and width of `input_channels*filter_height*filter_width`, where:

- ▶ `batch_size` is `srcDesc` first dimension
- ▶ `y_height/y_width` are computed from `cudaGetConvolutionNdForwardOutputDim()`
- ▶ `input_channels` is `srcDesc` second dimension (when in NCHW layout)
- ▶ `filter_height/filter_width` are `wDesc` third and fourth dimension

The `A` matrix is stored in format HW fully-packed in GPU memory.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **srcDesc**

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

### **srcData**

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

### **filterDesc**

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

### **convDesc**

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

### **colBuffer**

*Output.* Data pointer to GPU memory storing the output matrix.

## Returns

### **CUDNN\_STATUS\_BAD\_PARAM**

`srcData` or `colBuffer` is NULL.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

Any of `srcDesc`, `filterDesc`, `convDesc` has `dataType` of `CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8` or `CUDNN_DATA_INT8x4` `convDesc` has `groupCount` larger than 1.

### **CUDNN\_STATUS\_EXECUTION\_FAILED**

The CUDA kernel execution was unsuccessful.

### **CUDNN\_STATUS\_SUCCESS**

The output data array is successfully generated.

## 5.2.26. `cudaReorderFilterAndBias()`

This function `cudaReorderFilterAndBias()`, reorders the filter and bias values for tensors with data type `CUDNN_DATA_INT8x32` and tensor format `CUDNN_TENSOR_NCHW_VECT_C`. It can be used to enhance the inference time by separating the reordering operation from convolution. Currently, only 2D filters are supported.

```

cudaStatus_t cudaReorderFilterAndBias(
    cudaHandle_t handle,
    const cudaFilterDescriptor_t filterDesc,
    cudaReorderType_t reorderType,
    const void *filterData,
    void *reorderedFilterData,
    int reorderBias,

```

```
const void *biasData,
void *reorderedBiasData);
```

Filter and bias tensors with data type CUDNN\_DATA\_INT8x32 (also implying tensor format CUDNN\_TENSOR\_NCHW\_VECT\_C) requires permutation of output channel axes in order to take advantage of the Tensor Core IMMA instruction. This is done in every [cudnnConvolutionForward\(\)](#) and [cudnnConvolutionBiasActivationForward\(\)](#) call when the reorder type attribute of the convolution descriptor is set to CUDNN\_DEFAULT\_REORDER. Users can avoid the repeated reordering kernel call by first using this call to reorder the filter and bias tensor and call the convolution forward APIs with reorder type set to CUDNN\_NO\_REORDER.

For example, convolutions in a neural network of multiple layers can require reordering of kernels at every layer, which can take up a significant fraction of the total inference time. Using this function, the reordering can be done one time on the filter and bias data. This is followed by the convolution operations at the multiple layers, which enhance the inference time.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **filterDesc**

*Input.* Descriptor for the kernel dataset.

### **reorderType**

*Input.* Setting to either perform reordering or not. For more information, refer to [cudnnReorderType\\_t](#).

### **filterData**

*Input.* Pointer to the filter (kernel) data location in the device memory.

### **reorderedFilterData**

*Output.* Pointer to the location in the device memory where the reordered filter data will be written to, by this function. This tensor has the same dimensions as `filterData`.

### **reorderBias**

*Input.* If > 0, then reorders the bias data also. If <= 0 then does not perform reordering operations on the bias data.

### **biasData**

*Input.* Pointer to the bias data location in the device memory.

### **reorderedBiasData**

*Output.* Pointer to the location in the device memory where the reordered bias data will be written to, by this function. This tensor has the same dimensions as `biasData`.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

Reordering was successful.

### **CUDNN\_STATUS\_EXECUTION\_FAILED**

Either the reordering of the filter data or of the bias data failed.

**CUDNN\_STATUS\_BAD\_PARAM**

The handle, filter descriptor, filter data, or reordered data is NULL. Or, if the bias reordering is requested (`reorderBias > 0`), the bias data or reordered bias data is NULL. This status can also be returned if the filter dimension size is not 4.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

Filter descriptor data type is not `CUDNN_DATA_INT8x32`; the filter descriptor tensor is not in a vectorized layout (`CUDNN_TENSOR_NCHW_VECT_C`).

## 5.2.27. `cudaSetConvolution2dDescriptor()`

This function initializes a previously created convolution descriptor object into a 2D correlation. This function assumes that the tensor and filter descriptors correspond 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.

```

cudaStatus_t cudaSetConvolution2dDescriptor(
    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)

```

### Parameters

**convDesc**

*Input/Output.* Handle to a previously created convolution descriptor.

**pad\_h**

*Input.* Zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

**pad\_w**

*Input.* Zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

**u**

*Input.* Vertical filter stride.

**v**

*Input.* Horizontal filter stride.

**dilation\_h**

*Input.* Filter height dilation.

**dilation\_w**

*Input.* Filter width dilation.

**mode**

*Input.* Selects between `CUDNN_CONVOLUTION` and `CUDNN_CROSS_CORRELATION`.

**computeType**

*Input.* Compute precision.



## Returns

### CUDNN\_STATUS\_SUCCESS

The object was set successfully.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ The descriptor `convDesc` is `NIL`.
- ▶ One of the parameters `pad_h`, `pad_w` is strictly negative.
- ▶ One of the parameters `u`, `v` is negative or zero.
- ▶ One of the parameters `dilation_h`, `dilation_w` is negative or zero.
- ▶ The parameter `mode` has an invalid enumerant value.

## 5.2.28. `cudaSetConvolutionGroupCount()`

This function allows the user to specify the number of groups to be used in the associated convolution.

```
cudaStatus_t cudaSetConvolutionGroupCount(
    cudaConvolutionDescriptor_t convDesc,
    int groupCount)
```

## Returns

### CUDNN\_STATUS\_SUCCESS

The group count was set successfully.

### CUDNN\_STATUS\_BAD\_PARAM

An invalid convolution descriptor was provided.

## 5.2.29. `cudaSetConvolutionMathType()`

This function allows the user to specify whether or not the use of tensor op is permitted in the library routines associated with a given convolution descriptor.

```
cudaStatus_t cudaSetConvolutionMathType(
    cudaConvolutionDescriptor_t convDesc,
    cudaMathType_t mathType)
```

## 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.

## 5.2.30. `cudaSetConvolutionNdDescriptor()`

This function initializes a previously created generic convolution descriptor object into a `Nd` 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.

```

cudaStatus_t cudaSetConvolutionNdDescriptor(
    cudaConvolutionDescriptor_t    convDesc,
    int                            arrayLength,
    const int                      padA[],
    const int                      filterStrideA[],
    const int                      dilationA[],
    cudaConvolutionMode_t         mode,
    cudaDataType_t                 dataType)

```

## Parameters

### **convDesc**

*Input/Output.* Handle to a previously created convolution descriptor.

### **arrayLength**

*Input.* Dimension of the convolution.

### **padA**

*Input.* Array of dimension `arrayLength` containing the zero-padding size for each dimension. For every dimension, the padding represents the number of extra zeros implicitly concatenated at the start and at the end of every element of that dimension.

### **filterStrideA**

*Input.* Array of dimension `arrayLength` containing the filter stride for each dimension. For every dimension, the filter stride represents the number of elements to slide to reach the next start of the filtering window of the next point.

### **dilationA**

*Input.* Array of dimension `arrayLength` containing the dilation factor for each dimension.

### **mode**

*Input.* Selects between `CUDNN_CONVOLUTION` and `CUDNN_CROSS_CORRELATION`.

### **dataType**

*Input.* Selects the data type in which the computation will be done.



Note: `CUDNN_DATA_HALF` in `cudaSetConvolutionNdDescriptor()` with `HALF_CONVOLUTION_BWD_FILTER` is not recommended as it is known to not be useful for any practical use case for training and will be considered to be blocked in a future cuDNN release. The use of `CUDNN_DATA_HALF` for input tensors in `cudaSetTensorNdDescriptor()` and `CUDNN_DATA_FLOAT` in `cudaSetConvolutionNdDescriptor()` with `HALF_CONVOLUTION_BWD_FILTER` is recommended and is used with the automatic mixed precision (AMP) training in many well known deep learning frameworks.

## 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`.

### 5.2.31. `cudaSetConvolutionReorderType()`

This function sets the convolution reorder type for the given convolution descriptor.

```
cudaStatus_t cudaSetConvolutionReorderType(
    cudaConvolutionDescriptor_t convDesc,
    cudaReorderType_t reorderType);
```

#### Parameters

**convDesc**

*Input.* The convolution descriptor for which the reorder type should be set.

**reorderType**

*Input.* Set the reorder type to this value. For more information, refer to [cudaReorderType\\_t](#).

#### Returns

**CUDNN\_STATUS\_BAD\_PARAM**

The reorder type supplied is not supported.

**CUDNN\_STATUS\_SUCCESS**

Reorder type is set successfully.

---

# Chapter 6. `cuda_cnn_train.so` Library

This entity contains all routines related to convolutional neural networks needed during training time. The `cuda_cnn_train` library depends on `cuda_ops_infer`, `cuda_ops_train`, and `cuda_cnn_infer`.

For the backend data and descriptor types, refer to the [cuDNN Backend API](#) section.

## 6.1. Data Type References

These are the data type references in the `cuda_cnn_train.so` library.

### 6.1.1. Pointer To Opaque Struct Types

These are the pointers to the opaque struct types in the `cuda_cnn_train.so` library.

#### 6.1.1.1. `cudaFusedOpsConstParamPack_t`

`cudaFusedOpsConstParamPack_t` is a pointer to an opaque structure holding the description of the `cudaFusedOps` constant parameters. Use the function [`cudaCreateFusedOpsConstParamPack\(\)`](#) to create one instance of this structure, and the function [`cudaDestroyFusedOpsConstParamPack\(\)`](#) to destroy a previously-created descriptor.

#### 6.1.1.2. `cudaFusedOpsPlan_t`

`cudaFusedOpsPlan_t` is a pointer to an opaque structure holding the description of the `cudaFusedOpsPlan`. This descriptor contains the plan information, including the problem type and size, which kernels should be run, and the internal workspace partition. Use the function [`cudaCreateFusedOpsPlan\(\)`](#) to create one instance of this structure, and the function [`cudaDestroyFusedOpsPlan\(\)`](#) to destroy a previously-created descriptor.

#### 6.1.1.3. `cudaFusedOpsVariantParamPack_t`

`cudaFusedOpsVariantParamPack_t` is a pointer to an opaque structure holding the description of the `cudaFusedOps` variant parameters. Use the function

[cudaCreateFusedOpsVariantParamPack\(\)](#) to create one instance of this structure, and the function [cudaDestroyFusedOpsVariantParamPack\(\)](#) to destroy a previously-created descriptor.

## 6.1.2. Struct Types

These are the struct types in the `cuda_cnn_train.so` library.

### 6.1.2.1. `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 runs 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 the provided workspace is insufficient.
- ▶ `CUDNN_STATUS_INTERNAL_ERROR` if any error occurred during timing calculations or workspace deallocation.
- ▶ Otherwise, this will be the return status of [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.

## 6.1.3. Enumeration Types

These are the enumeration types in the `cuda_cnn_train.so` library.

### 6.1.3.1. cudnnFusedOps\_t

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

#### Members and Descriptions

**CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS = 0**

On a per-channel basis, it performs these operations in this order: `scale`, `add bias`, `activation`, `convolution`, and generate `batchnorm` statistics.

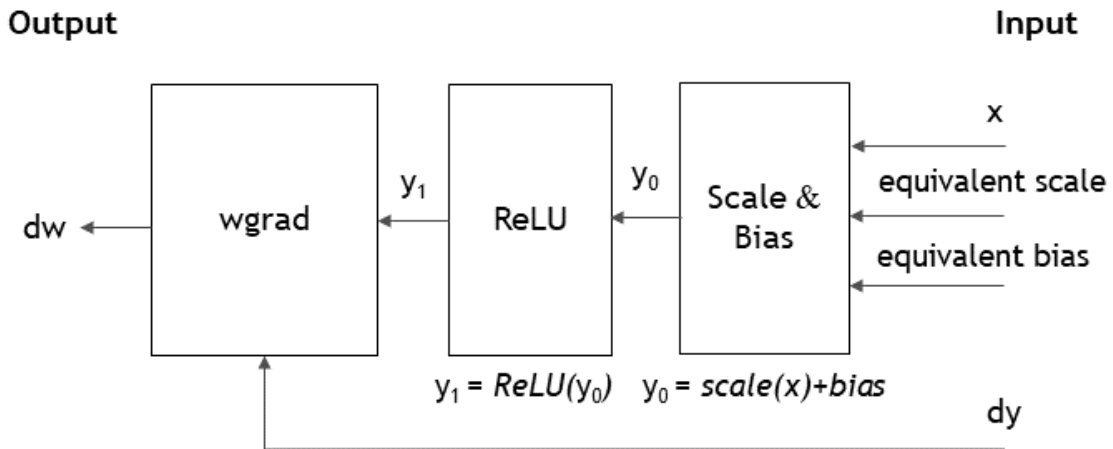
Figure 1. Operations Per Channel For

**CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS**

**CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD = 1**

On a per-channel basis, it performs these operations in this order: `scale`, `add bias`, `activation`, `convolution backward weights`, and generate `batchnorm` statistics.

#### CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD



**CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING = 2**

Computes the `equivalent scale` and `bias` from `ySum`, `ySqSum` and learned `scale`, `bias`. Optionally, update running statistics and generate saved stats.

**CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE = 3**

Computes the `equivalent scale` and `bias` from the learned running statistics and the learned `scale`, `bias`.

**CUDNN\_FUSED\_CONV\_SCALE\_BIAS\_ADD\_ACTIVATION = 4**

On a per-channel basis, performs these operations in this order: convolution, scale, add bias, element-wise addition with another tensor, and activation.

**CUDNN\_FUSED\_SCALE\_BIAS\_ADD\_ACTIVATION\_GEN\_BITMASK = 5**

On a per-channel basis, performs these operations in this order: scale and bias on one tensor, scale and bias on a second tensor, element-wise addition of these two tensors, and on the resulting tensor performs activation and generates activation bit mask.

**CUDNN\_FUSED\_DACTIVATION\_FORK\_DBATCHNORM = 6**

On a per-channel basis, performs these operations in this order: backward activation, fork (meaning, write out gradient for the residual branch), and backward batch norm.

### 6.1.3.2. cudnnFusedOpsConstParamLabel\_t

The cudnnFusedOpsConstParamLabel\_t is an enumerated type for the selection of the type of the cudnnFusedOps descriptor. For more information, refer to

[cudnnSetFusedOpsConstParamPackAttribute\(\)](#).

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

Table 31. Legend For Tables in cudnnFusedOpsConstParamLabel\_t

Short Form Used	Stands For
Setter	<a href="#">cudnnSetFusedOpsConstParamPackAttribute()</a>
Getter	<a href="#">cudnnGetFusedOpsConstParamPackAttribute()</a>
X_PointerPlaceholder_t	<a href="#">cudnnFusedOpsPointerPlaceholder_t</a>
x_ prefix in the Attribute column	Stands for CUDNN_PARAM_ in the enumerator name

Table 32. CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS In cudnnFusedOpsConstParamLabel\_t

For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in cudnnFusedOpsConstParamLabel_t			
Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_XDESC	In the setter, the *param should be xDesc, a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t</a> .	Tensor descriptor describing the size, layout, and datatype of the x (input) tensor.	NULL
X_XDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <a href="#">X_PointerPlaceholder_t</a> .	Describes whether xData pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_MODE	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnBatchNormMode_t</a> .	Describes the mode of operation for the scale, bias and the statistics. As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias, and statistics are all per-channel.	CUDNN_BATCHNORM_PER_ACTIVATION
X_BN_EQSCALEBIAS_DESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t</a> .	Tensor descriptor describing the size, layout, and datatype of the batchNorm equivalent scale and	NULL



**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS in cudnnFusedOpsConstParamLabel\_t**


Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		bias tensors. The shapes must match the mode specified in CUDNN_PARAM_BN_MODE. If set to NULL, both scale and bias operation will become a NOP.	
X_BN_EQSCALE_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceholder	Describes whether batchnorm equivalent scale pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, then the scale operation becomes a NOP.	CUDNN_PTR_NULL
X_BN_EQBIAS_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceholder	Describes whether batchnorm equivalent bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, then the bias operation becomes a NOP.	CUDNN_PTR_NULL
X_ACTIVATION_DESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnActivationDescriptor_t</a> *	Describes the activation operation.  As of cuDNN 7.6.0, only activation modes of CUDNN_ACTIVATION_RELU and CUDNN_ACTIVATION_IDENTITY are supported. If set to NULL or if the activation mode is set to CUDNN_ACTIVATION_IDENTITY,	NULL

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		then the activation in the op sequence becomes a NOP.	
X_CONV_DESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnConvolutionDescriptor_t*</a> .	Describes the convolution operation.	NULL
X_WDESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnFilterDescriptor_t*</a> .	Filter descriptor describing the size, layout and datatype of the w (filter) tensor.	NULL
X_WDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <a href="#">X_PointerPlaceholder_t*</a> .	Describes whether w (filter) tensor pointer in the <a href="#">VariantParamPack</a> will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_YDESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t*</a> .	Tensor descriptor describing the size, layout and datatype of the y (output) tensor.	NULL
X_YDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <a href="#">X_PointerPlaceholder_t*</a> .	Describes whether y (output) tensor pointer in the <a href="#">VariantParamPack</a> will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_YSTATS_DESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t*</a> .	Tensor descriptor describing the size, layout and datatype of the sum of y and sum of y square tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE.  If set to NULL, the y statistics generation	NULL

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		operation will become a NOP.	
X_YSUM_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolder_	Describes whether sum of y pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, the y statistics generation operation will become a NOP.	CUDNN_PTR_NULL
X_YSQSUM_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolder_	Describes whether sum of y square pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, the y statistics generation operation will become a NOP.	CUDNN_PTR_NULL

 **Note:**

- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_NULL, then the device pointer in the VariantParamPack needs to be NULL as well.
- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_ELEM\_ALIGNED or CUDNN\_PTR\_16B\_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and need to be at least element-aligned or 16 bytes-aligned, respectively.

As of cuDNN 7.6.0, if the conditions in [Table 33](#) are met, then the fully fused fast path will be triggered. Otherwise, a slower partially fused path will be triggered.

Table 33. Conditions for Fully Fused Fast Path (Forward) for `cudaFusedOpsConstParamLabel_t`

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

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

Table 34. CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsConstParamLabel\_t

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

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		bias tensors. The shapes must match the mode specified in CUDNN_PARAM_BN_MODE. If set to NULL, both scale and bias operation will become a NOP.	
X_BN_EQSCALE_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolderVariantParamPack	Describes whether batchnorm equivalent scale pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. If set to CUDNN_PTR_NULL, then the scale operation becomes a NOP.	CUDNN_PTR_NULL
X_BN_EQBIAS_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolderVariantParamPack	Describes whether batchnorm equivalent bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *. If set to CUDNN_PTR_NULL, then the bias operation becomes a NOP.	CUDNN_PTR_NULL
X_ACTIVATION_DESC	In the setter, the *param should be a pointer to a previously initialized cudnnActivationDescriptor_t*	Describes the activation operation. As of cuDNN 7.6.0, only the activation mode of CUDNN_ACTIVATION_RELU and CUDNN_ACTIVATION_IDENTITY is supported. If	NULL

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		set to NULL or if the activation mode is set to CUDNN_ACTIVATION_IDENTITY, then the activation in the op sequence becomes a NOP.	
X_CONV_DESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnConvolutionDescriptor_t*</a> .	Describes the convolution operation.	NULL
X_DWDESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnFilterDescriptor_t*</a> .	Filter descriptor describing the size, layout and datatype of the dw (filter gradient output) tensor.	NULL
X_DWDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <a href="#">X_PointerPlaceholder_t*</a> .	Describes whether dw (filter gradient output) tensor pointer in the <a href="#">VariantParamPack</a> will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_DYDESC	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t*</a> .	Tensor descriptor describing the size, layout and datatype of the dy (gradient input) tensor.	NULL
X_DYDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <a href="#">X_PointerPlaceholder_t*</a> .	Describes whether dy (gradient input) tensor pointer in the <a href="#">VariantParamPack</a> will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL

 **Note:**

- If the corresponding pointer placeholder in [ConstParamPack](#) is set to CUDNN\_PTR\_NULL, then the device pointer in the [VariantParamPack](#) needs to be NULL as well.

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

As of cuDNN 7.6.0, if the conditions in [Table 35](#) are met, then the fully fused fast path will be triggered. Otherwise a slower partially fused path will be triggered.

Table 35. Conditions for Fully Fused Fast Path (Backward) for `cudaFusedOpsConstParamLabel_t`

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



Parameter	Condition
	Filter gradient's alignment is CUDNN_PTR_16B_ALIGNED
CUDNN_PARAM_DYDESC CUDNN_PARAM_DYDATA_PLACEHOLDER	Tensor is 4 dimensional Datatype is CUDNN_DATA_HALF Layout is NHWC fully packed Alignment is CUDNN_PTR_16B_ALIGNED

Table 36. CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsConstParamLabel\_t

For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_TRAINING in cudnnFusedOpsConstParamLabel_t			
Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_BN_MODE	In the setter, the *param should be a pointer to a previously initialized <code>cudnnBatchNormMode_t*</code> .	Describes the mode of operation for the scale, bias and the statistics. As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias and statistics are all per-channel.	CUDNN_BATCHNORM_PER_ACTIVATION
X_YSTATS_DESC	In the setter, the *param should be a pointer to a previously initialized <code>cudnnTensorDescriptor_t*</code> .	Tensor descriptor describing the size, layout and datatype of the sum of y and sum of y square tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE.	NULL
X_YSUM_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized <code>X_PointerPlaceholder_t*</code> .	Describes whether sum of y pointer in the <code>VariantParamPack</code> will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_YQSUM_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder	Describes whether sum of y square pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_SCALEBIAS_MEANVAR_DESCRIPTOR	In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t	A common tensor descriptor describing the size, layout and datatype of the batchNorm trained scale, bias and statistics tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE (similar to the bnScaleBiasMeanVarDesc field in the cudnnBatchNormalization* API).	NULL
X_BN_SCALE_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder	Describes whether the batchNorm trained scale pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If the output of BN_EQSCALE is not needed, then this is not needed and may be NULL.	CUDNN_PTR_NULL
X_BN_BIAS_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder	Describes whether the batchNorm trained bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsConstParamLabel\_t**

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

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsConstParamLabel\_t**

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

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

Table 37. CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsConstParamLabel\_t

For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOpsConstParamLabel_t			
Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_BN_MODE	In the setter, the *param should be a pointer to a previously initialized cudnnBatchNormMode_t*	Describes the mode of operation for the scale, bias and the statistics.  As of cuDNN 7.6.0, only CUDNN_BATCHNORM_SPATIAL and CUDNN_BATCHNORM_SPATIAL_PERSISTENT are supported, meaning, scale, bias	CUDNN_BATCHNORM_PER_ACTIVATION

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
		and statistics are all per-channel.	
X_BN_SCALEBIAS_MEANVAR_DESCRIPTOR	In the setter, the *param should be a pointer to a previously initialized <a href="#">cudnnTensorDescriptor_t</a>	A common tensor descriptor describing the size, layout and datatype of the batchNorm trained scale, bias and statistics tensors. The shapes need to match the mode specified in CUDNN_PARAM_BN_MODE (similar to the bnScaleBiasMeanVarDesc field in the cudnnBatchNormalization* API).	NULL
X_BN_SCALE_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolder	Describes whether the batchNorm trained scale pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_BIAS_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolder	Describes whether the batchNorm trained bias pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_RUNNING_MEAN_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PoInterPlaceHolder	Describes whether the batchNorm running mean pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_RUNNING_VAR_PLACEHOLDER	In the setter, the *param should	Describes whether the batchNorm	CUDNN_PTR_NULL

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsConstParamLabel\_t**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
	be a pointer to a previously initialized <code>X_PoInterPlaceHolder</code>	running variance pointer in the <code>VariantParamPack</code> will be <code>NULL</code> , or if not, user promised pointer alignment *.	
<code>X_BN_EQSCALEBIAS_DESC</code>	In the setter, the *param should be a pointer to a previously initialized <code>cudnnTensorDescriptor</code>	Tensor descriptor describing the size, layout and datatype of the batchNorm equivalent scale and bias tensors. The shapes need to match the mode specified in <code>CUDNN_PARAM_BN_MODE</code> .	<code>NULL</code>
<code>X_BN_EQSCALE_PLACEHOLDER</code>	In the setter, the *param should be a pointer to a previously initialized <code>X_PoInterPlaceHolder</code>	Describes whether batchnorm equivalent scale pointer in the <code>VariantParamPack</code> will be <code>NULL</code> , or if not, user promised pointer alignment *.  If set to <code>CUDNN_PTR_NULL</code> , then the computation for this output becomes a NOP.	<code>CUDNN_PTR_NULL</code>
<code>X_BN_EQBIAS_PLACEHOLDER</code>	In the setter, the *param should be a pointer to a previously initialized <code>X_PoInterPlaceHolder</code>	Describes whether batchnorm equivalent bias pointer in the <code>VariantParamPack</code> will be <code>NULL</code> , or if not, user promised pointer alignment *.  If set to <code>CUDNN_PTR_NULL</code> , then the computation for this output becomes a NOP.	<code>CUDNN_PTR_NULL</code>

Table 38. CUDNN\_FUSED\_CONVOLUTION\_SCALE\_BIAS\_ADD\_RELU in cudnnFusedOpsConstParamLabel\_t

<b>For the attribute CUDNN_FUSED_CONVOLUTION_SCALE_BIAS_ADD_RELU in cudnnFusedOpsConstParamLabel_t</b> <b>This operation performs the following computation, where * denotes convolution operator: <math>y=1(w*x)+2z+b</math></b>			
Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_XDESC	In the setter, the *param should be xDesc, a pointer to a previously initialized cudnnTensorDescriptor_t.	Tensor descriptor describing the size, layout and datatype of the x (input) tensor.	NULL
X_XDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t.	Describes whether xData pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_CONV_DESC	In the setter, the *param should be a pointer to a previously initialized cudnnConvolutionDescriptor_t*.	Describes the convolution operation.	NULL
X_WDESC	In the setter, the *param should be a pointer to a previously initialized cudnnFilterDescriptor_t*.	Filter descriptor describing the size, layout and datatype of the w (filter) tensor.	NULL
X_WDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceHolder_t.	Describes whether w (filter) tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL
X_BN_EQSCALEBIAS_DESC	In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t.	Tensor descriptor describing the size, layout and datatype of the $\alpha_1$ scale and bias tensors. The tensor should have shape (1,K,1,1), K is the number of output features.	NULL



**For the attribute CUDNN\_FUSED\_CONVOLUTION\_SCALE\_BIAS\_ADD\_RELU in cudnnFusedOpsConstParamLabel\_t**  
**This operation performs the following computation, where \* denotes convolution operator:  $y=1(w*x)+2 z+b$**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
X_BN_EQSCALE_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceholder	Describes whether batchnorm equivalent scale or $\alpha_1$ tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, then $\alpha_1$ scaling becomes an NOP.	CUDNN_PTR_NULL
X_ZDESC	In the setter, the *param should be xDesc, a pointer to a previously initialized cudnnTensorDescriptor_t.	Tensor descriptor describing the size, layout and datatype of the z tensor.  If unset, then z scale-add term becomes a NOP.	NULL
CUDNN_PARAM_ZDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceholder	Describes whether z tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, then z scale-add term becomes a NOP.	CUDNN_PTR_NULL
CUDNN_PARAM_BN_Z_EQSCALE	In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t.	Tensor descriptor describing the size, layout and datatype of the $\alpha_2$ tensor.  If set to NULL then scaling for input z becomes a NOP.	NULLPTR
CUDNN_PARAM_BN_Z_EQSCALE	In the setter, the *param should be a pointer to a	Describes whether batchnorm z-equivalent scaling	CUDNN_PTR_NULL

**For the attribute CUDNN\_FUSED\_CONVOLUTION\_SCALE\_BIAS\_ADD\_RELU in cudnnFusedOpsConstParamLabel\_t**  
**This operation performs the following computation, where \* denotes convolution operator:  $y=1(w*x)+2 z+b$**

Attribute	Expected Descriptor Type Passed in, in the Setter	Description	Default Value After Creation
	previously initialized X_PointerPlaceholder	pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.  If set to CUDNN_PTR_NULL, then the scaling for input z becomes a NOP.	
X_ACTIVATION_DESC	In the setter, the *param should be a pointer to a previously initialized cudnnActivationDescriptor_t*.	Describes the activation operation.  As of 7.6.0, only activation modes of CUDNN_ACTIVATION_RELU and CUDNN_ACTIVATION_IDENTITY are supported. If set to NULL or if the activation mode is set to CUDNN_ACTIVATION_IDENTITY, then the activation in the op sequence becomes a NOP.	NULL
X_YDESC	In the setter, the *param should be a pointer to a previously initialized cudnnTensorDescriptor_t*.	Tensor descriptor describing the size, layout and datatype of the y (output) tensor.	NULL
X_YDATA_PLACEHOLDER	In the setter, the *param should be a pointer to a previously initialized X_PointerPlaceholder	Describes whether y (output) tensor pointer in the VariantParamPack will be NULL, or if not, user promised pointer alignment *.	CUDNN_PTR_NULL

### 6.1.3.3. cudnnFusedOpsPointerPlaceholder\_t

cudnnFusedOpsPointerPlaceholder\_t is an enumerated type used to select the alignment type of the cudnnFusedOps descriptor pointer.

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

### 6.1.3.4. cudnnFusedOpsVariantParamLabel\_t

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

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

Table 39. Legend For Tables in cudnnFusedOpsVariantParamLabel\_t

Short-Form Used	Stands For
Setter	<a href="#">cudnnSetFusedOpsVariantParamPackAttribute()</a>
Getter	<a href="#">cudnnGetFusedOpsVariantParamPackAttribute()</a>
x_ prefix in the Attribute key column	Stands for CUDNN_PTR_ or CUDNN_SCALAR_ in the enumerator name.

Table 40. CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS in cudnnFusedOpsVariantParamLabel\_t

For the attribute CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS in cudnnFusedOpsVariantParamLabel_t				
Attribute key	Expected Descriptor Type Passed in, in the Setter	I/O Type	Description	Default Value
X_XDATA	void *	input	Pointer to $x$ (input) tensor on device, need to agree with previously set CUDNN_PARAM_XDATA_PLACEHOLDER attribute *.	NULL
X_BN_EQSCALE	void *	input	Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *.	NULL
X_BN_EQBIAS	void *	input	Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *.	NULL
X_WDATA	void *	input	Pointer to $w$ (filter) tensor on device, need to agree with previously set CUDNN_PARAM_WDATA_PLACEHOLDER attribute *.	NULL
X_YDATA	void *	output	Pointer to $y$ (output) tensor on device, need to agree with previously set CUDNN_PARAM_YDATA_PLACEHOLDER attribute *.	NULL
X_YSUM	void *	output	Pointer to sum of $y$ tensor on device, need to agree with previously set CUDNN_PARAM_YSUM_PLACEHOLDER attribute *.	NULL
X_YSQSUM	void *	output	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_YSQSUM_PLACEHOLDER attribute *.	NULL

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_CONV\_BNSTATS in cudnnFusedOpsVariantParamLabel\_t**

Attribute key	Expected Descriptor Type Passed in, in the Setter	I/O Type	Description	Default Value
X_WORKSPACE	void *	input	Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0.	NULL
X_SIZE_T_WORKSPACE_SIZE	size_t	input	Pointer to a size_t value in host memory describing the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan.	0

**Note:**

- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_NULL, then the device pointer in the VariantParamPack needs to be NULL as well
- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_ELEM\_ALIGNED or CUDNN\_PTR\_16B\_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 41. CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsVariantParamLabel\_t

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsVariantParamLabel\_t**

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

**For the attribute CUDNN\_FUSED\_SCALE\_BIAS\_ACTIVATION\_WGRAD in cudnnFusedOpsVariantParamLabel\_t**

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

**Note:**

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

VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 42. CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsVariantParamLabel\_t

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

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsVariantParamLabel\_t**

Attribute key	Expected Description Type Passed in, in the Setter	I/O Type	Description	Default Value
			CUDNN_PARAM_BN_RUNNING_MEAN_PLACEHOLDER attribute *.	
X_BN_RUNNING_VAR	void *	input/output	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_VAR_PLACEHOLDER attribute *.	NULL
X_BN_EQSCALE	void *	output	Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *.	NULL
X_BN_EQBIAS	void *	output	Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *.	NULL
X_INT64_T_BN_ACCUMULATION_COUNT	int64_t *	input	<p>Pointer to a scalar value in int64_t on host memory.</p> <p>This value should describe the number of tensor elements accumulated in the sum of <math>y</math> and sum of <math>y</math> square tensors.</p> <p>For example, in the single GPU use case, if the mode is CUDNN_BATCHNORM_SPATIAL or CUDNN_BATCHNORM_SPATIAL_PERSISTENT, the value should be equal to <math>N*H*W</math> of the tensor from which the statistics are calculated.</p> <p>In multi-GPU use case, if all-reduce has been performed on the sum of <math>y</math> and sum of <math>y</math> square tensors, this value should be the sum of the single GPU accumulation count on each of the GPUs.</p>	0



**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_TRAINING in cudnnFusedOpsVariantParamLabel\_t**

Attribute key	Expected Description Type Passed in, in the Setter	I/O Type	Description	Default Value
X_DOUBLE_BN_EXP_AVG_FACTOR	double *	input	Pointer to a scalar value in double on host memory.  Factor used in the moving average computation. See <code>exponentialAverageFactor</code> in <code>cudnnBatchNormalization*</code> APIs.	0.0
X_DOUBLE_BN_EPSILON	double *	input	Pointer to a scalar value in double on host memory.  A conditioning constant used in the batch normalization formula. Its value should be equal to or greater than the value defined for <code>CUDNN_BN_MIN_EPSILON</code> in <code>cudnn.h</code> .  See <code>exponentialAverageFactor</code> in <code>cudnnBatchNormalization*</code> APIs.	0.0
X_WORKSPACE	void *	input	Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0.	NULL
X_SIZE_T_WORKSPACE_SIZE_IN_BYTES	size_t *	input	Pointer to a <code>size_t</code> value in host memory describing the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in <code>cudnnMakeFusedOpsPlan</code> .	0

**Note:**

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

VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 43. CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsVariantParamLabel\_t

For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOpsVariantParamLabel_t				
Attribute key	Expected Descripto Type Passed in, in the Setter	I/O Type	Description	Default Value
X_BN_SCALE	void *	input	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_BN_SCALE_PLACEHOLDER attribute *.	NULL
X_BN_BIAS	void *	input	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_BN_BIAS_PLACEHOLDER attribute *.	NULL
X_BN_RUNNING_MEAN	void *	input/output	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_MEAN_PLACEHOLDER attribute *.	NULL
X_BN_RUNNING_VAR	void *	input/output	Pointer to sum of $y$ square tensor on device, need to agree with previously set CUDNN_PARAM_BN_RUNNING_VAR_PLACEHOLDER attribute *.	NULL
X_BN_EQSCALE	void *	output	Pointer to batchnorm equivalent scale tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *.	NULL
X_BN_EQBIAS	void *	output	Pointer to batchnorm equivalent bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_EQBIAS_PLACEHOLDER attribute *.	NULL

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsVariantParamLabel\_t**

Attribute key	Expected Descriptor Type Passed in, in the Setter	I/O Type	Description	Default Value
X_DOUBLE_BN_EPSILON	double *	input	<p>Pointer to a scalar value in double on host memory.</p> <p>A conditioning constant used in the batch normalization formula. Its value should be equal to or greater than the value defined for CUDNN_BN_MIN_EPSILON in cudnn.h.</p> <p>See exponentialAverageFactor in cudnnBatchNormalization* APIs.</p>	0.0
X_WORKSPACE	void *	input	<p>Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0.</p>	NULL
X_SIZE_T_WORKSPACE_SIZE_IN_BYTES	size_t *	input	<p>Pointer to a size_t value in host memory describing the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan.</p>	0



**Note:**

- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_NULL, then the device pointer in the VariantParamPack needs to be NULL as well.
- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_ELEM\_ALIGNED or CUDNN\_PTR\_16B\_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

Table 44. CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsVariantParamLabel\_t

For the attribute CUDNN_FUSED_BN_FINALIZE_STATISTICS_INFERENCE in cudnnFusedOpsVariantParamLabel_t				
Attribute key	Expected Descriptor Type Passed in, in the Setter	I/O Type	Description	Default Value
X_XDATA	void *	input	Pointer to x (image) tensor on device, need to agree with previously set CUDNN_PARAM_XDATA_PLACEHOLDER attribute *.	NULL
X_WDATA	void *	input	Pointer to w (filter) tensor on device, need to agree with previously set CUDNN_PARAM_WDATA_PLACEHOLDER attribute *.	NULL
X_BN_EQSCALE	void *	input	Pointer to alpha1 or batchnorm equivalent scale tensor on device; need to agree with previously set CUDNN_PARAM_BN_EQSCALE_PLACEHOLDER attribute *.	NULL
X_ZDATA	void *	input	Pointer to z ( tensor on device; Need to agree with previously set CUDNN_PARAM_YDATA_PLACEHOLDER attribute *.	NULL
X_BN_Z_EQSCALE	void *	input	Pointer to alpha2, equivalent scale tensor for z; Need to agree with previously set CUDNN_PARAM_BN_Z_EQSCALE_PLACEHOLDER attribute *.	NULL
X_BN_Z_EQBIAS	void *	input	Pointer to batchnorm equivalent	NULL

**For the attribute CUDNN\_FUSED\_BN\_FINALIZE\_STATISTICS\_INFERENCE in cudnnFusedOpsVariantParamLabel\_t**

Attribute key	Expected Descriptor Type Passed in, in the Setter	I/O Type	Description	Default Value
			bias tensor on device, need to agree with previously set CUDNN_PARAM_BN_Z_EQBIAS_PLACEHOLDER attribute *.	
X_YDATA	void *	output	Pointer to y (output) tensor on device, need to agree with previously set CUDNN_PARAM_YDATA_PLACEHOLDER attribute *.	NULL
X_WORKSPACE	void *	input	Pointer to user allocated workspace on device. Can be NULL if the workspace size requested is 0.	NULL
X_SIZE_T_WORKSPACE_SIZE	size_t*	input	Pointer to a size_t value in host memory describing the user allocated workspace size in bytes. The amount needs to be equal or larger than the amount requested in cudnnMakeFusedOpsPlan.	0

**Note:**

- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_NULL, then the device pointer in the VariantParamPack needs to be NULL as well.
- ▶ If the corresponding pointer placeholder in ConstParamPack is set to CUDNN\_PTR\_ELEM\_ALIGNED or CUDNN\_PTR\_16B\_ALIGNED, then the device pointer in the VariantParamPack may not be NULL and needs to be at least element-aligned or 16 bytes-aligned, respectively.

## 6.2. API Functions

These are the API functions in the `cuda_cnn_train.so` library.

### 6.2.1. `cudaCnnTrainVersionCheck()`

This function checks whether the version of the `CnnTrain` subset of the library is consistent with the other sub-libraries.

```
cudaStatus_t cudaCnnTrainVersionCheck(void)
```

#### Returns

##### **CUDNN\_STATUS\_SUCCESS**

The version is consistent with other sub-libraries.

##### **CUDNN\_STATUS\_VERSION\_MISMATCH**

The version of `CnnTrain` is not consistent with other sub-libraries. Users should check the installation and make sure all sub-component versions are consistent.

### 6.2.2. `cudaConvolutionBackwardBias()`

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.

```
cudaStatus_t cudaConvolutionBackwardBias(
    cudaHandle_t          handle,
    const void            *alpha,
    const cudaTensorDescriptor_t dyDesc,
    const void            *dy,
    const void            *beta,
    const cudaTensorDescriptor_t dbDesc,
    void                  *db)
```

#### Parameters

##### **handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [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]*resultValue + beta[0]*priorDstValue
```

For more information, refer to [Scaling Parameters](#).

##### **dyDesc**

*Input.* Handle to the previously initialized input tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**dy**

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

**dbDesc**

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

**db**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The operation was launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

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

### 6.2.3. `cudaConvolutionBackwardFilter()`

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`.

```

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)
    
```

## Parameters

**handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [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
```

For more information, refer to [Scaling Parameters](#).

**xDesc**

*Input.* Handle to a previously initialized tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**x**

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

**dyDesc**

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

**dy**

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

**convDesc**

*Input.* Previously initialized convolution descriptor. For more information, refer to [cudaConvolutionDescriptor\\_t](#).

**algo**

*Input.* Enumerant that specifies which convolution algorithm should be used to compute the results. For more information, refer to [cudaConvolutionBwdFilterAlgo\\_t](#).

**workSpace**

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

**workSpaceSizeInBytes**

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

**dwDesc**

*Input.* Handle to a previously initialized filter gradient descriptor. For more information, refer to [cudaFilterDescriptor\\_t](#).

**dw**

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

## Supported configurations

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



Table 45. Supported Configurations for `cudaConvolutionBackwardFilter()`

Data Type Configurations	xDesc, dyDesc, and dwDesc Data Type	convDesc Data Type
TRUE_HALF_CONFIG (only supported on architectures with true FP16 support, meaning, compute capability 5.3 and later)	CUDNN_DATA_HALF	CUDNN_DATA_HALF
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT
PSEUDO_BFLOAT16_CONFIG	CUDNN_DATA_BFLOAT16	CUDNN_DATA_FLOAT
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE

### Supported algorithms

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

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

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

- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_0 (`_ALGO_0`)
- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_1 (`_ALGO_1`)
- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_3 (`_ALGO_3`)
- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_FFT (`_FFT`)
- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_FFT\_TILING (`_FFT_TILING`)
- ▶ CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_WINOGRAD\_NONFUSED (`_WINOGRAD_NONFUSED`)
- ▶ CUDNN\_TENSOR\_NCHW (`_NCHW`)
- ▶ CUDNN\_TENSOR\_NHWC (`_NHWC`)
- ▶ CUDNN\_TENSOR\_NCHW\_VECT\_C (`_NCHW_VECT_C`)

Table 46. Supported Algorithms for  
 cudnnConvolutionBackwardFilter() 2D Convolutions:  
 dwDesc: \_NHWC

Filter descriptor dwDesc: <u>_NHWC (refer to <a href="#">cudnnTensorFormat_t</a>)</u>					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for dyDesc	Tensor Formats Supported for dxDesc	Data Type Configuration Supported	Important
_ALGO_0 and _ALGO_1		NHWC HWC-packed.	NHWC HWC-packed	PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG	

Table 47. Supported Algorithms for  
 cudnnConvolutionBackwardFilter() 2D Convolutions:  
 dwDesc: \_NCHW

Filter descriptor dwDesc: <u>_NCHW</u>					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for dyDesc	Tensor Formats Supported for dxDesc	Data Type Configuration Supported	Important
_ALGO_0	No	All except _NCHW_VECT_C	NCHW CHW-packed	PSEUDO_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: greater than 0 for all dimensions convDesc Group Count Support: Greater than 0
_ALGO_1	Yes	All except _NCHW_VECT_C	NCHW CHW-packed	PSEUDO_HALF_CONFIG TRUE_HALF_CONFIG PSEUDO_BFLOAT16_CONFIG FLOAT_CONFIG DOUBLE_CONFIG	Dilation: greater than 0 for all dimensions convDesc Group Count Support: Greater than 0

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

Filter descriptor <code>dwDesc: _NCHW</code>					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
					zero-padding width
<code>_ALGO_3</code>	No	All except <code>_NCHW_VECT_C</code>	NCHW CHW-packed	<code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code>	Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0
<code>_WINOGRAD_NONFUSED</code>	Yes	All except <code>_NCHW_VECT_C</code>	NCHW CHW-packed	<code>TRUE_HALF_CONFIG</code> <code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code>	Dilation: 1 for all dimensions convDesc Group Count Support: Greater than 0  convDesc vertical and horizontal filter stride must equal 1  dwDesc filter (height, width) must be (3,3) or (5,5)  If dwDesc filter (height, width) is (5,5), then the data type config <code>TRUE_HALF_CONFIG</code> is not supported.
<code>_FFT_TILING</code>	Yes	NCHW CHW-packed	NCHW CHW-packed	<code>PSEUDO_HALF_CONFIG</code> <code>FLOAT_CONFIG</code>	Dilation: 1 for all dimensions

Filter descriptor <code>dwDesc: _NCHW</code>					
Algo Name	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
				DOUBLE_CONFIG	<p><code>convDesc</code> Group Count Support: Greater than 0</p> <p><code>dyDesc</code> width or height must equal 1 (the same dimension as in <code>xDesc</code>). The other dimension must be less than or equal to 256, meaning, the largest 1D tile size currently supported.</p> <p><code>convDesc</code> vertical and horizontal filter stride must equal 1</p> <p><code>dwDesc</code> filter height must be greater than <code>convDesc</code> zero-padding height</p> <p><code>dwDesc</code> filter width must be greater than <code>convDesc</code> zero-padding width</p>

Table 48. Supported Algorithms for `cudaConvolutionBackwardFilter()` 3D Convolutions:  
`dwDesc: _NCHW`

Filter descriptor <code>dwDesc: _NCHW</code>					
Algo Name (3D Convolutions)	Deterministic (Yes or No)	Tensor Formats Supported for <code>dyDesc</code>	Tensor Formats Supported for <code>dxDesc</code>	Data Type Configuration Supported	Important
<code>_ALGO_0</code>	No	All except <code>_NCDHW_VECT_C</code>	<ul style="list-style-type: none"> <li>▶ NCDHW CDHW-packed</li> <li>▶ NCDHW W-packed</li> <li>▶ NDHWC</li> </ul>	<code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code>	Dilation: greater than 0 for all dimensions <code>convDesc</code> Group Count Support: Greater than 0
<code>_ALGO_1</code>	No	All except <code>_NCDHW_VECT_C</code>	<ul style="list-style-type: none"> <li>▶ NCDHW CDHW-packed</li> <li>▶ NCDHW W-packed</li> <li>▶ NDHWC</li> </ul>	<code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code>	Dilation: greater than 0 for all dimensions <code>convDesc</code> Group Count Support: Greater than 0
<code>_ALGO_3</code>	No	NCDHW fully-packed	NCDHW fully-packed	<code>PSEUDO_HALF_CONFIG</code> <code>PSEUDO_BFLOAT16_CONFIG</code> <code>FLOAT_CONFIG</code> <code>DOUBLE_CONFIG</code>	Dilation: greater than 0 for all dimensions <code>convDesc</code> Group Count Support: Greater than 0

Table 49. Supported Algorithms for `cudannConvolutionBackwardFilter()` 3D Convolutions:  
`dwDesc: _NHWC`

Filter descriptor <code>dwDesc: _NHWC</code>					
Algo Name (3D Convolutions)	Deterministic (Yes or No)	Tensor Formats Supported for <code>xDesc</code>	Tensor Formats Supported for <code>dyDesc</code>	Data Type Configuration Supported	Important
<code>_ALGO_1</code>	Yes	NDHWC HWC-packed	NDHWC HWC-packed	PSEUDO_HALF_CONFIG PSEUDO_BFLOTT16_CONFIG FLOAT_CONFIG TRUE_HALF_CONFIG	Dilation: greater than 0 for all dimensions 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`, OR `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 `dwDesc` 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 object creation associated with the filter data.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 6.2.4. **cudaCreateFusedOpsConstParamPack ()**

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

```
cudaStatus_t cudaCreateFusedOpsConstParamPack(
    cudaFusedOpsConstParamPack_t *constPack,
    cudaFusedOps_t ops);
```

### Parameters

**constPack**

*Input.* The opaque structure that is created by this function. For more information, refer to [cudaFusedOpsConstParamPack\\_t](#).

**ops**

*Input.* The specific sequence of computations to perform in the `cudaFusedOps` computations, as defined in the enumerant type [cudaFusedOps\\_t](#).

### Returns

**CUDNN\_STATUS\_BAD\_PARAM**

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

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

**CUDNN\_STATUS\_SUCCESS**

If the descriptor is created successfully.

## 6.2.5. **cudaCreateFusedOpsPlan ()**

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

```
cudaStatus_t cudaCreateFusedOpsPlan(
    cudaFusedOpsPlan_t *plan,
    cudaFusedOps_t ops);
```

### Parameters

**plan**

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

**ops**

*Input.* The specific sequence of fused operations computations for which this plan descriptor should be created. For more information, refer to [cudaFusedOps\\_t](#).



## Returns

### **CUDNN\_STATUS\_BAD\_PARAM**

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

### **CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### **CUDNN\_STATUS\_SUCCESS**

The plan descriptor is created successfully.

## 6.2.6. `cudnnCreateFusedOpsVariantParamPack()`

This function creates the variant pack descriptor for the `cudnnFusedOps` computation.

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

## Parameters

### **varPack**

*Input.* Pointer to the descriptor created by this function. For more information, refer to [cudnnFusedOpsVariantParamPack\\_t](#).

### **ops**

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

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The descriptor is successfully created.

### **CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

### **CUDNN\_STATUS\_BAD\_PARAM**

If any input is invalid.

## 6.2.7. `cudnnDestroyFusedOpsConstParamPack()`

This function destroys a previously-created [cudnnFusedOpsConstParamPack\\_t](#) structure.

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

## Parameters

### **constPack**

*Input.* The [cudnnFusedOpsConstParamPack\\_t](#) structure that should be destroyed.

## Returns

### CUDNN\_STATUS\_SUCCESS

If the descriptor is destroyed successfully.

### CUDNN\_STATUS\_INTERNAL\_ERROR

If the ops enum value is not supported or invalid.

## 6.2.8. cudnnDestroyFusedOpsPlan()

This function destroys the plan descriptor provided.

```
cudaStatus_t cudnnDestroyFusedOpsPlan(
    cudaFusedOpsPlan_t plan);
```

## Parameters

### plan

*Input.* The descriptor that should be destroyed by this function.

## Returns

### CUDNN\_STATUS\_SUCCESS

If either the plan descriptor is `NULL` or the descriptor is successfully destroyed.

## 6.2.9. cudnnDestroyFusedOpsVariantParamPack()

This function destroys a previously-created descriptor for `cudaFusedOps` constant parameters.

```
cudaStatus_t cudnnDestroyFusedOpsVariantParamPack(
    cudaFusedOpsVariantParamPack_t varPack);
```

## Parameters

### varPack

*Input.* The descriptor that should be destroyed.

## Returns

### CUDNN\_STATUS\_SUCCESS

The descriptor is successfully destroyed.

## 6.2.10. cudnnFindConvolutionBackwardFilterAlgorithm()

This function attempts all algorithms available for [cudnnConvolutionBackwardFilter\(\)](#). It will attempt both the provided `convDesc` `mathType` and `CUDNN_DEFAULT_MATH` (assuming the two differ).

```
cudaStatus_t cudnnFindConvolutionBackwardFilterAlgorithm(
    cudaHandle_t          handle,
    const cudaTensorDescriptor_t xDesc,
    const cudaTensorDescriptor_t dyDesc,
    const cudaConvolutionDescriptor_t convDesc,
```

```

const cudnnFilterDescriptor_t      dwDesc,
const int                          requestedAlgoCount,
int                                *returnedAlgoCount,
cudnnConvolutionBwdFilterAlgoPerf_t *perfResults)

```



Note: Algorithms without the `CUDNN_TENSOR_OP_MATH` availability will only be tried with `CUDNN_DEFAULT_MATH`, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of `cudnnConvolutionBwdFilterAlgoPerf_t`. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudnnGetConvolutionBackwardFilterAlgorithmMaxCount()`.



Note:

- ▶ This function is host blocking.
- ▶ It is recommended to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **xDesc**

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

### **dyDesc**

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

### **convDesc**

*Input.* Previously initialized convolution descriptor.

### **dwDesc**

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

### **requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

### **returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

### **perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The query was successful.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is not allocated properly.

- ▶ `xDesc`, `dyDesc`, or `dwDesc` are 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.

## 6.2.11. `cudaFindConvolutionBackwardFilterAlgorithmEx`

This function attempts all algorithms available for `cudaConvolutionBackwardFilter()`. It will attempt both the provided `convDesc` `mathType` and `CUDNN_DEFAULT_MATH` (assuming the two differ).

```

cudaStatus_t cudaFindConvolutionBackwardFilterAlgorithmEx(
    cudaHandle_t          handle,
    const cudaTensorDescriptor_t xDesc,
    const void           *x,
    const cudaTensorDescriptor_t dyDesc,
    const void           *dy,
    const cudaConvolutionDescriptor_t convDesc,
    const cudaFilterDescriptor_t dwDesc,
    void                 *dw,
    const int             requestedAlgoCount,
    int                  *returnedAlgoCount,
    cudaConvolutionBwdFilterAlgoPerf_t *perfResults,
    void                 *workSpace,
    size_t                workSpaceSizeInBytes)
    
```

**Note:** Algorithms without the `CUDNN_TENSOR_OP_MATH` availability will only be tried with `CUDNN_DEFAULT_MATH`, and returned as such.

Memory is allocated via `cudaMalloc()`. The performance metrics are returned in the user-allocated array of `cudaConvolutionBwdFilterAlgoPerf_t`. These metrics are written in a sorted fashion where the first element has the lowest compute time. The total number of resulting algorithms can be queried through the API `cudaGetConvolutionBackwardFilterAlgorithmMaxCount()`.

**Note:** 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 is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A `NIL` pointer is considered a `workSpace` of 0 bytes.

**workSpaceSizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ `handle` is not allocated properly.
- ▶ `xDesc`, `dyDesc`, or `dwDesc` are 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.

## 6.2.12. **cudnnFusedOpsExecute()**

This function executes the sequence of `cudnnFusedOps` operations.

```
cudnnStatus_t cudnnFusedOpsExecute(
    cudnnHandle_t handle,
    const cudnnFusedOpsPlan_t plan,
    cudnnFusedOpsVariantParamPack_t varPack);
```

### Parameters

**handle**

*Input.* Pointer to the cuDNN library context.

**plan**

*Input.* Pointer to a previously-created and initialized plan descriptor.

**varPack**

*Input.* Pointer to the descriptor to the variant parameters pack.

### Returns

**CUDNN\_STATUS\_BAD\_PARAM**

If the type of `cudnnFusedOps_t` in the plan descriptor is unsupported.

## 6.2.13. **cudnnGetConvolutionBackwardFilterAlgorithmMaxCount()**

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.

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithmMaxCount(
    cudnnHandle_t handle,
    int *count)
```

### 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.

## 6.2.14. cudnnGetConvolutionBackwardFilterAlgorithm\_v7

This function serves as a heuristic for obtaining the best suited algorithm for [cudnnConvolutionBackwardFilter\(\)](#) for the given layer specifications. This function will return all algorithms (including CUDNN\_TENSOR\_OP\_MATH and CUDNN\_DEFAULT\_MATH versions of algorithms where CUDNN\_TENSOR\_OP\_MATH may be available) sorted by expected (based on internal heuristic) relative performance with fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, use [cudnnFindConvolutionBackwardFilterAlgorithm\(\)](#). The total number of resulting algorithms can be queried through the returnedAlgoCount variable.

```

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)
    
```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### xDesc

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

### dyDesc

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

### convDesc

*Input.* Previously initialized convolution descriptor.

### dwDesc

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

### requestedAlgoCount

*Input.* The maximum number of elements to be stored in perfResults.

### returnedAlgoCount

*Output.* The number of output elements stored in perfResults.

### perfResults

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

## Returns

### CUDNN\_STATUS\_SUCCESS

The query was successful.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ One of the parameters `handle`, `xDesc`, `dyDesc`, `convDesc`, `dwDesc`, `perfResults`, or `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.

## 6.2.15. cudnnGetConvolutionBackwardFilterWorkspaceSize

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_v7()` 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.

```

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)
    
```

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### xDesc

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

### dyDesc

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

### convDesc

*Input.* Previously initialized convolution descriptor.

### dwDesc

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

### algo

*Input.* Enumerant that specifies the chosen convolution algorithm.



**sizeInBytes**

*Output.* Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified `algo`.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The numbers of feature maps of the input tensor and output tensor differ.
- ▶ The `dataType` of the two tensor descriptors or the filter are different.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

## 6.2.16. `cudaGetFusedOpsConstParamPackAttribute()`

This function retrieves the values of the descriptor pointed to by the `param` pointer input. The type of the descriptor is indicated by the enum value of `paramLabel` input.

```
cudaStatus_t cudaGetFusedOpsConstParamPackAttribute(
    const cudaFusedOpsConstParamPack_t constPack,
    cudaFusedOpsConstParamLabel_t paramLabel,
    void *param,
    int *isNULL);
```

**Parameters**

**constPack**

*Input.* The opaque `cudaFusedOpsConstParamPack_t` structure that contains the various problem size information, such as the shape, layout and the type of tensors, and the descriptors for convolution and activation, for the selected sequence of `cudaFusedOps_t` computations.

**paramLabel**

*Input.* Several types of descriptors can be retrieved by this getter function. The `param` input points to the descriptor itself, and this input indicates the type of the descriptor pointed to by the `param` input. The `cudaFusedOpsConstParamLabel_t` enumerant type enables the selection of the type of the descriptor. Refer to the `param` description below.

**param**

*Input.* Data pointer to the host memory associated with the descriptor that should be retrieved. The type of this descriptor depends on the value of `paramLabel`. For the given `paramLabel`, if the associated value inside the `constPack` is set to `NULL` or by default `NULL`, then cuDNN will copy the value or the opaque structure in the `constPack` to the host memory buffer pointed to by `param`. For more information, see the table in `cudaFusedOpsConstParamLabel_t`.

**isNULL**

*Input/Output.* Users must pass a pointer to an integer in the host memory in this field. If the value in the `constPack` associated with the given `paramLabel` is by default `NULL` or previously set by the user to `NULL`, then cuDNN will write a non-zero value to the location pointed by `isNULL`.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The descriptor values are retrieved successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

If either `constPack`, `param` or `isNULL` is `NULL`; or if `paramLabel` is invalid.

## 6.2.17. **cudaGetFusedOpsVariantParamPackAttribute()**

This function retrieves the settings of the variable parameter pack descriptor.

```
cudaStatus_t cudaGetFusedOpsVariantParamPackAttribute(
    const cudaFusedOpsVariantParamPack_t varPack,
    cudaFusedOpsVariantParamLabel_t paramLabel,
    void *ptr);
```

**Parameters**

**varPack**

*Input.* Pointer to the `cudaFusedOps` variant parameter pack (`varPack`) descriptor.

**paramLabel**

*Input.* Type of the buffer pointer parameter (in the `varPack` descriptor). For more information, refer to [cudaFusedOpsConstParamLabel\\_t](#). The retrieved descriptor values vary according to this type.

**ptr**

*Output.* Pointer to the host or device memory where the retrieved value is written by this function. The data type of the pointer, and the host/device memory location, depend on the `paramLabel` input selection. For more information, refer to [cudaFusedOpsVariantParamLabel\\_t](#).

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The descriptor values are retrieved successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

If either `varPack` or `ptr` is `NULL`, or if `paramLabel` is set to invalid value.

## 6.2.18. **cudaMakeFusedOpsPlan()**

This function determines the optimum kernel to execute, and the workspace size the user should allocate, prior to the actual execution of the fused operations by [cudaFusedOpsExecute\(\)](#).

```
cudaStatus_t cudaMakeFusedOpsPlan(
```

```

    cudnnHandle_t handle,
    cudnnFusedOpsPlan_t plan,
    const cudnnFusedOpsConstParamPack_t constPack,
    size_t *workspaceSizeInBytes);

```

## Parameters

### **handle**

*Input.* Pointer to the cuDNN library context.

### **plan**

*Input.* Pointer to a previously-created and initialized plan descriptor.

### **constPack**

*Input.* Pointer to the descriptor to the const parameters pack.

### **workspaceSizeInBytes**

*Output.* The amount of workspace size the user should allocate for the execution of this plan.

## Returns

### **CUDNN\_STATUS\_BAD\_PARAM**

If any of the inputs is `NULL`, or if the type of `cudnnFusedOps_t` in the `constPack` descriptor is unsupported.

### **CUDNN\_STATUS\_SUCCESS**

The function executed successfully.

## 6.2.19. `cudnnSetFusedOpsConstParamPackAttribute()`

This function sets the descriptor pointed to by the `param` pointer input. The type of the descriptor to be set is indicated by the enum value of the `paramLabel` input.

```

cudnnStatus_t cudnnSetFusedOpsConstParamPackAttribute(
    cudnnFusedOpsConstParamPack_t constPack,
    cudnnFusedOpsConstParamLabel_t paramLabel,
    const void *param);

```

## Parameters

### **constPack**

*Input.* The opaque `cudnnFusedOpsConstParamPack_t` structure that contains the various problem size information, such as the shape, layout and the type of tensors, the descriptors for convolution and activation, and settings for operations such as convolution and activation.

### **paramLabel**

*Input.* Several types of descriptors can be set by this setter function. The `param` input points to the descriptor itself, and this input indicates the type of the descriptor pointed to by the `param` input. The `cudnnFusedOpsConstParamLabel_t` enumerant type enables the selection of the type of the descriptor.

**param**

*Input.* Data pointer to the host memory, associated with the specific descriptor. The type of the descriptor depends on the value of `paramLabel`. For more information, refer to the table in [cudaFusedOpsConstParamLabel\\_t](#).

If this pointer is set to `NULL`, then the cuDNN library will record as such. If not, then the values pointed to by this pointer (meaning, the value or the opaque structure underneath) will be copied into the `constPack` during `cudaSetFusedOpsConstParamPackAttribute()` operation.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The descriptor is set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

If `constPack` is `NULL`, or if `paramLabel` or the ops setting for `constPack` is invalid.

**6.2.20. cudaSetFusedOpsVariantParamPackAttribute ()**

This function sets the variable parameter pack descriptor.

```
cudaStatus_t cudaSetFusedOpsVariantParamPackAttribute(
    cudaFusedOpsVariantParamPack_t varPack,
    cudaFusedOpsVariantParamLabel_t paramLabel,
    void *ptr);
```

**Parameters**

**varPack**

*Input.* Pointer to the `cudaFusedOps` variant parameter pack (`varPack`) descriptor.

**paramLabel**

*Input.* Type to which the buffer pointer parameter (in the `varPack` descriptor) is set by this function. For more information, refer to [cudaFusedOpsConstParamLabel\\_t](#).

**ptr**

*Input.* Pointer, to the host or device memory, to the value to which the descriptor parameter is set. The data type of the pointer, and the host/device memory location, depend on the `paramLabel` input selection. For more information, refer to [cudaFusedOpsVariantParamLabel\\_t](#).

**Returns**

**CUDNN\_STATUS\_BAD\_PARAM**

If `varPack` is `NULL` or if `paramLabel` is set to an unsupported value.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was set successfully.

---

# Chapter 7. `cudaadv_infer.so` Library

This entity contains all other features and algorithms. This includes RNNs, CTC loss, and multi-head attention. The `cudaadv_infer` library depends on `cudaops_infer`.

## 7.1. Data Type References

These are the data type references in the `cudaadv_infer.so` library.

### 7.1.1. Pointer To Opaque Struct Types

These are the pointers to the opaque struct types in the `cudaadv_infer.so` library.

#### 7.1.1.1. `cudaAttnDescriptor_t`

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

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

Use the [`cudaCreateAttnDescriptor\(\)`](#) function to create an instance of the attention descriptor object and [`cudaDestroyAttnDescriptor\(\)`](#) to delete the previously created descriptor. Use the [`cudaSetAttnDescriptor\(\)`](#) function to configure the descriptor.

#### 7.1.1.2. `cudaPersistentRNNPlan_t`

This function is deprecated starting in cuDNN 8.0.0.

`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.

### 7.1.1.3. `cudaRNNDataDescriptor_t`

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

### 7.1.1.4. `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.

### 7.1.1.5. `cudaSeqDataDescriptor_t`

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

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

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

The `cudaSeqDataDescriptor_t` object holds the following parameters:

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

Use the `cudaCreateSeqDataDescriptor()` function to create one instance of the sequence data descriptor object and `cudaDestroySeqDataDescriptor()` to delete a previously created descriptor. Use the `cudaSetSeqDataDescriptor()` function to configure the descriptor.

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

## 7.1.2. Enumeration Types

These are the enumeration types in the `cuda_adv_infer.so` library.

### 7.1.2.1. `cudaDirectionMode_t`

`cudaDirectionMode_t` is an enumerated type used to specify the recurrence pattern in the `cudaRNNTForwardInference()`, `cudaRNNTForwardTraining()`, `cudaRNNTBackwardData()` and `cudaRNNTBackwardWeights()` routines.

#### Values

**CUDNN\_UNIDIRECTIONAL**

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

**CUDNN\_BIDIRECTIONAL**

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

### 7.1.2.2. `cudaForwardMode_t`

`cudaForwardMode_t` is an enumerated type to specify inference or training mode in RNN API. This parameter allows the cuDNN library to tune more precisely the size of the workspace buffer that could be different in inference and training regimens.

#### Values

**CUDNN\_FWD\_MODE\_INFERENCE**

Selects the inference mode.

**CUDNN\_FWD\_MODE\_TRAINING**

Selects the training mode.

### 7.1.2.3. `cudaMultiHeadAttnWeightKind_t`

`cudaMultiHeadAttnWeightKind_t` is an enumerated type that specifies a group of weights or biases in the `cudaGetMultiHeadAttnWeights()` function.

#### Values

**CUDNN\_MH\_ATTN\_Q\_WEIGHTS**

Selects the input projection weights for queries.

**CUDNN\_MH\_ATTN\_K\_WEIGHTS**

Selects the input projection weights for keys.

**CUDNN\_MH\_ATTN\_V\_WEIGHTS**

Selects the input projection weights for values.

**CUDNN\_MH\_ATTN\_O\_WEIGHTS**

Selects the output projection weights.

**CUDNN\_MH\_ATTN\_Q\_BIASES**

Selects the input projection biases for queries.

**CUDNN\_MH\_ATTN\_K\_BIASES**

Selects the input projection biases for keys.

**CUDNN\_MH\_ATTN\_V\_BIASES**

Selects the input projection biases for values.

**CUDNN\_MH\_ATTN\_O\_BIASES**

Selects the output projection biases.

### 7.1.2.4. **cudaRNNBiasMode\_t**

cudaRNNBiasMode\_t is an enumerated type used to specify the number of bias vectors for RNN functions. Refer to the description of the [cudaRNNMode\\_t](#) enumerated type for the equations for each cell type based on the bias mode.

#### Values

**CUDNN\_RNN\_NO\_BIAS**

Applies RNN cell formulas that do not use biases.

**CUDNN\_RNN\_SINGLE\_INP\_BIAS**

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

**CUDNN\_RNN\_DOUBLE\_BIAS**

Applies RNN cell formulas that use two bias vectors.

**CUDNN\_RNN\_SINGLE\_REC\_BIAS**

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

### 7.1.2.5. **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.

### 7.1.2.6. **cudaRNNDataLayout\_t**

cudaRNNDataLayout\_t is an enumerated type used to select the RNN data layout. It is used in the API calls [cudaGetRNNDataDescriptor\(\)](#) and [cudaSetRNNDataDescriptor\(\)](#).

#### Values

**CUDNN\_RNN\_DATA\_LAYOUT\_SEQ\_MAJOR\_UNPACKED**

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

**CUDNN\_RNN\_DATA\_LAYOUT\_SEQ\_MAJOR\_PACKED**

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



**CUDNN\_RNN\_DATA\_LAYOUT\_BATCH\_MAJOR\_UNPACKED**

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

### 7.1.2.7. cudnnRNNInputMode\_t

cudnnRNNInputMode\_t is an enumerated type used to specify the behavior of the first layer in the [cudnnRNNForwardInference\(\)](#), [cudnnRNNForwardTraining\(\)](#), [cudnnRNNBackwardData\(\)](#) and [cudnnRNNBackwardWeights\(\)](#) routines.

#### Values

**CUDNN\_LINEAR\_INPUT**

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

**CUDNN\_SKIP\_INPUT**

No operation is performed at the input of the first recurrent layer. If

CUDNN\_SKIP\_INPUT is used the leading dimension of the input tensor must be equal to the hidden state size of the network.

### 7.1.2.8. cudnnRNNMode\_t

cudnnRNNMode\_t is an enumerated type used to specify the type of network used in the [cudnnRNNForwardInference](#), [cudnnRNNForwardTraining](#), [cudnnRNNBackwardData](#) and [cudnnRNNBackwardWeights](#) routines.

#### Values

**CUDNN\_RNN\_RELU**

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

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

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

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

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

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

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

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

### CUDNN\_RNN\_TANH

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

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

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

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

If `cudaRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS` or `CUDNN_RNN_SINGLE_REC_BIAS`, then the following equation with bias  $b$  applies:

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

If `cudaRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_NO_BIAS`, then the following equation applies:

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

### CUDNN\_LSTM

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

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

In addition, the following applies:

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

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

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

$$f_t = \sigma(W_f x_t + R_f h_{t-1} + b_{Wf} + b_{Rf})$$

$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_{Wo} + b_{Ro})$$

$$c'_t = \tanh(W_c x_t + R_c h_{t-1} + b_{Wc} + b_{Rc})$$

$$c_t = f_t \circ c_{t-1} + i_t \circ c'_t$$

$$h_t = o_t \circ \tanh(c_t)$$

If `cudaadvRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS` or `CUDNN_RNN_SINGLE_REC_BIAS`, then the following equations with bias  $b$  apply:

$$i_t = \sigma (W_i x_t + R_i h_{t-1} + b_i)$$

$$f_t = \sigma (W_f x_t + R_f h_{t-1} + b_f)$$

$$o_t = \sigma (W_o x_t + R_o h_{t-1} + b_o)$$

$$c'_t = \tanh (W_c x_t + R_c h_{t-1} + b_c)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ c'_t$$

$$h_t = o_t \circ \tanh (c_t)$$

If `cudaadvRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_NO_BIAS`, then the following equations apply:

$$i_t = \sigma (W_i x_t + R_i h_{t-1})$$

$$f_t = \sigma (W_f x_t + R_f h_{t-1})$$

$$o_t = \sigma (W_o x_t + R_o h_{t-1})$$

$$c'_t = \tanh (W_c x_t + R_c h_{t-1})$$

$$c_t = f_t \circ c_{t-1} + i_t \circ c'_t$$

$$h_t = o_t \circ \tanh (c_t)$$

#### CUDNN\_GRU

A three-gate network consisting of Gated Recurrent Units.

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

In addition, the following applies:

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

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

$$i_t = \sigma (W_i x_t + R_i h_{t-1} + b_{Wi} + b_{Ri})$$

$$r_t = \sigma (W_r x_t + R_r h_{t-1} + b_{Wr} + b_{Rr})$$

$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1} + b_{RH}) + b_{WH})$$

$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

If `cudaRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS`, then the following equations with bias  $b$  apply:

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i)$$

$$r_t = \sigma(W_r x_t + R_r h_{t-1} + b_r)$$

$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1}) + b_{WH})$$

$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

If `cudaRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_REC_BIAS`, then the following equations with bias  $b$  apply:

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i)$$

$$r_t = \sigma(W_r x_t + R_r h_{t-1} + b_r)$$

$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1} + b_{RH}))$$

$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

If `cudaRNNBiasMode_t biasMode` in `rnnDesc` is `CUDNN_RNN_NO_BIAS`, then the following equations apply:

$$i_t = \sigma(W_i x_t + R_i h_{t-1})$$

$$r_t = \sigma(W_r x_t + R_r h_{t-1})$$

$$h'_t = \tanh(W_h x_t + r_t \circ (R_h h_{t-1}))$$

$$h_t = (1 - i_t) \circ h'_t + i_t \circ h_{t-1}$$

### 7.1.2.9. `cudaRNNPaddingMode_t`

`cudaRNNPaddingMode_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.

### 7.1.2.10. cudnnSeqDataAxis\_t

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

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

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

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

#### Values

**CUDNN\_SEQDATA\_TIME\_DIM**

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

**CUDNN\_SEQDATA\_BATCH\_DIM**

Identifies the `BATCH` dimension or specifies the `BATCH` in the data layout.

**CUDNN\_SEQDATA\_BEAM\_DIM**

Identifies the `BEAM` dimension or specifies the `BEAM` in the data layout.

**CUDNN\_SEQDATA\_VECT\_DIM**

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

## 7.2. API Functions

These are the API functions in the `cudaadvinfer.so` library.

### 7.2.1. cudnnAdvInferVersionCheck ()

This function checks to see whether the version of the AdvInfer subset of the library is consistent with the other sub-libraries.

```
cudnnStatus_t cudnnAdvInferVersionCheck(void)
```

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The version is consistent with other sub-libraries.

**CUDNN\_STATUS\_VERSION\_MISMATCH**

The version of AdvInfer is not consistent with other sub-libraries. Users should check the installation and make sure all sub-component versions are consistent.

## 7.2.2. cudnnBuildRNNDynamic()

This function compiles the RNN persistent code using CUDA runtime compilation library (NVRTC) when the `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` algo is selected. The code is tailored to the current GPU and specific hyperparameters (`miniBatch`). This call is expected to be expensive in terms of runtime and should be invoked infrequently. Note that the `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` algo does not support variable length sequences within the batch.

```

cudnnStatus_t cudnnBuildRNNDynamic(
    cudnnHandle_t handle,
    cudnnRNNDescriptor_t rnnDesc,
    int32_t miniBatch);

```

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**miniBatch**

*Input.* The exact number of sequences in a batch.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The code was built and linked successfully.

**CUDNN\_STATUS\_MAPPING\_ERROR**

A GPU/CUDA resource, such as a texture object, shared memory, or zero-copy memory is not available in the required size or there is a mismatch between the user resource and cuDNN internal resources. A resource mismatch may occur, for example, when calling `cudnnSetStream()`. There could be a mismatch between the user provided CUDA stream and the internal CUDA events instantiated in the cuDNN handle when `cudnnCreate()` was invoked.

This error status may not be correctable when it is related to texture dimensions, shared memory size, or zero-copy memory availability. If `CUDNN_STATUS_MAPPING_ERROR` is returned by `cudnnSetStream()`, then it is typically correctable, however, it means that the cuDNN handle was created on one GPU and the user stream passed to this function is associated with another GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The resources could not be allocated.

**CUDNN\_STATUS\_RUNTIME\_PREREQUISITE\_MISSING**

The prerequisite runtime library could not be found.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The current hyper-parameters are invalid.

### 7.2.3. `cudaCreateAttnDescriptor()`

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

```
cudaStatus_t cudaCreateAttnDescriptor(cudaAttnDescriptor_t *attnDesc);
```

Use the [cudaSetAttnDescriptor\(\)](#) function to configure the attention descriptor and [cudaDestroyAttnDescriptor\(\)](#) to destroy it and release the allocated memory.

#### Parameters

**attnDesc**

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

#### Returns

**CUDNN\_STATUS\_SUCCESS**

The descriptor object was created successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

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

**CUDNN\_STATUS\_ALLOC\_FAILED**

The memory allocation failed.

### 7.2.4. `cudaCreatePersistentRNNPlan()`

This function has been deprecated in cuDNN 8.0. Use [cudaBuildRNNDynamic\(\)](#) instead of `cudaCreatePersistentRNNPlan()`.

```
cudaStatus_t cudaCreatePersistentRNNPlan(
    cudaRNNDescriptor_t      rnnDesc,
    const int                minibatch,
    const cudaDataType_t     dataType,
    cudaPersistentRNNPlan_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 RNN model hyperparameters. This function call is expected to be expensive in terms of runtime and should be used infrequently. However, the user must invoke `cudaCreatePersistentRNNPlan()` every time the number of input vectors changes in a minibatch. For more information, refer to [cudaRNNDescriptor\\_t](#), [cudaDataType\\_t](#), and [cudaPersistentRNNPlan\\_t](#).

#### Parameters

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**minibatch**

*Input.* The exact number of vectors in a batch.

**dataType**

*Input.* Specifies data type for RNN weights/biases and input and output data.

**plan**

*Output.* Pointer to where the address to the newly created RNN persistent plan should be written.

**Returns****CUDNN\_STATUS\_SUCCESS**

The object was created successfully.

**CUDNN\_STATUS\_MAPPING\_ERROR**

A GPU/CUDA resource, such as a texture object, shared memory, or zero-copy memory is not available in the required size or there is a mismatch between the user resource and cuDNN internal resources. A resource mismatch may occur, for example, when calling `cudnnSetStream()`. There could be a mismatch between the user provided CUDA stream and the internal CUDA events instantiated in the cuDNN handle when `cudnnCreate()` was invoked.

This error status may not be correctable when it is related to texture dimensions, shared memory size, or zero-copy memory availability. If `CUDNN_STATUS_MAPPING_ERROR` is returned by `cudnnSetStream()`, then it is typically correctable, however, it means that the cuDNN handle was created on one GPU and the user stream passed to this function is associated with another GPU.

**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.

**7.2.5. cudnnCreateRNNDataDescriptor()**

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

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

**Parameters****RNNDataDesc**

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



## Returns

### CUDNN\_STATUS\_SUCCESS

The RNN data descriptor object was created successfully.

### CUDNN\_STATUS\_BAD\_PARAM

The `RNNDataDesc` argument is `NULL`.

### CUDNN\_STATUS\_ALLOC\_FAILED

The resources could not be allocated.

## 7.2.6. `cudnnCreateRNNDescriptor()`

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

```

cudnnStatus_t cudnnCreateRNNDescriptor(
    cudnnRNNDescriptor_t *rnnDesc)

```

## Parameters

### `rnnDesc`

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was created successfully.

### CUDNN\_STATUS\_BAD\_PARAM

The `rnnDesc` argument is `NULL`.

### CUDNN\_STATUS\_ALLOC\_FAILED

The resources could not be allocated.

## 7.2.7. `cudnnCreateSeqDataDescriptor()`

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

```

cudnnStatus_t cudnnCreateSeqDataDescriptor(cudnnSeqDataDescriptor_t *seqDataDesc);

```

Use the [cudnnSetSeqDataDescriptor\(\)](#) function to configure the sequence data descriptor and [cudnnDestroySeqDataDescriptor\(\)](#) to destroy it and release the allocated memory.

## Parameters

### `seqDataDesc`

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The descriptor object was created successfully.

### CUDNN\_STATUS\_BAD\_PARAM

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

### CUDNN\_STATUS\_ALLOC\_FAILED

The memory allocation failed.

## 7.2.8. cudnnDestroyAttnDescriptor()

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

```
cudaStatus_t cudnnDestroyAttnDescriptor(cudaAttnDescriptor_t attnDesc);
```

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

## Parameters

### attnDesc

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The descriptor was destroyed successfully.

## 7.2.9. cudnnDestroyPersistentRNNPlan()

This function has been deprecated in cuDNN 8.0.

This function destroys a previously created persistent RNN plan object. Invoking `cudnnDestroyPersistentRNNPlan()` with the NULL argument is a no operation (NOP).

```
cudaStatus_t cudnnDestroyPersistentRNNPlan(
    cudaPersistentRNNPlan_t plan)
```

The `cudnnDestroyPersistentRNNPlan()` function is not able to detect if the `plan` argument holds a valid address. Undefined behavior will occur in cases of passing an invalid pointer, not returned by the `cudnnCreatePersistentRNNPlan()` function, or in the double deletion scenario of a valid address.

## Parameters

### plan

*Input.* Pointer to the RNN persistent plan object to be destroyed.

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was destroyed successfully.

## 7.2.10. cudnnDestroyRNNDataDescriptor()

This function destroys a previously created RNN data descriptor object. Invoking `cudnnDestroyRNNDataDescriptor()` with the `NULL` argument is a no operation (NOP).

```
cudnnStatus_t cudnnDestroyRNNDataDescriptor(
    cudnnRNNDataDescriptor_t RNNDataDesc)
```

The `cudnnDestroyRNNDataDescriptor()` function is not able to detect if the `RNNDataDesc` argument holds a valid address. Undefined behavior will occur in cases of passing an invalid pointer, not returned by the `cudnnCreateRNNDataDescriptor()` function, or in the double deletion scenario of a valid address.

## Parameters

### RNNDataDesc

*Input.* Pointer to the RNN data descriptor object to be destroyed.

## Returns

### CUDNN\_STATUS\_SUCCESS

The RNN data descriptor object was destroyed successfully.

## 7.2.11. cudnnDestroyRNNDescriptor()

This function destroys a previously created RNN descriptor object. Invoking `cudnnDestroyRNNDescriptor()` with the `NULL` argument is a no operation (NOP).

```
cudnnStatus_t cudnnDestroyRNNDescriptor(
    cudnnRNNDescriptor_t rnnDesc)
```

The `cudnnDestroyRNNDescriptor()` function is not able to detect if the `rnnDesc` argument holds a valid address. Undefined behavior will occur in cases of passing an invalid pointer, not returned by the `cudnnCreateRNNDescriptor()` function, or in the double deletion scenario of a valid address.

## Parameters

### rnnDesc

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

## Returns

### CUDNN\_STATUS\_SUCCESS

The object was destroyed successfully.

## 7.2.12. cudnnDestroySeqDataDescriptor()

This function destroys the sequence data descriptor object and releases its memory. The seqDataDesc argument can be NULL. Invoking cudnnDestroySeqDataDescriptor() with a NULL argument is a no operation (NOP).

```
cudnnStatus_t cudnnDestroySeqDataDescriptor(cudnnSeqDataDescriptor_t seqDataDesc);
```

The cudnnDestroySeqDataDescriptor() function is not able to detect if the seqDataDesc argument holds a valid address. Undefined behavior will occur in case of passing an invalid pointer, not returned by the [cudnnCreateSeqDataDescriptor\(\)](#) function, or in the double deletion scenario of a valid address.

### Parameters

#### seqDataDesc

*Input.* Pointer to the sequence data descriptor object to be destroyed.

### Returns

#### CUDNN\_STATUS\_SUCCESS

The descriptor was destroyed successfully.

## 7.2.13. cudnnFindRNNForwardInferenceAlgorithmEx()

This function has been deprecated in cuDNN 8.0.

This function attempts all available cuDNN algorithms for [cudnnRNNForwardInference\(\)](#), using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of cudnnAlgorithmPerformance\_t. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

```
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)
```

## 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 the [cudnnGetRNNWorkspaceSize\(\)](#) function for querying the workspace size required to execute the RNN.

### **xDesc**

*Input.* An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element  $n$  to element  $n+1$  but may not increase. Each tensor descriptor must have the same second dimension (vector length).

### **x**

*Input.* Data pointer to GPU memory associated with the tensor descriptors in the array `xDesc`. The data are expected to be packed contiguously with the first element of iteration  $n+1$  following directly from the last element of iteration  $n$ .

### **hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument.

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 0.0, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- ▶ Setting `findIntensity` within the range  $[-1, 0)$  sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- ▶ Values outside the range  $[-1, 1]$  are truncated to the range  $[-1, 1]$ , and then interpreted as per the above.
- ▶ Setting `findIntensity` to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.

- ▶ This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

**returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

**perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `cxDesc`, `wDesc`, `hyDesc` or `cyDesc`, or one of the descriptors in `xDesc` or `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `cxDesc`, `wDesc`, `yDesc`, `hyDesc` or `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.

## 7.2.14. cudnnGetAttnDescriptor()

This function retrieves settings from the previously created attention descriptor. The user can assign `NULL` to any pointer except `attnDesc` when the retrieved value is not needed.

```
cudnnStatus_t cudnnGetAttnDescriptor(
    cudnnAttnDescriptor_t attnDesc,
    unsigned *attnMode,
    int *nHeads,
    double *smScaler,
    cudnnDataType_t *dataType,
```



```

    cudnnDataType_t *computePrec,
    cudnnMathType_t *mathType,
    cudnnDropoutDescriptor_t *attnDropoutDesc,
    cudnnDropoutDescriptor_t *postDropoutDesc,
    int *qSize,
    int *kSize,
    int *vSize,
    int *qProjSize,
    int *kProjSize,
    int *vProjSize,
    int *oProjSize,
    int *qoMaxSeqLength,
    int *kvMaxSeqLength,
    int *maxBatchSize,
    int *maxBeamSize);

```

## Parameters

### **attnDesc**

*Input.* Attention descriptor.

### **attnMode**

*Output.* Pointer to the storage for binary attention flags.

### **nHeads**

*Output.* Pointer to the storage for the number of attention heads.

### **smScaler**

*Output.* Pointer to the storage for the softmax smoothing/sharpening coefficient.

### **dataType**

*Output.* Data type for attention weights, sequence data inputs, and outputs.

### **computePrec**

*Output.* Pointer to the storage for the compute precision.

### **mathType**

*Output.* NVIDIA Tensor Core settings.

### **attnDropoutDesc**

*Output.* Descriptor of the dropout operation applied to the softmax output.

### **postDropoutDesc**

*Output.* Descriptor of the dropout operation applied to the multi-head attention output.

### **qSize, kSize, vSize**

*Output.* Q, K, and V embedding vector lengths.

### **qProjSize, kProjSize, vProjSize**

*Output.* Q, K, and V embedding vector lengths after input projections.

### **oProjSize**

*Output.* Pointer to store the output vector length after projection.

### **qoMaxSeqLength**

*Output.* Largest sequence length expected in sequence data descriptors related to Q, O, dQ, dO inputs and outputs.

### **kvMaxSeqLength**

*Output.* Largest sequence length expected in sequence data descriptors related to K, V, dK, dV inputs and outputs.

**maxBatchSize**

*Output.* Largest batch size expected in the [cudaSeqDataDescriptor\\_t](#) container.

**maxBeamSize**

*Output.* Largest beam size expected in the [cudaSeqDataDescriptor\\_t](#) container.

Returns

**CUDNN\_STATUS\_SUCCESS**

Requested attention descriptor fields were retrieved successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was found.

## 7.2.15. [cudaGetMultiHeadAttnBuffers\(\)](#)


This function computes weight, work, and reserve space buffer sizes used by the following functions:

- ▶ [cudaMultiHeadAttnForward\(\)](#)
- ▶ [cudaMultiHeadAttnBackwardData\(\)](#)
- ▶ [cudaMultiHeadAttnBackwardWeights\(\)](#)

```

cudaStatus_t cudaGetMultiHeadAttnBuffers (
    cudaHandle_t handle,
    const cudaAttnDescriptor_t attnDesc,
    size_t *weightSizeInBytes,
    size_t *workSpaceSizeInBytes,
    size_t *reserveSpaceSizeInBytes);
    
```

Assigning NULL to the `reserveSpaceSizeInBytes` argument indicates that the user does not plan to invoke multi-head attention gradient functions: [cudaMultiHeadAttnBackwardData\(\)](#) and [cudaMultiHeadAttnBackwardWeights\(\)](#). This situation occurs in the inference mode.



**Note:** NULL cannot be assigned to `weightSizeInBytes` and `workSpaceSizeInBytes` pointers.

The user must allocate weight, work, and reserve space buffer sizes in the GPU memory using `cudaMalloc()` with the reported buffer sizes. The buffers can be also carved out from a larger chunk of allocated memory but the buffer addresses must be at least 16B aligned.

The work-space buffer is used for temporary storage. Its content can be discarded or modified after all GPU kernels launched by the corresponding API complete. The reserve-space buffer is used to transfer intermediate results from [cudaMultiHeadAttnForward\(\)](#) to [cudaMultiHeadAttnBackwardData\(\)](#), and from [cudaMultiHeadAttnBackwardData\(\)](#) to [cudaMultiHeadAttnBackwardWeights\(\)](#). The content of the reserve-space buffer cannot be modified until all GPU kernels launched by the above three multi-head attention API functions finish.

All multi-head attention weight and bias tensors are stored in a single weight buffer. For speed optimizations, the cuDNN API may change tensor layouts and their relative

locations in the weight buffer based on the provided attention parameters. Use the [`cudaGetMultiHeadAttnWeights\(\)`](#) function to obtain the start address and the shape of each weight or bias tensor.

## Parameters

**handle**

*Input.* The current cuDNN context handle.

**attnDesc**

*Input.* Pointer to a previously initialized attention descriptor.

**weightSizeInBytes**

*Output.* Minimum buffer size required to store all multi-head attention trainable parameters.

**workspaceSizeInBytes**

*Output.* Minimum buffer size required to hold all temporary surfaces used by the forward and gradient multi-head attention API calls.

**reserveSpaceSizeInBytes**

*Output.* Minimum buffer size required to store all intermediate data exchanged between forward and backward (gradient) multi-head attention functions. Set this parameter to `NULL` in the inference mode indicating that gradient API calls will not be invoked.

## Returns

**CUDNN\_STATUS\_ARCH\_MISMATCH**

The GPU device does not support the input data type.

**CUDNN\_STATUS\_SUCCESS**

The requested buffer sizes were computed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was found.

## 7.2.16. `cudaGetMultiHeadAttnWeights()`

This function obtains the shape of the weight or bias tensor. It also retrieves the start address of tensor data located in the `weight` buffer. Use the `wKind` argument to select a particular tensor. For more information, refer to [`cudaMultiHeadAttnWeightKind\_t`](#) for the description of the enumerant type.

```
cudaStatus_t cudaGetMultiHeadAttnWeights(
    cudaHandle_t handle,
    const cudaAttnDescriptor_t attnDesc,
    cudaMultiHeadAttnWeightKind_t wKind,
    size_t weightSizeInBytes,
    const void *weights,
    cudaTensorDescriptor_t wDesc,
    void **wAddr);
```

Biases are used in the input and output projections when the `CUDNN_ATTN_ENABLE_PROJ_BIASES` flag is set in the attention descriptor. Refer to [`cudaSetAttnDescriptor\(\)`](#) for the description of flags to control projection biases.

When the corresponding weight or bias tensor does not exist, the function writes `NULL` to the storage location pointed by `wAddr` and returns zeros in the `wDesc` tensor descriptor. The return status of the `cudaGetMultiHeadAttnWeights()` function is `CUDNN_STATUS_SUCCESS` in this case.

The cuDNN `multiHeadAttention` sample code demonstrates how to access multi-head attention weights. Although the buffer with weights and biases should be allocated in the GPU memory, the user can copy it to the host memory and invoke the `cudaGetMultiHeadAttnWeights()` function with the host weights address to obtain tensor pointers in the host memory. This scheme allows the user to inspect trainable parameters directly in the CPU memory.

## Parameters

**handle**

*Input.* The current cuDNN context handle.

**attnDesc**

*Input.* A previously configured attention descriptor.

**wKind**

*Input.* Enumerant type to specify which weight or bias tensor should be retrieved.

**weightSizeInBytes**

*Input.* Buffer size that stores all multi-head attention weights and biases.

**weights**

*Input.* Pointer to the `weight` buffer in the host or device memory.

**wDesc**

*Output.* The descriptor specifying weight or bias tensor shape. For weights, the `wDesc.dimA[]` array has three elements: [`nHeads`, projected size, original size]. For biases, the `wDesc.dimA[]` array also has three elements: [`nHeads`, projected size, 1]. The `wDesc.strideA[]` array describes how tensor elements are arranged in memory.

**wAddr**

*Output.* Pointer to a location where the start address of the requested tensor should be written. When the corresponding projection is disabled, the address written to `wAddr` is `NULL`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The weight tensor descriptor and the address of data in the device memory were successfully retrieved.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered. For example, `wKind` did not have a valid value or `weightSizeInBytes` was too small.

## 7.2.17. `cudaGetRNNBackwardWeightsAlgorithmMaxCount()`

This function has been deprecated in cuDNN 8.0.

## 7.2.18. cudnnGetRNNBiasMode()

This function has been deprecated in cuDNN 8.0. Use [cudnnGetRNNDescriptor\\_v8\(\)](#) instead of `cudnnGetRNNBiasMode()`.

```

cudnnStatus_t cudnnGetRNNBiasMode(
    cudnnRNNDescriptor_t  rnnDesc,
    cudnnRNNBiasMode_t   *biasMode)
    
```

This function retrieves the RNN bias mode that was configured by [cudnnSetRNNBiasMode\(\)](#). The default value of `biasMode` in `rnnDesc` after [cudnnCreateRNNDescriptor\(\)](#) is `CUDNN_RNN_DOUBLE_BIAS`.

### Parameters

**rnnDesc**

*Input.* A previously created RNN descriptor.

**\*biasMode**

*Output.* Pointer to where RNN bias mode should be saved.

### Returns

**CUDNN\_STATUS\_BAD\_PARAM**

Either the `rnnDesc` or `*biasMode` is NULL.

**CUDNN\_STATUS\_SUCCESS**

The `biasMode` parameter was retrieved successfully.

## 7.2.19. cudnnGetRNNDataDescriptor()

This function retrieves a previously created RNN data descriptor object.

```

cudnnStatus_t cudnnGetRNNDataDescriptor(
    cudnnRNNDataDescriptor_t  RNNDataDesc,
    cudnnDataType_t           *dataType,
    cudnnRNNDataLayout_t     *layout,
    int                        *maxSeqLength,
    int                        *batchSize,
    int                        *vectorSize,
    int                        arrayLengthRequested,
    int                        seqLengthArray[],
    void                       *paddingFill);
    
```

### Parameters

**RNNDataDesc**

*Input.* A previously created and initialized RNN descriptor.

**dataType**

*Output.* Pointer to the host memory location to store the datatype of the RNN data tensor.

**layout**

*Output.* Pointer to the host memory location to store the memory layout of the RNN data tensor.

**maxSeqLength**

*Output.* The maximum sequence length within this RNN data tensor, including the padding vectors.

**batchSize**

*Output.* The number of sequences within the mini-batch.

**vectorSize**

*Output.* The vector length (meaning, embedding size) of the input or output tensor at each time-step.

**arrayLengthRequested**

*Input.* The number of elements that the user requested for seqLengthArray.

**seqLengthArray**

*Output.* Pointer to the host memory location to store the integer array describing the length (meaning, number of time-steps) of each sequence. This is allowed to be a NULL pointer if arrayLengthRequested is 0.

**paddingFill**

*Output.* Pointer to the host memory location to store the user defined symbol. The symbol should be interpreted as the same data type as the RNN data tensor.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The parameters are fetched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

Any one of these have occurred:

- ▶ Any of rnnDataDesc, dataType, layout, maxSeqLength, batchSize, vectorSize, OR paddingFill is NULL.
- ▶ seqLengthArray is NULL while arrayLengthRequested is greater than zero.
- ▶ arrayLengthRequested is less than zero.

**7.2.20. cudnnGetRNNDescriptor\_v6()**

This function has been deprecated in cuDNN 8.0. Use [cudnnGetRNNDescriptor\\_v8\(\)](#) instead of cudnnGetRNNDescriptor\_v6().

```

cudnnStatus_t cudnnGetRNNDescriptor_v6(
    cudnnHandle_t handle,
    cudnnRNNDescriptor_t rnnDesc,
    int *hiddenSize,
    int *numLayers,
    cudnnDropoutDescriptor_t *dropoutDesc,
    cudnnRNNInputMode_t *inputMode,
    cudnnDirectionMode_t *direction,
    cudnnRNNMode_t *cellMode,
    cudnnRNNAlgo_t *algo,
    cudnnDataType_t *mathPrec) {

```

This function retrieves RNN network parameters that were configured by [cudnnSetRNNDescriptor\\_v6\(\)](#). 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 parameters.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor.

### **rnnDesc**

*Input.* A previously created and initialized RNN descriptor.

### **hiddenSize**

*Output.* Pointer to where the size of the hidden state should be stored (the same value is used in every RNN layer).

### **numLayers**

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

### **dropoutDesc**

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

### **inputMode**

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

### **direction**

*Output.* Pointer to where RNN unidirectional/bidirectional mode should be saved.

### **mode**

*Output.* Pointer to where RNN cell type should be saved.

### **algo**

*Output.* Pointer to where RNN algorithm type should be stored.

### **mathPrec**

*Output.* Pointer to where the math precision type 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 function is NULL.

## 7.2.21. [cudnnGetRNNDescriptor\\_v8\(\)](#)

This function retrieves RNN network parameters that were configured by [cudnnSetRNNDescriptor\\_v8\(\)](#). The user can assign NULL to any pointer except `rnnDesc` when the retrieved value is not needed. The function does not check the validity of retrieved parameters.

```
cudnnStatus_t cudnnGetRNNDescriptor_v8(
    cudnnRNNDescriptor_t rnnDesc,
    cudnnRNNAlgo_t *algo,
    cudnnRNNMode_t *cellMode,
    cudnnRNNBiasMode_t *biasMode,
```

```

    cudnnDirectionMode_t *dirMode,
    cudnnRNNInputMode_t *inputMode,
    cudnnDataType_t *dataType,
    cudnnDataType_t *mathPrec,
    cudnnMathType_t *mathType,
    int32_t *inputSize,
    int32_t *hiddenSize,
    int32_t *projSize,
    int32_t *numLayers,
    cudnnDropoutDescriptor_t *dropoutDesc,
    uint32_t *auxFlags);

```

## Parameters

### **rnnDesc**

*Input.* A previously created and initialized RNN descriptor.

### **algo**

*Output.* Pointer to where RNN algorithm type should be stored.

### **cellMode**

*Output.* Pointer to where RNN cell type should be saved.

### **biasMode**

*Output.* Pointer to where RNN bias mode [cudnnRNNBiasMode\\_t](#) should be saved.

### **dirMode**

*Output.* Pointer to where RNN unidirectional/bidirectional mode should be saved.

### **inputMode**

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

### **dataType**

*Output.* Pointer to where the data type of RNN weights/biases should be stored.

### **mathPrec**

*Output.* Pointer to where the math precision type should be stored.

### **mathType**

*Output.* Pointer to where the preferred option for Tensor Cores are saved.

### **inputSize**

*Output.* Pointer to where the RNN input vector size is stored.

### **hiddenSize**

*Output.* Pointer to where the size of the hidden state should be stored (the same value is used in every RNN layer).

### **projSize**

*Output.* Pointer to where the LSTM cell output size after the recurrent projection is stored.

### **numLayers**

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

### **dropoutDesc**

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

### **auxFlags**

*Output.* Pointer to miscellaneous RNN options (flags) that do not require passing additional numerical values to configure.



## Returns

### CUDNN\_STATUS\_SUCCESS

RNN parameters were successfully retrieved from the RNN descriptor.

### CUDNN\_STATUS\_BAD\_PARAM

An invalid input argument was found (`rnnDesc` was `NULL`).

### CUDNN\_STATUS\_NOT\_INITIALIZED

The RNN descriptor was configured with the legacy [`cudaSetRNNDescriptor\_v6\(\)`](#) call.

## 7.2.22. `cudaGetRNNForwardInferenceAlgorithmMaxCount`

This function has been deprecated in cuDNN 8.0.

## 7.2.23. `cudaGetRNNLinLayerBiasParams()`

This function has been deprecated in cuDNN 8.0. Use [`cudaGetRNNWeightParams\(\)`](#) instead of `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`.



Note: 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 returns the total number of vector elements in `linLayerBiasDesc` as follows:

```

filterDimA[0]=total_size,
filterDimA[1]=1,
filterDimA[2]=1
    
```

For more information, see the description of the [`cudaGetFilterNdDescriptor\(\)`](#) function.

In cuDNN 7.1.1, 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()`.

The RNN implementation in cuDNN uses two bias vectors before the cell non-linear function. Note that the RNN implementation in cuDNN depends on the number of bias vectors before the cell non-linear function. Refer to the equations in the `cudaRNNMode_t` description, for the enumerant type based on the value of `cudaRNNBiasMode_t` `biasMode` in `rnnDesc`. If nonexistent biases are referenced by `linLayerID`, then this function sets `linLayerBiasDesc` to a zeroed filter descriptor where:

```
filterDimA[0]=0,
filterDimA[1]=0, and
filterDimA[2]=2
```

and sets `linLayerBias` to `NULL`. Refer to the details for the function parameter `linLayerID` to determine the relevant values of `linLayerID` based on `biasMode`.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **pseudoLayer**

*Input.* The pseudo-layer to query. In unidirectional RNNs, a pseudo-layer is the same as a physical layer (`pseudoLayer=0` is the RNN input layer, `pseudoLayer=1` is the first hidden layer). In bidirectional RNNs, there are twice as many pseudo-layers in comparison to physical layers.

- ▶ `pseudoLayer=0` refers to the forward part of the physical input layer
- ▶ `pseudoLayer=1` refers to the backward part of the physical input layer
- ▶ `pseudoLayer=2` is the forward part of the first hidden layer, and so on

### **xDesc**

*Input.* A fully packed tensor descriptor describing the input to one recurrent iteration (to retrieve the RNN input width).

### **wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN.

### **w**

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

### **linLayerID**

*Input.* Linear ID index of the weight matrix.

If `cellMode` in `rnnDesc` was set to `CUDNN_RNN_RELU` or `CUDNN_RNN_TANH`:

- ▶ Value 0 references the weight matrix used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Value 1 references the weight matrix used in conjunction with the hidden state from the previous time step or the initial hidden state.

If `cellMode` in `rnnDesc` was set to `CUDNN_LSTM`:

- ▶ Values 0, 1, 2, and 3 reference weight matrices used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Values 4, 5, 6, and 7 reference weight matrices used in conjunction with the hidden state from the previous time step or the initial hidden state.
- ▶ Value 8 corresponds to the projection matrix, if enabled.

Values and their LSTM gates:

- ▶ `linLayerID0` and 4 correspond to the input gate.
- ▶ `linLayerID1` and 5 correspond to the forget gate.
- ▶ `linLayerID2` and 6 correspond to the new cell state calculations with a hyperbolic tangent.
- ▶ `linLayerID3` and 7 correspond to the output gate.

If `cellMode` in `rnnDesc` was set to `CUDNN_GRU`:

- ▶ Values 0, 1, and 2 reference weight matrices used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Values 3, 4, and 5 reference weight matrices used in conjunction with the hidden state from the previous time step or the initial hidden state.

Values and their GRU gates:

- ▶ `linLayerID0` and 3 correspond to the reset gate.
- ▶ `linLayerID1` and 4 references to the update gate.
- ▶ `linLayerID2` and 5 correspond to the new hidden state calculations with a hyperbolic tangent.

#### **linLayerBiasDesc**

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

#### **linLayerBias**

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

`linLayerBiasDesc`.

## Returns

#### **CUDNN\_STATUS\_SUCCESS**

The query was successful.

#### **CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

#### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ One of the following arguments is `NULL`: `handle`, `rnnDesc`, `xDesc`, `wDesc`, `linLayerBiasDesc`, or `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.


## 7.2.24. `cudaGetRNNLinLayerMatrixParams()`

This function has been deprecated in cuDNN 8.0. Use [cudaGetRNNWeightParams\(\)](#) instead of `cudaGetRNNLinLayerMatrixParams()`.

```

cudaStatus_t cudaGetRNNLinLayerMatrixParams(
    cudaHandle_t          handle,
    const cudaRNNDescrptor_t  rnnDesc,
    const int             pseudoLayer,
    const cudaTensorDescrptor_t  xDesc,
    const cudaFilterDescrptor_t  wDesc,
    const void           *w,
    const int            linLayerID,
    cudaFilterDescrptor_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`.

 **Note:** The `cudaGetRNNLinLayerMatrixParams()` function was enhanced in cuDNN version 7.1.1 without changing its prototype. Instead of reporting the total number of elements in each weight matrix in the `linLayerMatDesc` filter descriptor, the function returns the matrix size as two dimensions: rows and columns. Moreover, when a weight matrix does not exist, for example, due to `CUDNN_SKIP_INPUT` mode, the function returns `NULL` in `linLayerMat` and all fields of `linLayerMatDesc` are zero.

The `cudaGetRNNLinLayerMatrixParams()` 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 [cudaGetFilterNdDescriptor\(\)](#) function). In cuDNN 7.1.1, 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 [cudaGetFilterNdDescriptor\(\)](#).

### 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 unidirectional RNNs, a pseudo-layer is the same as a physical layer (`pseudoLayer=0` is the RNN input layer, `pseudoLayer=1` is the first hidden layer). In bidirectional RNNs, there are twice as many pseudo-layers in comparison to physical layers.

- ▶ `pseudoLayer=0` refers to the forward part of the physical input layer
- ▶ `pseudoLayer=1` refers to the backward part of the physical input layer
- ▶ `pseudoLayer=2` is the forward part of the first hidden layer, and so on

**xDesc**

*Input.* A fully packed tensor descriptor describing the input to one recurrent iteration (to retrieve the RNN input width).

**wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

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

**linLayerID**

*Input.* The linear layer to obtain information about:

- ▶ If `mode` in `rnnDesc` was set to `CUDNN_RNN_RELU` or `CUDNN_RNN_TANH`:
  - ▶ Value 0 references the bias applied to the input from the previous layer (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS` or `CUDNN_RNN_DOUBLE_BIAS`).
  - ▶ Value 1 references the bias applied to the recurrent input (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_DOUBLE_BIAS` or `CUDNN_RNN_SINGLE_REC_BIAS`).
- ▶ If `mode` in `rnnDesc` was set to `CUDNN_LSTM`:
  - ▶ Values of 0, 1, 2 and 3 reference bias applied to the input from the previous layer (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS` or `CUDNN_RNN_DOUBLE_BIAS`).
  - ▶ Values of 4, 5, 6 and 7 reference bias applied to the recurrent input (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_DOUBLE_BIAS` or `CUDNN_RNN_SINGLE_REC_BIAS`).
  - ▶ Values and their associated gates:
    - ▶ Values 0 and 4 reference the input gate.
    - ▶ Values 1 and 5 reference the forget gate.
    - ▶ Values 2 and 6 reference the new memory gate.
    - ▶ Values 3 and 7 reference the output gate.

- ▶ If `mode` in `rnnDesc` was set to `CUDNN_GRU`:
  - ▶ Values of 0, 1 and 2 reference bias applied to the input from the previous layer (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_SINGLE_INP_BIAS` or `CUDNN_RNN_DOUBLE_BIAS`).
  - ▶ Values of 3, 4 and 5 reference bias applied to the recurrent input (relevant if `biasMode` in `rnnDesc` is `CUDNN_RNN_DOUBLE_BIAS` or `CUDNN_RNN_SINGLE_REC_BIAS`).
  - ▶ Values and their associated gates:
    - ▶ Values 0 and 3 reference the reset gate.
    - ▶ Values 1 and 4 reference the update gate.
    - ▶ Values 2 and 5 reference the new memory gate.

For more information on modes and bias modes, refer to [cudaRNNMode\\_t](#).

**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`.

**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`, or `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.

## 7.2.25. `cudaGetRNNMatrixMathType()`

This function has been deprecated in cuDNN 8.0. Use [`cudaGetRNNDescriptor\_v8\(\)`](#) instead of `cudaGetRNNMatrixMathType()`.

```
cudaStatus_t cudaGetRNNMatrixMathType(
    cudaRNNDescriptor_t rnnDesc,
    cudaMathType_t *mType);
```

This function retrieves the preferred settings for NVIDIA Tensor Cores on NVIDIA Volta™ (SM 7.0) or higher GPUs. Refer to the [`cudaMathType\_t`](#) description for more details.

### Parameters

#### **rnnDesc**

*Input.* A previously created and initialized RNN descriptor.

#### **mType**

*Output.* Address where the preferred Tensor Core settings should be stored.

### Returns

#### **CUDNN\_STATUS\_SUCCESS**

The requested RNN descriptor field was retrieved successfully.

#### **CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was found (`rnnDesc` or `mType` was NULL).

## 7.2.26. `cudaGetRNNPaddingMode()`

This function has been deprecated in cuDNN 8.0. Use [`cudaGetRNNDescriptor\_v8\(\)`](#) instead of `cudaGetRNNPaddingMode()`.

```
cudaStatus_t cudaGetRNNPaddingMode(
    cudaRNNDescriptor_t      rnnDesc,
    cudaRNNPaddingMode_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.

## 7.2.27. cudnnGetRNNParamsSize()

This function has been deprecated in cuDNN 8.0. Use [cudnnGetRNNWeightSpaceSize\(\)](#) instead of `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 input dimensions defined by `xDesc`.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**xDesc**

*Input.* A fully packed tensor descriptor describing the input to one recurrent iteration.

**sizeInBytes**

*Output.* Minimum amount of GPU memory needed as parameter space to be able to execute an RNN with the specified descriptor and input tensors.

**dataType**

*Input.* The data type of the parameters.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ The descriptor `xDesc` is invalid.
- ▶ The descriptor `xDesc` is not fully packed.
- ▶ The combination of `dataType` and tensor descriptor data type is invalid.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The combination of the RNN descriptor and tensor descriptors is not supported.

## 7.2.28. cudnnGetRNNProjectionLayers()

This function has been deprecated in cuDNN 8.0. Use [cudnnGetRNNDescriptor\\_v8\(\)](#) instead of `cudnnGetRNNProjectionLayers()`.

```

cudnnStatus_t cudnnGetRNNProjectionLayers(
    cudnnHandle_t          handle,
    
```



```

    cudnnRNNDescriptor_t  rnnDesc,
    int                   *recProjSize,
    int                   *outProjSize)

```

This function retrieves the current RNN projection parameters. By default, the projection feature is disabled so invoking this function 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.

## 7.2.29. cudnnGetRNNTempSpaceSizes()

This function computes the work and reserve space buffer sizes based on the RNN network geometry stored in `rnnDesc`, designated usage (inference or training) defined by the `fMode` argument, and the current RNN data dimensions (`maxSeqLength`, `batchSize`) retrieved from `xDesc`. When RNN data dimensions change, the `cudnnGetRNNTempSpaceSizes()` must be called again because RNN temporary buffer sizes are not monotonic.

```

cudnnStatus_t cudnnGetRNNTempSpaceSizes(
    cudnnHandle_t handle,
    cudnnRNNDescriptor_t rnnDesc,
    cudnnForwardMode_t fMode,
    cudnnRNNDataDescriptor_t xDesc,
    size_t *workSpaceSize,
    size_t *reserveSpaceSize);

```

The user can assign NULL to `workSpaceSize` or `reserveSpaceSize` pointers when the corresponding value is not needed.

## Parameters

### handle

*Input.* The current cuDNN context handle.

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**fMode**

*Input.* Specifies whether temporary buffers are used in inference or training modes. The reserve-space buffer is not used during inference. Therefore, the returned size of the reserve space buffer will be zero when the `fMode` argument is `CUDNN_FWD_MODE_INFERENCE`.

**xDesc**

*Input.* A single RNN data descriptor that specifies current RNN data dimensions: `maxSeqLength` and `batchSize`.

**workspaceSize**

*Output.* Minimum amount of GPU memory in bytes needed as a workspace buffer. The workspace buffer is not used to pass intermediate results between APIs but as a temporary read/write buffer.

**reserveSpaceSize**

*Output.* Minimum amount of GPU memory in bytes needed as the reserve-space buffer. The reserve space buffer is used to pass intermediate results from `cudaRNNForward()` to `RNN BackwardData` and `BackwardWeights` routines that compute first order derivatives with respect to RNN inputs or trainable weight and biases.

Returns

**CUDNN\_STATUS\_SUCCESS**

RNN temporary buffer sizes were computed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was detected.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

An incompatible or unsupported combination of input arguments was detected.

## 7.2.30. `cudaGetRNNTrainingReserveSize()`

This function has been deprecated in cuDNN 8.0. Use `cudaGetRNNTempSpaceSizes()` instead of `cudaGetRNNTrainingReserveSize()`.

```

cudaStatus_t cudaGetRNNTrainingReserveSize(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_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 input dimensions defined by `xDesc`. The same reserved space buffer must be passed to `cudaRNNForwardTraining()`, `cudaRNNBackwardData()`, and `cudaRNNBackwardWeights()`. 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 the `cudaGetRNNWorkspaceSize()` function for querying the workspace size required to execute the RNN.

### xDesc

*Input.* An array of tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element  $n$  to element  $n+1$  but may not increase. Each tensor descriptor must have the same second dimension (vector length).

### sizeInBytes

*Output.* Minimum amount of GPU memory needed as reserve space to be able to train an RNN with the specified descriptor and input tensors.

## Returns

### CUDNN\_STATUS\_SUCCESS

The query was successful.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors in `xDesc` is invalid.
- ▶ The descriptors in `xDesc` have inconsistent second dimensions, strides or data types.
- ▶ The descriptors in `xDesc` have increasing first dimensions.
- ▶ The descriptors in `xDesc` are not fully packed.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The data types in tensors described by `xDesc` are not supported.

## 7.2.31. `cudaGetRNNWeightParams()`

This function is used to obtain the start address and shape of every RNN weight matrix and bias vector in each pseudo-layer within the recurrent network.

```

cudaStatus_t cudaGetRNNWeightParams(
    cudaHandle_t handle,
    cudaRNNDescriptor_t rnnDesc,
    int32_t pseudoLayer,
    size_t weightSpaceSize,
    const void *weightSpace,
    int32_t linLayerID,

```

```

    cudnnTensorDescriptor_t mDesc,
void **mAddr,
    cudnnTensorDescriptor_t bDesc,
void **bAddr);

```

## 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 unidirectional RNNs, a pseudo-layer is the same as a physical layer (`pseudoLayer=0` is the RNN input layer, `pseudoLayer=1` is the first hidden layer). In bidirectional RNNs, there are twice as many pseudo-layers in comparison to physical layers:

- ▶ `pseudoLayer=0` refers to the forward direction sub-layer of the physical input layer
- ▶ `pseudoLayer=1` refers to the backward direction sub-layer of the physical input layer
- ▶ `pseudoLayer=2` is the forward direction sub-layer of the first hidden layer, and so on

### weightSpaceSize

*Input.* Size of the weight space buffer in bytes.

### weightSpace

*Input.* Pointer to the weight space buffer.

### linLayerID

*Input.* Weight matrix or bias vector linear ID index.

If `cellMode` in `rnnDesc` was set to `CUDNN_RNN_RELU` or `CUDNN_RNN_TANH`:

- ▶ Value 0 references the weight matrix or bias vector used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Value 1 references the weight matrix or bias vector used in conjunction with the hidden state from the previous time step or the initial hidden state.

If `cellMode` in `rnnDesc` was set to `CUDNN_LSTM`:

- ▶ Values 0, 1, 2 and 3 reference weight matrices or bias vectors used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Values 4, 5, 6 and 7 reference weight matrices or bias vectors used in conjunction with the hidden state from the previous time step or the initial hidden state.
- ▶ Value 8 corresponds to the projection matrix, if enabled (there is no bias in this operation).

Values and their LSTM gates:

- ▶ `linLayerID0` and 4 correspond to the input gate.
- ▶ `linLayerID1` and 5 correspond to the forget gate.
- ▶ `linLayerID2` and 6 correspond to the new cell state calculations with hyperbolic tangent.
- ▶ `linLayerID3` and 7 correspond to the output gate.

If `cellMode` in `rnnDesc` was set to `CUDNN_GRU`:

- ▶ Values 0, 1 and 2 reference weight matrices or bias vectors used in conjunction with the input from the previous layer or input to the RNN model.
- ▶ Values 3, 4 and 5 reference weight matrices or bias vectors used in conjunction with the hidden state from the previous time step or the initial hidden state.

Values and their GRU gates:

- ▶ `linLayerID0` and 3 correspond to the reset gate.
- ▶ `linLayerID1` and 4 reference to the update gate.
- ▶ `linLayerID2` and 5 correspond to the new hidden state calculations with hyperbolic tangent.

For more information on modes and bias modes, refer to [cudaRNNMode\\_t](#).

**mDesc**

*Output.* Handle to a previously created tensor descriptor. The shape of the corresponding weight matrix is returned in this descriptor in the following format: `dimA[3] = {1, rows, cols}`. The reported number of tensor dimensions is zero when the weight matrix does not exist. This situation occurs for input GEMM matrices of the first layer when `CUDNN_SKIP_INPUT` is selected or for the LSTM projection matrix when the feature is disabled.

**mAddr**

*Output.* Pointer to the beginning of the weight matrix within the weight space buffer. When the weight matrix does not exist, the returned address is `NULL`.

**bDesc**

*Output.* Handle to a previously created tensor descriptor. The shape of the corresponding bias vector is returned in this descriptor in the following format: `dimA[3] = {1, rows, 1}`. The reported number of tensor dimensions is zero when the bias vector does not exist.

**bAddr**

*Output.* Pointer to the beginning of the bias vector within the weight space buffer. When the bias vector does not exist, the returned address is `NULL`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was completed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was encountered. For example, the value of `pseudoLayer` is out of range or `linLayerID` is negative or larger than 8.

**CUDNN\_STATUS\_INVALID\_VALUE**

Some weight/bias elements are outside the weight space buffer boundaries.

**CUDNN\_STATUS\_NOT\_INITIALIZED**

The RNN descriptor was configured with the legacy [`cudaSetRNNDescrptor\_v6\(\)`](#) call.

## 7.2.32. `cudaGetRNNWeightSpaceSize()`

This function reports the required size of the weight space buffer in bytes. The weight space buffer holds all RNN weight matrices and bias vectors.

```
cudaStatus_t cudaGetRNNWeightSpaceSize(
    cudaHandle_t handle,
    cudaRNNDescrptor_t rnnDesc,
    size_t *weightSpaceSize);
```

### Parameters

**handle**

*Input.* The current cuDNN context handle.

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**weightSpaceSize**

*Output.* Minimum size in bytes of GPU memory needed for all RNN trainable parameters.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was encountered. For example, any input argument was NULL.

**CUDNN\_STATUS\_NOT\_INITIALIZED**

The RNN descriptor was configured with the legacy [`cudaSetRNNDescrptor\_v6\(\)`](#) call.

## 7.2.33. `cudaGetRNNWorkspaceSize()`

This function has been deprecated in cuDNN 8.0. Use [`cudaGetRNNTempSpaceSizes\(\)`](#) instead of `cudaGetRNNWorkspaceSize()`.

```
cudaStatus_t cudaGetRNNWorkspaceSize(
    cudaHandle_t handle,
    const cudaRNNDescrptor_t rnnDesc,
    const int seqLength,
    const cudaTensorDescrptor_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 input dimensions defined by `xDesc`.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN library descriptor.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **seqLength**

*Input.* Number of iterations to unroll over. Workspace that is allocated, based on the size that this function provides, cannot be used for sequences longer than `seqLength`.

### **xDesc**

*Input.* An array of tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element  $n$  to element  $n+1$  but may not increase. For example, if you have multiple time series in a batch, they can be different lengths. This dimension is the batch size for the particular iteration of the sequence, and so it should decrease when a sequence in the batch has been terminated.

Each tensor descriptor must have the same second dimension (vector length).

### **sizeInBytes**

*Output.* Minimum amount of GPU memory needed as workspace to be able to execute an RNN with the specified descriptor and input tensors.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The query was successful.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors in `xDesc` is invalid.
- ▶ The descriptors in `xDesc` have inconsistent second dimensions, strides or data types.
- ▶ The descriptors in `xDesc` have increasing first dimensions.
- ▶ The descriptors in `xDesc` are not fully packed.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

The data types in tensors described by `xDesc` are not supported.

## 7.2.34. `cudaGetSeqDataDescriptor()`

This function retrieves settings from a previously created sequence data descriptor. The user can assign `NULL` to any pointer except `seqDataDesc` when the retrieved value is

not needed. The `nbDimsRequested` argument applies to both `dimA[]` and `axes[]` arrays. A positive value of `nbDimsRequested` or `seqLengthSizeRequested` is ignored when the corresponding array, `dimA[]`, `axes[]`, or `seqLengthArray[]` is `NULL`.

```

cudaStatus_t cudaGetSeqDataDescriptor(
    const cudaSeqDataDescriptor_t seqDataDesc,
    cudaDataType_t *dataType,
    int *nbDims,
    int nbDimsRequested,
    int dimA[],
    cudaSeqDataAxis_t axes[],
    size_t *seqLengthArraySize,
    size_t seqLengthSizeRequested,
    int seqLengthArray[],
    void *paddingFill);

```

The `cudaGetSeqDataDescriptor()` function does not report the actual strides in the sequence data buffer. Those strides can be handy in computing the offset to any sequence data element. The user must precompute strides based on the `axes[]` and `dimA[]` arrays reported by the `cudaGetSeqDataDescriptor()` function. Below is sample code that performs this task:

```

// Array holding sequence data strides.
size_t strA[CUDNN_SEQDATA_DIM_COUNT] = {0};

// Compute strides from dimension and order arrays.
size_t stride = 1;
for (int i = nbDims - 1; i >= 0; i--) {
    int j = int(axes[i]);
    if (unsigned(j) < CUDNN_SEQDATA_DIM_COUNT-1 && strA[j] == 0) {
        strA[j] = stride;
        stride *= dimA[j];
    } else {
        fprintf(stderr, "ERROR: invalid axes[%d]=%d\n\n", i, j);
        abort();
    }
}

```

Now, the `strA[]` array can be used to compute the index to any sequence data element, for example:

```

// Using four indices (batch, beam, time, vect) with ranges already checked.
size_t base = strA[CUDNN_SEQDATA_BATCH_DIM] * batch
              + strA[CUDNN_SEQDATA_BEAM_DIM] * beam
              + strA[CUDNN_SEQDATA_TIME_DIM] * time;
val = seqDataPtr[base + vect];

```

The above code assumes that all four indices (`batch`, `beam`, `time`, `vect`) are less than the corresponding value in the `dimA[]` array. The sample code also omits the `strA[CUDNN_SEQDATA_VECT_DIM]` stride because its value is always 1, meaning, elements of one vector occupy a contiguous block of memory.

## Parameters

### **seqDataDesc**

*Input.* Sequence data descriptor.

### **dataType**

*Output.* Data type used in the sequence data buffer.

### **nbDims**

*Output.* The number of active dimensions in the `dimA[]` and `axes[]` arrays.



**nbDimsRequested**

*Input.* The maximum number of consecutive elements that can be written to `dimA[]` and `axes[]` arrays starting from index zero. The recommended value for this argument is `CUDNN_SEQDATA_DIM_COUNT`.

**dimA[]**

*Output.* Integer array holding sequence data dimensions.

**axes[]**

*Output.* Array of `cudaSeqDataAxis_t` that defines the layout of sequence data in memory.

**seqLengthArraySize**

*Output.* The number of required elements in `seqLengthArray[]` to save all sequence lengths.

**seqLengthSizeRequested**

*Input.* The maximum number of consecutive elements that can be written to the `seqLengthArray[]` array starting from index zero.

**seqLengthArray[]**

*Output.* Integer array holding sequence lengths.

**paddingFill**

*Output.* Pointer to a storage location of `dataType` with the fill value that should be written to all padding vectors. Use `NULL` when an explicit initialization of output padding vectors was not requested.

Returns

**CUDNN\_STATUS\_SUCCESS**

Requested sequence data descriptor fields were retrieved successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was found.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

An inconsistent internal state was encountered.

## 7.2.35. `cudaMultiHeadAttnForward()`

The `cudaMultiHeadAttnForward()` function computes the forward responses of the multi-head attention layer. When `reserveSpaceSizeInBytes=0` and `reserveSpace=NULL`, the function operates in the inference mode in which backward (gradient) functions are not invoked, otherwise, the training mode is assumed. In the training mode, the reserve space is used to pass intermediate results from `cudaMultiHeadAttnForward()` to `cudaMultiHeadAttnBackwardData()` and from `cudaMultiHeadAttnBackwardData()` to `cudaMultiHeadAttnBackwardWeights()`.

```

cudaStatus_t cudaMultiHeadAttnForward(
    cudaHandle_t handle,
    const cudaAttnDescriptor_t attnDesc,
    int currIdx,
    const int loWinIdx[],
    const int hiWinIdx[],
    const int devSeqLengthsQO[],
    const int devSeqLengthsKV[],

```

```
const cudnnSeqDataDescriptor_t qDesc,
const void *queries,
const void *residuals,
const cudnnSeqDataDescriptor_t kDesc,
const void *keys,
const cudnnSeqDataDescriptor_t vDesc,
const void *values,
const cudnnSeqDataDescriptor_t oDesc,
    void *out,
size_t weightSizeInBytes,
const void *weights,
size_t workSpaceSizeInBytes,
void *workSpace,
size_t reserveSpaceSizeInBytes,
void *reserveSpace);
```

In the inference mode, the `currIdx` specifies the time-step or sequence index of the embedding vectors to be processed. In this mode, the user can perform one iteration for time-step zero (`currIdx=0`), then update Q, K, V vectors and the attention window, and execute the next step (`currIdx=1`). The iterative process can be repeated for all time-steps.

When all Q time-steps are available (for example, in the training mode or in the inference mode on the encoder side in self-attention), the user can assign a negative value to `currIdx` and the `cudnnMultiHeadAttnForward()` API will automatically sweep through all Q time-steps.

The `loWinIdx[]` and `hiWinIdx[]` host arrays specify the attention window size for each Q time-step. In a typical self-attention case, the user must include all previously visited embedding vectors but not the current or future vectors. In this situation, the user should set:

```
currIdx=0: loWinIdx[0]=0; hiWinIdx[0]=0; // initial time-step, no attention window
currIdx=1: loWinIdx[1]=0; hiWinIdx[1]=1; // attention window spans one vector
currIdx=2: loWinIdx[2]=0; hiWinIdx[2]=2; // attention window spans two vectors
(...)
```

When `currIdx` is negative in `cudnnMultiHeadAttnForward()`, the `loWinIdx[]` and `hiWinIdx[]` arrays must be fully initialized for all time-steps. When `cudnnMultiHeadAttnForward()` is invoked with `currIdx=0`, `currIdx=1`, `currIdx=2`, etc., then the user can update `loWinIdx[currIdx]` and `hiWinIdx[currIdx]` elements only before invoking the forward response function. All other elements in the `loWinIdx[]` and `hiWinIdx[]` arrays will not be accessed. Any adaptive attention window scheme can be implemented that way.

Use the following settings when the attention window should be the maximum size, for example, in cross-attention:

```
currIdx=0: loWinIdx[0]=0; hiWinIdx[0]=maxSeqLenK;
currIdx=1: loWinIdx[1]=0; hiWinIdx[1]=maxSeqLenK;
currIdx=2: loWinIdx[2]=0; hiWinIdx[2]=maxSeqLenK;
(...)
```

The `maxSeqLenK` value above should be equal to or larger than `dimA[CUDNN_SEQDATA_TIME_DIM]` in the `kDesc` descriptor. A good choice is to use `maxSeqLenK=INT_MAX` from `limits.h`.



**Note:** The actual length of any K sequence defined in `seqLengthArray[]` in `cudnnSetSeqDataDescriptor()` can be shorter than `maxSeqLenK`. The effective attention

window span is computed based on `seqLengthArray[]` stored in the K sequence descriptor and indices held in `loWinIdx[]` and `hiWinIdx[]` arrays.

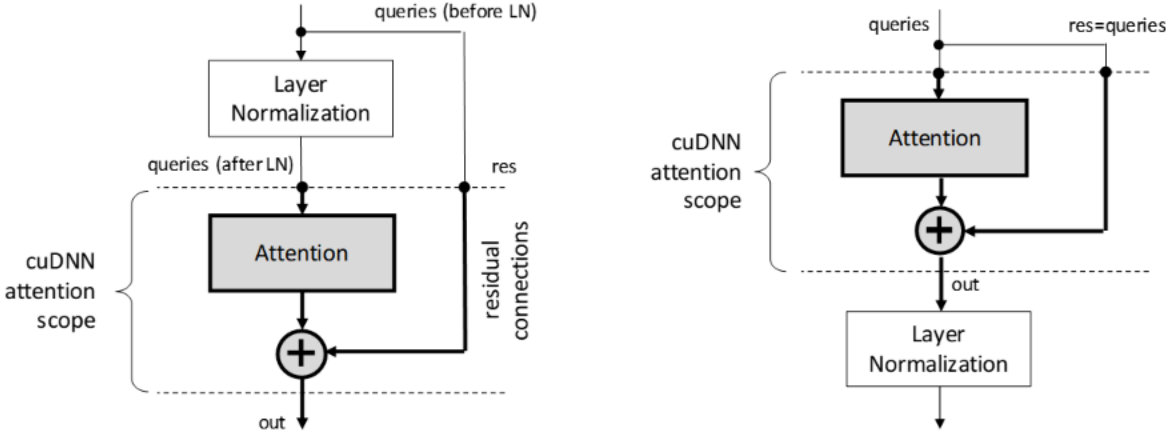
`devSeqLengthsQO[]` and `devSeqLengthsKV[]` are pointers to device (not host) arrays with Q, O, and K, V sequence lengths. Note that the same information is also passed in the corresponding descriptors of type `cudaSeqDataDescriptor_t` on the host side. The need for extra device arrays comes from the asynchronous nature of cuDNN calls and limited size of the constant memory dedicated to GPU kernel arguments. When the `cudaMultiHeadAttnForward()` API returns, the sequence length arrays stored in the descriptors can be immediately modified for the next iteration. However, the GPU kernels launched by the forward call may not have started at this point. For this reason, copies of sequence arrays are needed on the device side to be accessed directly by GPU kernels. Those copies cannot be created inside the `cudaMultiHeadAttnForward()` function for very large K, V inputs without the device memory allocation and CUDA stream synchronization.

To reduce the `cudaMultiHeadAttnForward()` API overhead, `devSeqLengthsQO[]` and `devSeqLengthsKV[]` device arrays are not validated to contain the same settings as `seqLengthArray[]` in the sequence data descriptors.

Sequence lengths in the `kDesc` and `vDesc` descriptors should be the same. Similarly, sequence lengths in the `qDesc` and `oDesc` descriptors should match. The user can define six different data layouts in the `qDesc`, `kDesc`, `vDesc` and `oDesc` descriptors. Refer to the `cudaSetSeqDataDescriptor()` function for the discussion of those layouts. All multi-head attention API calls require that the same layout is used in all sequence data descriptors.

In the transformer model, the multi-head attention block is tightly coupled with the layer normalization and residual connections. `cudaMultiHeadAttnForward()` does not encompass the layer normalization but it can be used to handle residual connections as depicted in the following figure.

Figure 2. Multi-Head Attention Block is Tightly Coupled with the Layer Normalization and Residual Connections



Queries and residuals share the same `qDesc` descriptor in `cudaMultiHeadAttnForward()`. When residual connections are disabled, the residuals pointer should be `NULL`. When residual connections are enabled, the vector length in `qDesc` should match the vector length specified in the `oDesc` descriptor, so that a vector addition is feasible.

The `queries`, `keys`, and `values` pointers are not allowed to be `NULL`, even when K and V are the same inputs or Q, K, V are the same inputs.

## Parameters

### **handle**

*Input.* The current cuDNN context handle.

### **attnDesc**

*Input.* A previously initialized attention descriptor.

### **currIdx**

*Input.* Time-step in queries to process. When the `currIdx` argument is negative, all Q time-steps are processed. When `currIdx` is zero or positive, the forward response is computed for the selected time-step only. The latter input can be used in inference mode only, to process one time-step while updating the next attention window and Q, R, K, V inputs in-between calls.

### **loWinIdx[], hiWinIdx[]**

*Input.* Two host integer arrays specifying the start and end indices of the attention window for each Q time-step. The start index in K, V sets is inclusive, and the end index is exclusive.

### **devSeqLengthsQO[]**

*Input.* Device array specifying sequence lengths of query, residual, and output sequence data.

### **devSeqLengthsKV[]**

*Input.* Device array specifying sequence lengths of key and value input data.

### **qDesc**

*Input.* Descriptor for the query and residual sequence data.

### **queries**

*Input.* Pointer to queries data in the device memory.

### **residuals**

*Input.* Pointer to residual data in device memory. Set this argument to `NULL` if no residual connections are required.

### **kDesc**

*Input.* Descriptor for the `keys` sequence data.

### **keys**

*Input.* Pointer to `keys` data in device memory.

### **vDesc**

*Input.* Descriptor for the `values` sequence data.

### **values**

*Input.* Pointer to `values` data in device memory.

**oDesc**

*Input.* Descriptor for the multi-head attention output sequence data.

**out**

*Output.* Pointer to device memory where the output response should be written.

**weightSizeInBytes**

*Input.* Size of the weight buffer in bytes where all multi-head attention trainable parameters are stored.

**weights**

*Input.* Pointer to the weight buffer in device memory.

**workSpaceSizeInBytes**

*Input.* Size of the work-space buffer in bytes used for temporary API storage.

**workSpace**

*Input/Output.* Pointer to the work-space buffer in device memory.

**reserveSpaceSizeInBytes**

*Input.* Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls. This parameter should be zero in the inference mode and non-zero in the training mode.

**reserveSpace**

*Input/Output.* Pointer to the reserve-space buffer in device memory. This argument should be `NULL` in inference mode and `non-NULL` in the training mode.

## Returns

**CUDNN\_STATUS\_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered. Some examples include:

- ▶ a required input pointer was `NULL`
- ▶ `currIdx` was out of bound
- ▶ the descriptor value for `attention`, `query`, `key`, `value`, and `output` were incompatible with one another

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

An inconsistent internal state was encountered.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

A requested option or a combination of input arguments is not supported.

**CUDNN\_STATUS\_ALLOC\_FAILED**

Insufficient amount of shared memory to launch a GPU kernel.

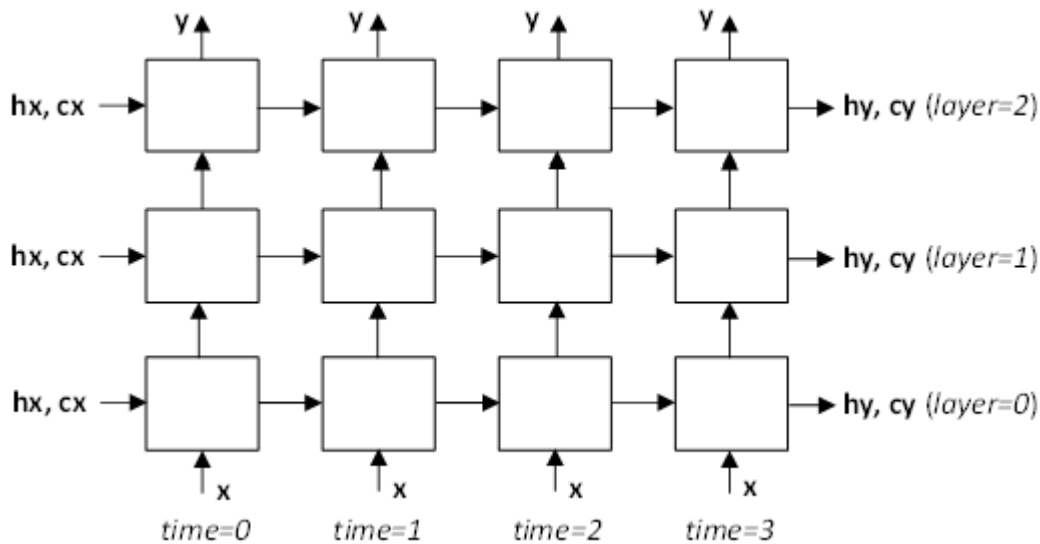
### 7.2.36. cudnnRNNForward()

This routine computes the forward response of the recurrent neural network described by `rnnDesc` with inputs in `x`, `hx`, `cx`, and weights/biases in the `weightSpace` buffer. RNN outputs are written to `y`, `hy`, and `cy` buffers. Locations of `x`, `y`, `hx`, `cx`, `hy`, and `cy` signals in the multi-layer RNN model are shown in the following figure. Note that internal RNN signals between time-steps and between layers are not exposed to the user.

```

cudnnStatus_t cudnnRNNForward(
    cudnnHandle_t handle,
    cudnnRNNDescriptor_t rnnDesc,
    cudnnForwardMode_t fwdMode,
    const int32_t devSeqLengths[],
    cudnnRNNDataDescriptor_t xDesc,
    const void *x,
    cudnnRNNDataDescriptor_t yDesc,
    void *y,
    cudnnTensorDescriptor_t hDesc,
    const void *hx,
    void *hy,
    cudnnTensorDescriptor_t cDesc,
    const void *cx,
    void *cy,
    size_t weightSpaceSize,
    const void *weightSpace,
    size_t workSpaceSize,
    void *workSpace,
    size_t reserveSpaceSize,
    void *reserveSpace);
    
```

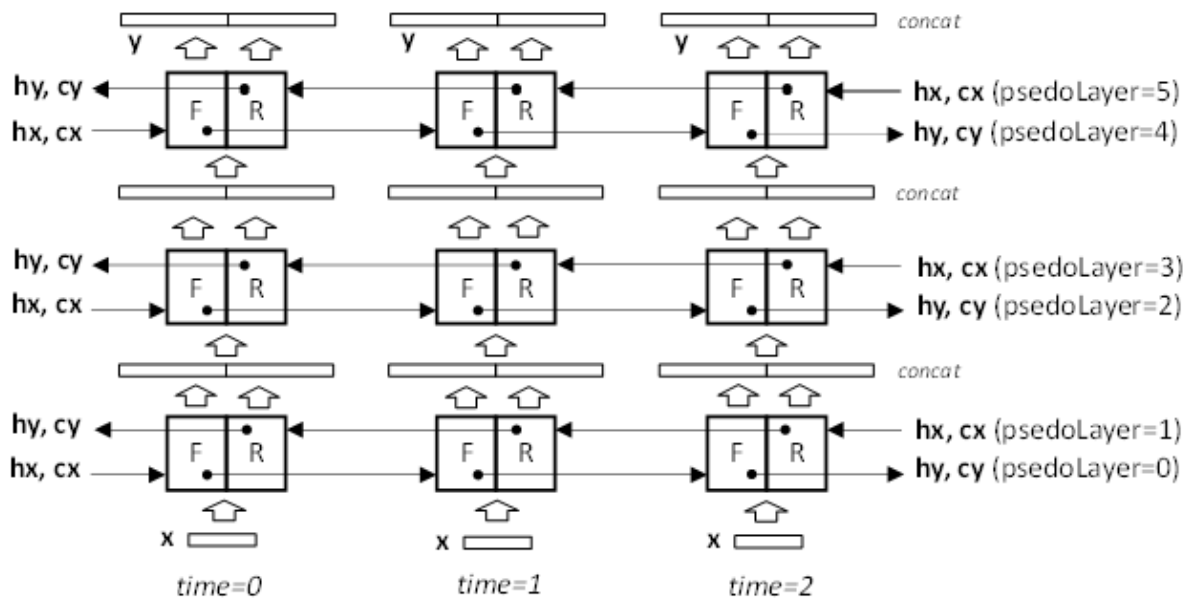
Figure 3. Locations of `x`, `y`, `hx`, `cx`, `hy`, and `cy` Signals in the Multi-Layer RNN Model



The next figure depicts data flow when the RNN model is bidirectional. In this mode each RNN physical layer consists of two consecutive pseudo-layers, each with its own weights, biases, the initial hidden state `hx`, and for LSTM, also the initial cell state `cx`.

Even pseudo-layers 0, 2, 4 process input vectors from left to right or in the forward (F) direction. Odd pseudo-layers 1, 3, 5 process input vectors from right to left or in the reverse (R) direction. Two successive pseudo-layers operate on the same input vectors, just in a different order. Pseudo-layers 0 and 1 access the original sequences stored in the  $x$  buffer. Outputs of F and R cells are concatenated so vectors fed to the next two pseudo-layers have lengths of  $2 \times \text{hiddenSize}$  or  $2 \times \text{projSize}$ . Input GEMMs in subsequent pseudo-layers adjust vector lengths to  $1 \times \text{hiddenSize}$ .

Figure 4. Data Flow when the RNN Model is Bidirectional



When the `fwdMode` parameter is set to `CUDNN_FWD_MODE_TRAINING`, the `cudaRNNForward()` function stores intermediate data required to compute first order derivatives in the reserve space buffer. Work and reserve space buffer sizes should be computed by the `cudaGetRNNTempSpaceSizes()` function with the same `fwdMode` setting as used in the `cudaRNNForward()` call.

The same layout type must be specified in `xDesc` and `yDesc` descriptors. The same sequence lengths must be configured in `xDesc`, `yDesc` and in the device array `devSeqLengths`. Starting in cuDNN 8.9.1, the `devSeqLengths` parameter is no longer required and can be set to `NULL`. The variable sequence length array is transferred automatically to GPU memory by the `cudaRNNForward()` function.

The `cudaRNNForward()` function does not verify that sequence lengths stored in `devSeqLengths` in GPU memory are the same as in `xDesc` and `yDesc` descriptors in CPU memory. Sequence length arrays from `xDesc` and `yDesc` descriptors are checked for consistency, however.

## Parameters

### **handle**

*Input.* The current cuDNN context handle.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **fwdMode**

*Input.* Specifies inference or training mode (`CUDNN_FWD_MODE_INFERENCE` and `CUDNN_FWD_MODE_TRAINING`). In the training mode, additional data is stored in the reserve space buffer. This information is used in the backward pass to compute derivatives.

### **devSeqLengths**

*Input.* A copy of `seqLengthArray` from `xDesc` or `yDesc` RNN data descriptor. The `devSeqLengths` array must be stored in GPU memory as it is accessed asynchronously by GPU kernels, possibly after the `cudnnRNNForward()` function exists. In cuDNN 8.9.1 and later versions, `devSeqLengths` should be `NULL`.

### **xDesc**

*Input.* A previously initialized descriptor corresponding to the RNN model primary input. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` must match that of `yDesc`. The parameter `vectorSize` must match the `inputSize` argument passed to the `cudnnSetRNNDescriptor_v8()` function.

### **x**

*Input.* Data pointer to the GPU memory associated with the RNN data descriptor `xDesc`. The vectors are expected to be arranged in memory according to the layout specified by `xDesc`. The elements in the tensor (including padding vectors) must be densely packed.

### **yDesc**

*Input.* A previously initialized RNN data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` must match that of `xDesc`. The parameter `vectorSize` depends on whether LSTM projection is enabled and whether the network is bi-directional. Specifically:

- ▶ For uni-directional models, the parameter `vectorSize` must match the `hiddenSize` argument passed to `cudnnSetRNNDescriptor_v8()`. If the LSTM projection is enabled, the `vectorSize` must be the same as the `projSize` argument passed to `cudnnSetRNNDescriptor_v8()`.
- ▶ For bi-directional models, if the RNN `cellMode` is `CUDNN_LSTM` and the projection feature is enabled, the parameter `vectorSize` must be 2x the `projSize` argument passed to `cudnnSetRNNDescriptor_v8()`. Otherwise, it should be 2x the `hiddenSize` value.

### **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.

#### **hDesc**

*Input.* A tensor descriptor describing the initial or final hidden state of RNN. Hidden state data are fully packed. The first dimension of the tensor depends on the `dirMode` argument passed to the [cudnnSetRNNDescriptor\\_v8\(\)](#) function.

- ▶ If `dirMode` is `CUDNN_UNIDIRECTIONAL`, then the first dimension should match the `numLayers` argument passed to [cudnnSetRNNDescriptor\\_v8\(\)](#).
- ▶ If `dirMode` is `CUDNN_BIDIRECTIONAL`, then the first dimension should be double the `numLayers` argument passed to [cudnnSetRNNDescriptor\\_v8\(\)](#).

The second dimension must match the `batchSize` parameter described in `xDesc`. The third dimension depends on whether RNN mode is `CUDNN_LSTM` and whether the LSTM projection is enabled. Specifically:

- ▶ If RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the third dimension must match the `projSize` argument passed to the [cudnnSetRNNProjectionLayers\(\)](#) call.
- ▶ Otherwise, the third dimension must match the `hiddenSize` argument passed to the [cudnnSetRNNDescriptor\\_v8\(\)](#) call used to initialize `rnnDesc`.

#### **hx**

*Input.* Pointer to the GPU buffer with the RNN initial hidden state. Data dimensions are described by the `hDesc` tensor descriptor. If a `NULL` pointer is passed, the initial hidden state of the network will be initialized to zero.

#### **hy**

*Output.* Pointer to the GPU buffer where the final RNN hidden state should be stored. Data dimensions are described by the `hDesc` tensor descriptor. If a `NULL` pointer is passed, the final hidden state of the network will not be saved.

#### **cDesc**

*Input.* For LSTM networks only. A tensor descriptor describing the initial or final cell state for LSTM networks only. Cell state data are fully packed. The first dimension of the tensor depends on the `dirMode` argument passed to the [cudnnSetRNNDescriptor\\_v8\(\)](#) call.

- ▶ If `dirMode` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument passed to [cudnnSetRNNDescriptor\\_v8\(\)](#).
- ▶ If `dirMode` is `CUDNN_BIDIRECTIONAL` the first dimension should match double the `numLayers` argument passed to [cudnnSetRNNDescriptor\\_v8\(\)](#).

The second tensor dimension must match the `batchSize` parameter in `xDesc`. The third dimension must match the `hiddenSize` argument passed to the [cudnnSetRNNDescriptor\\_v8\(\)](#) call.

**cx**

*Input.* For LSTM networks only. Pointer to the GPU buffer with the initial LSTM state data. Data dimensions are described by the `cDesc` tensor descriptor. If a `NULL` pointer is passed, the initial cell state of the network will be initialized to zero.

**cy**

*Output.* For LSTM networks only. Pointer to the GPU buffer where final LSTM state data should be stored. Data dimensions are described by the `cDesc` tensor descriptor. If a `NULL` pointer is passed, the final LSTM cell state will not be saved.

**weightSpaceSize**

*Input.* Specifies the size in bytes of the provided weight-space buffer.

**weightSpace**

*Input.* Address of the weight space buffer in GPU memory.

**workSpaceSize**

*Input.* Specifies the size in bytes of the provided workspace buffer.

**workSpace**

*Input/Output.* Address of the workspace buffer in GPU memory to store temporary data.

**reserveSpaceSize**

*Input.* Specifies the size in bytes of the reserve-space buffer.

**reserveSpace**

*Input/Output.* Address of the reserve-space buffer in GPU memory.

## Returns

**CUDNN\_STATUS\_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

At least one of the following conditions are met:

- ▶ variable sequence length input is passed while `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is specified
- ▶ `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is requested on pre-Pascal devices
- ▶ the 'double' floating point type is used for input/output and the `CUDNN_RNN_ALGO_PERSIST_STATIC` algo

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered. For example:

- ▶ some input descriptors are `NULL`
- ▶ at least one of the settings in `rnnDesc`, `xDesc`, `yDesc`, `hDesc`, or `cDesc` descriptors is invalid
- ▶ `weightSpaceSize`, `workSpaceSize`, or `reserveSpaceSize` is too small

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate CPU memory.

## 7.2.37. **cudaRNNForwardInference()**

This function has been deprecated in cuDNN 8.0. Use [cudaRNNForward\(\)](#) instead of `cudaRNNForwardInference()`.

```

cudaStatus_t cudaRNNForwardInference(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_t *xDesc,
    const void            *x,
    const cudaTensorDescriptor_t  hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t  cxDesc,
    const void            *cx,
    const cudaFilterDescriptor_t  wDesc,
    const void            *w,
    const cudaTensorDescriptor_t  *yDesc,
    void                  *y,
    const cudaTensorDescriptor_t  hyDesc,
    void                  *hy,
    const cudaTensorDescriptor_t  cyDesc,
    void                  *cy,
    void                  *workspace,
    size_t                workspaceSizeInBytes)

```

This routine executes the recurrent neural network described by `rnnDesc` with inputs `x`, `hx`, and `cx`, weights `w` and outputs `y`, `hy`, and `cy`. `workspace` is required for intermediate storage. This function does not store intermediate data required for training; [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 the [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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

**cy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `cyDesc`. If a `NULL` pointer is passed, the final cell state of the network will not be saved.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `cxDesc`, `wDesc`, `hyDesc`, `cyDesc` or one of the descriptors in `xDesc`, or `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `cxDesc`, `wDesc`, `yDesc`, `hyDesc`, or `cyDesc` have incorrect strides or dimensions.
- ▶ `workspaceSizeInBytes` is too small.

**CUDNN\_STATUS\_INVALID\_VALUE**

[`cudaSetPersistentRNNPlan\(\)`](#) was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 7.2.38. `cudaRNNForwardInferenceEx()`

This function has been deprecated in cuDNN 8.0. Use [`cudaRNNForward\(\)`](#) instead of `cudaRNNForwardInferenceEx()`.

```

cudaStatus_t cudaRNNForwardInferenceEx(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const cudaRNNDataDescriptor_t  xDesc,
    const void            *x,
    const cudaTensorDescriptor_t  hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_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()` function allows the user to use an 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 an unpacked layout, both sequence major (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function [cudnnRNNForwardInference\(\)](#), the sequences in the mini-batch need to be sorted in descending order according to length.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **xDesc**

*Input.* A previously initialized RNN Data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` need to match that of `yDesc`.

### **x**

*Input.* Data pointer to the GPU memory associated with the RNN data descriptor `xDesc`. The vectors are expected to be laid out in memory according to the layout specified by `xDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

### **hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`:

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

The second dimension must match the `batchSize` parameter described in `xDesc`. The third dimension depends on whether RNN mode is `CUDNN_LSTM` and whether LSTM projection is enabled. Specifically:

- ▶ If RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the third dimension must match the `recProjSize` argument passed to [`cudnnSetRNNProjectionLayers\(\)`](#) call used to set `rnnDesc`.
- ▶ Otherwise, the third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the `batchSize` parameter in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`.

**cx**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `cxDesc`. If a `NULL` pointer is passed, the initial cell state of the network will be initialized to zero.

**wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

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

**yDesc**

*Input.* A previously initialized RNN data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` must match that of `dyDesc` and `dxDesc`. The parameter `vectorSize` depends on whether RNN mode is `CUDNN_LSTM` and whether LSTM projection is enabled and whether the network is bidirectional. Specifically:



- ▶ For uni-directional network, if the 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 a uni-directional network, the parameter `vectorSize` must match the `hiddenSize` argument used to initialize `rnnDesc`. If the network is bidirectional, then multiply the value by 2.

**y**

*Output.* Data pointer to the GPU memory associated with the RNN data descriptor `yDesc`. The vectors are expected to be laid out in memory according to the layout specified by `yDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hyDesc**

*Input.* A fully packed tensor descriptor describing the final hidden state of the RNN. The descriptor must be set exactly the same way as `hxDesc`.

**hy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `hyDesc`. If a `NULL` pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input.* A fully packed tensor descriptor describing the final cell state for LSTM networks. The descriptor must be set exactly the same way as `cxDesc`.

**cy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `cyDesc`. If a `NULL` pointer is passed, the final cell state of the network will not be saved.

**kDesc**

Reserved. User may pass in `NULL`.

**keys**

Reserved. Users may pass in `NULL`.

**cDesc**

Reserved. Users may pass in `NULL`.

**cAttn**

Reserved. Users may pass in `NULL`.

**iDesc**

Reserved. Users may pass in `NULL`.

**iAttn**

Reserved. Users may pass in `NULL`.

**qDesc**

Reserved. Users may pass in `NULL`.

**queries**

Reserved. Users 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`, or `cyDesc` is invalid, or has incorrect strides or dimensions.
- ▶ `reserveSpaceSizeInBytes` is too small.
- ▶ `workspaceSizeInBytes` is too small.

**CUDNN\_STATUS\_INVALID\_VALUE**

`cudaSetPersistentRNNPlan()` was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

**7.2.39. `cudaRNNGetClip()`**

This function has been deprecated in cuDNN 8.0. Use `cudaRNNGetClip_v8()` instead of `cudaRNNGetClip()`.

```

cudaStatus_t cudaRNNGetClip(
    cudaHandle_t          handle,
    cudaRNNDescriptor_t  rnnDesc,
    cudaRNNClipMode_t    *clipMode,
    cudaRNNClipNanOpt_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`.

## 7.2.40. cudnnRNNGetClip\_v8()

Retrieves the current LSTM cell clipping parameters, and stores them in the arguments provided. The user can assign `NULL` to any pointer except `rnnDesc` when the retrieved value is not needed. The function does not check the validity of retrieved parameters.

```
cudnnStatus_t cudnnRNNGetClip_v8(
    cudnnRNNDescriptor_t rnnDesc,
    cudnnRNNClipMode_t *clipMode,
    cudnnNanPropagation_t *clipNanOpt,
    double *lclip,
    double *rclip);
```

## Parameters

### rnnDesc

*Input.* A previously initialized RNN descriptor.

### clipMode

*Output.* Pointer to the location where the retrieved `cudnnRNNClipMode_t` value 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.

### clipNanOpt

*Output.* Pointer to the location where the retrieved `cudnnNanPropagation_t` value is stored.

### lclip, rclip

*Output.* Pointers to the location where the retrieved LSTM cell clipping range [`lclip`, `rclip`] is stored.

## Returns

### CUDNN\_STATUS\_SUCCESS

LSTM clipping parameters were successfully retrieved from the RNN descriptor.

### CUDNN\_STATUS\_BAD\_PARAM

An invalid input argument was found (`rnnDesc` was `NULL`).

## 7.2.41. cudnnRNNSetClip()

This function has been deprecated in cuDNN 8.0. Use [cudnnRNNSetClip\\_v8\(\)](#) instead of `cudnnRNNSetClip()`.

```

cudnnStatus_t cudnnRNNSetClip(
    cudnnHandle_t          handle,
    cudnnRNNDescriptor_t   rnnDesc,
    cudnnRNNClipMode_t     clipMode,
    cudnnNanPropagation_t  clipNanOpt,
    double                 lclip,
    double                 rclip);

```

Sets the LSTM cell clipping mode. The LSTM clipping is disabled by default. When enabled, clipping is applied to all layers. This `cudnnRNNSetClip()` function may be called multiple times.

## Parameters

### clipMode

*Input.* Enables or disables the LSTM cell clipping. When `clipMode` is set to `CUDNN_RNN_CLIP_NONE` no LSTM cell state clipping is performed. When `clipMode` is `CUDNN_RNN_CLIP_MINMAX` the cell state activation to other units is clipped.

### lclip, rclip

*Input.* The range [`lclip`, `rclip`] to which the LSTM cell clipping should be set.

### clipNanOpt

*Input.* When set to `CUDNN_PROPAGATE_NAN` (see the description for [cudnnNanPropagation\\_t](#)), NaN is propagated from the LSTM cell, or it can be set to one of the clipping range boundary values, instead of propagating.

## Returns

### CUDNN\_STATUS\_SUCCESS

The function launched successfully.

### CUDNN\_STATUS\_BAD\_PARAM

Returns this value if `lclip > rclip`; or if either `lclip` or `rclip` is NaN.

## 7.2.42. cudnnRNNSetClip\_v8()

Sets the LSTM cell clipping mode. The LSTM clipping is disabled by default. When enabled, clipping is applied to all layers. This [cudnnRNNSetClip\(\)](#) function does not affect the work, reserve, and weight-space buffer sizes and may be called multiple times.

```

cudnnStatus_t cudnnRNNSetClip_v8(
    cudnnRNNDescriptor_t rnnDesc,
    cudnnRNNClipMode_t clipMode,
    cudnnNanPropagation_t clipNanOpt,
    double lclip,
    double rclip);

```

### Parameters

**rnnDesc**

*Input.* A previously initialized RNN descriptor.

**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 is clipped.

**clipNanOpt**

*Input.* When set to `CUDNN_PROPAGATE_NAN` (see the description for [cudnnNanPropagation\\_t](#)), NaN is propagated from the LSTM cell, or it can be set to one of the clipping range boundary values, instead of propagating.

**lclip, rclip**

*Input.* The range [`lclip`, `rclip`] to which the LSTM cell clipping should be set.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The function completed successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was found, for example:

- ▶ `rnnDesc` was NULL
- ▶ `lclip > rclip`
- ▶ either `lclip` or `rclip` is NaN

## 7.2.43. cudnnSetAttnDescriptor()

This function configures a multi-head attention descriptor that was previously created using the [cudnnCreateAttnDescriptor\(\)](#) function. The function sets attention parameters that are necessary to compute internal buffer sizes, dimensions of weight and bias tensors, or to select optimized code paths.

```

cudnnStatus_t cudnnSetAttnDescriptor(
    cudnnAttnDescriptor_t attnDesc,
    unsigned attnMode,
    int nHeads,
    double smScaler,

```

```

    cudnnDataType_t dataType,
    cudnnDataType_t computePrec,
    cudnnMathType_t mathType,
    cudnnDropoutDescriptor_t attnDropoutDesc,
    cudnnDropoutDescriptor_t postDropoutDesc,
    int qSize,
    int kSize,
    int vSize,
    int qProjSize,
    int kProjSize,
    int vProjSize,
    int oProjSize,
    int qoMaxSeqLength,
    int kvMaxSeqLength,
    int maxBatchSize,
    int maxBeamSize);

```

Input sequence data descriptors in [cudnnMultiHeadAttnForward\(\)](#), [cudnnMultiHeadAttnBackwardData\(\)](#) and [cudnnMultiHeadAttnBackwardWeights\(\)](#) functions are checked against the configuration parameters stored in the attention descriptor. Some parameters must match exactly while `max` arguments such as `maxBatchSize` or `qoMaxSeqLength` establish upper limits for the corresponding dimensions.

The multi-head attention model can be described by the following equations:

$$\mathbf{h}_i = (\mathbf{W}_{V_i} \mathbf{V}) \text{softmax}(\text{smScaler}(\mathbf{K}^T \mathbf{W}_{K_i}^T) (\mathbf{W}_{Q_i} \mathbf{q})), \text{ for } i = 0 \dots \text{nHeads} - 1$$

$$\text{MultiHeadAttn}(\mathbf{q}, \mathbf{K}, \mathbf{V}, \mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_O) = \sum_{i=0}^{\text{nHeads}-1} \mathbf{W}_{O_i} \mathbf{h}_i$$

Where:

- ▶ `nHeads` is the number of independent attention heads that evaluate  $\mathbf{h}_i$  vectors.
- ▶  $\mathbf{q}$  is a primary input, a single `query` column vector.
- ▶  $\mathbf{K}, \mathbf{V}$  are two matrices of `key` and `value` column vectors.

For simplicity, the above equations are presented using a single embedding vector  $\mathbf{q}$  but the cuDNN API can handle multiple  $\mathbf{q}$  candidates in the beam search scheme, process  $\mathbf{q}$  vectors from multiple sequences bundled into a batch, or automatically iterate through all embedding vectors (time-steps) of a sequence. Thus, in general,  $\mathbf{q}, \mathbf{K}, \mathbf{V}$  inputs are tensors with additional pieces of information such as the active length of each sequence or how unused padding vectors should be saved.

In some publications,  $\mathbf{W}_{O_i}$  matrices are combined into one output projection matrix and  $\mathbf{h}_i$  vectors are merged explicitly into a single vector. This is an equivalent notation. In the cuDNN library,  $\mathbf{W}_{O_i}$  matrices are conceptually treated the same way as  $\mathbf{W}_{Q_i}, \mathbf{W}_{K_i}$  or  $\mathbf{W}_{V_i}$  input projection weights. See the description of the [cudnnGetMultiHeadAttnWeights\(\)](#) function for more details.

Weight matrices  $\mathbf{W}_{Q_i}, \mathbf{W}_{K_i}, \mathbf{W}_{V_i}$  and  $\mathbf{W}_{O_i}$  play similar roles, adjusting vector lengths in  $\mathbf{q}, \mathbf{K}, \mathbf{V}$  inputs and in the multi-head attention final output. The user can disable any or all projections by setting `qProjSize`, `kProjSize`, `vProjSize` or `oProjSize` arguments to zero.

Embedding vector sizes in  $\mathbf{q}, \mathbf{K}, \mathbf{V}$  and the vector lengths after projections need to be selected in such a way that matrix multiplications described above are feasible.

Otherwise, `CUDNN_STATUS_BAD_PARAM` is returned by the `cudaSetAttnDescriptor()` function. All four weight matrices are used when it is desirable to maintain rank deficiency of  $\mathbf{W}_{KQ_i} = \mathbf{W}_{K_i}^T \mathbf{W}_{Q_i}$  or  $\mathbf{W}_{OV_i} = \mathbf{W}_{O_i} \mathbf{W}_{V_i}$  matrices to eliminate one or more dimensions during linear transformations in each head. This is a form of feature extraction. In such cases, the projected sizes are smaller than the original vector lengths.

For each attention head, weight matrix sizes are defined as follows:

- ▶  $\mathbf{W}_{Q_i}$  - size [qProjSize x qSize],  $i = 0 \dots nHeads - 1$
- ▶  $\mathbf{W}_{K_i}$  - size [kProjSize x kSize],  $i = 0 \dots nHeads - 1$ , kProjSize = qProjSize
- ▶  $\mathbf{W}_{V_i}$  - size [vProjSize x vSize],  $i = 0 \dots nHeads - 1$
- ▶  $\mathbf{W}_{O_i}$  - size [oProjSize x (vProjSize > 0 ? vProjSize : vSize)],  $i = 0 \dots nHeads - 1$

When the output projection is disabled (`oProjSize = 0`), the output vector length is `nHeads * (vProjSize > 0 ? vProjSize : vSize)`, meaning, the output is a concatenation of all  $\mathbf{h}_i$  vectors. In the alternative interpretation, a concatenated matrix  $\mathbf{W}_O = [\mathbf{W}_{O_0}, \mathbf{W}_{O_1}, \mathbf{W}_{O_2}, \dots]$  forms the identity matrix.

Softmax is a normalized, exponential vector function that takes and outputs vectors of the same size. The multi-head attention API utilizes softmax of the `CUDNN_SOFTMAX_ACCURATE` type to reduce the likelihood of the floating-point overflow.

The `smScaler` parameter is the softmax sharpening/smoothing coefficient. When `smScaler = 1.0`, softmax uses the natural exponential function  $\exp(x)$  or  $2.7183^x$ . When `smScaler < 1.0`, for example `smScaler = 0.2`, the function used by the softmax block will not grow as fast because  $\exp(0.2^x) \approx 1.2214^x$ .

The `smScaler` parameter can be adjusted to process larger ranges of values fed to softmax. When the range is too large (or `smScaler` is not sufficiently small for the given range), the output vector of the softmax block becomes categorical, meaning, one vector element is close to 1.0 and other outputs are zero or very close to zero. When this occurs, the Jacobian matrix of the softmax block is also close to zero so deltas are not back-propagated during training from output to input except through residual connections, if these connections are enabled. The user can set `smScaler` to any positive floating-point value or even zero. The `smScaler` parameter is not trainable.

The `qoMaxSeqLength`, `kvMaxSeqLength`, `maxBatchSize`, and `maxBeamSize` arguments declare the maximum sequence lengths, maximum batch size, and maximum beam size respectively, in the `cudaSeqDataDescriptor_t` containers. The actual dimensions supplied to forward and backward (gradient) API functions should not exceed the `max` limits. The `max` arguments should be set carefully because too large values will result in excessive memory usage due to oversized work and reserve space buffers.

The `attnMode` argument is treated as a binary mask where various on/off options are set. These options can affect the internal buffer sizes, enforce certain argument checks, select optimized code execution paths, or enable attention variants that do not require additional numerical arguments. An example of such options is the inclusion of biases in input and output projections.

The `attnDropoutDesc` and `postDropoutDesc` arguments are descriptors that define two dropout layers active in the training mode. The first dropout operation defined

by `attnDropoutDesc`, is applied directly to the softmax output. The second dropout operation, specified by `postDropoutDesc`, alters the multi-head attention output, just before the point where residual connections are added.



**Note:** The `cudnnSetAttnDescriptor()` function performs a shallow copy of `attnDropoutDesc` and `postDropoutDesc`, meaning, the addresses of both dropout descriptors are stored in the attention descriptor and not the entire structure. Therefore, the user should keep dropout descriptors during the entire life of the attention descriptor.

## Parameters

### **attnDesc**

*Output.* Attention descriptor to be configured.

### **attnMode**

*Input.* Enables various attention options that do not require additional numerical values. See the table below for the list of supported flags. The user should assign a preferred set of bitwise OR-ed flags to this argument.

### **nHeads**

*Input.* Number of attention heads.

### **smScaler**

*Input.* Softmax smoothing ( $1.0 \geq \text{smScaler} \geq 0.0$ ) or sharpening ( $\text{smScaler} > 1.0$ ) coefficient. Negative values are not accepted.

### **dataType**

*Input.* Data type used to represent attention inputs, attention weights and attention outputs.

### **computePrec**

*Input.* Compute precision.

### **mathType**

*Input.* NVIDIA Tensor Core settings.

### **attnDropoutDesc**

*Input.* Descriptor of the dropout operation applied to the softmax output. See the table below for a list of unsupported features.

### **postDropoutDesc**

*Input.* Descriptor of the dropout operation applied to the multi-head attention output, just before the point where residual connections are added. See the table below for a list of unsupported features.

### **qSize, kSize, vSize**

*Input.* **Q**, **K**, **V** embedding vector lengths.

### **qProjSize, kProjSize, vProjSize**

*Input.* **Q**, **K**, **V** embedding vector lengths after input projections. Use zero to disable the corresponding projection.

### **oProjSize**

*Input.* The **h<sub>i</sub>** vector length after the output projection. Use zero to disable this projection.



**qoMaxSeqLength**

*Input.* Largest sequence length expected in sequence data descriptors related to **Q**, **O**, **dQ** and **dO** inputs and outputs.

**kvMaxSeqLength**

*Input.* Largest sequence length expected in sequence data descriptors related to **K**, **V**, **dK** and **dV** inputs and outputs.

**maxBatchSize**

*Input.* Largest batch size expected in any `cudaSeqDataDescriptor_t` container.

**maxBeamSize**

*Input.* Largest beam size expected in any `cudaSeqDataDescriptor_t` container.

Supported `attnMode` flags

**CUDNN\_ATTN\_QUERYMAP\_ALL\_TO\_ONE**

Forward declaration of mapping between **Q** and **K**, **V** vectors when the beam size is greater than one in the **Q** input. Multiple **Q** vectors from the same beam bundle map to the same **K**, **V** vectors. This means that beam sizes in the **K**, **V** sets are equal to one.

**CUDNN\_ATTN\_QUERYMAP\_ONE\_TO\_ONE**

Forward declaration of mapping between **Q** and **K**, **V** vectors when the beam size is greater than one in the **Q** input. Multiple **Q** vectors from the same beam bundle map to different **K**, **V** vectors. This requires beam sizes in **K**, **V** sets to be the same as in the **Q** input.

**CUDNN\_ATTN\_DISABLE\_PROJ\_BIASES**

Use no biases in the attention input and output projections.

**CUDNN\_ATTN\_ENABLE\_PROJ\_BIASES**

Use extra biases in the attention input and output projections. In this case the projected  $\bar{\mathbf{K}}$  vectors are computed as  $\bar{\mathbf{K}}_i = \mathbf{W}_{Kj} \mathbf{K} + \mathbf{b} * [1, 1, \dots, 1]_{1 \times n}$ , where  $n$  is the number of columns in the **K** matrix. In other words, the same column vector **b** is added to all columns of **K** after the weight matrix multiplication.

Supported combinations of `dataType`, `computePrec`, and `mathType`

Table 50. Supported Combinations for `cudaSetAttnDescriptor()`

<code>dataType</code>	<code>computePrec</code>	<code>mathType</code>
CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DEFAULT_MATH
CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION
CUDNN_DATA_HALF	CUDNN_DATA_HALF, CUDNN_DATA_FLOAT	CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH, CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION

## Unsupported features

1. The `paddingFill` argument in `cudaSeqDataDescriptor_t` is currently ignored by all multi-head attention functions.

## Returns

### CUDNN\_STATUS\_SUCCESS

The attention descriptor was configured successfully.

### CUDNN\_STATUS\_BAD\_PARAM

An invalid input argument was encountered. Some examples include:

- ▶ post projection **Q** and **K** sizes were not equal
- ▶ `dataType`, `computePrec`, or `mathType` were invalid
- ▶ one or more of the following arguments were either negative or zero: `nHeads`, `qSize`, `kSize`, `vSize`, `qoMaxSeqLength`, `kvMaxSeqLength`, `maxBatchSize`, `maxBeamSize`
- ▶ one or more of the following arguments were negative: `qProjSize`, `kProjSize`, `vProjSize`, `smScaler`

### CUDNN\_STATUS\_NOT\_SUPPORTED

A requested option or a combination of input arguments is not supported.

## 7.2.44. `cudaSetPersistentRNNPlan()`

This function has been deprecated in cuDNN 8.0.

This function sets the persistent RNN plan to be executed when using `rnnDesc` and `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` algo.

```
cudaStatus_t cudaSetPersistentRNNPlan(
    cudaRNNDescriptor_t    rnnDesc,
    cudaPersistentRNNPlan_t plan)
```

## 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`.

## 7.2.45. `cudaSetRNNAlgorithmDescriptor()`

This function has been deprecated in cuDNN 8.0.

## 7.2.46. cudnnSetRNNBiasMode()

This function has been deprecated in cuDNN 8.0. Use [cudnnSetRNNDescriptor\\_v8\(\)](#) instead of `cudnnSetRNNBiasMode()`.

```

cudnnStatus_t cudnnSetRNNBiasMode(
    cudnnRNNDescriptor_t  rnnDesc,
    cudnnRNNBiasMode_t   biasMode)

```

The `cudnnSetRNNBiasMode()` function sets the number of bias vectors for a previously created and initialized RNN descriptor. This function should be called to enable the specified bias mode in an RNN. The default value of `biasMode` in `rnnDesc` after [cudnnCreateRNNDescriptor\(\)](#) is `CUDNN_RNN_DOUBLE_BIAS`.

### Parameters

**rnnDesc**

*Input/Output.* A previously created RNN descriptor.

**biasMode**

*Input.* Sets the number of bias vectors. For more information, refer to [cudnnRNNBiasMode\\_t](#).

### Returns

**CUDNN\_STATUS\_BAD\_PARAM**

Either the `rnnDesc` is `NULL` or `biasMode` has an invalid enumerant value.

**CUDNN\_STATUS\_SUCCESS**

The `biasMode` was set successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

Non-default bias mode (an enumerated type besides `CUDNN_RNN_DOUBLE_BIAS`) applied to an RNN algo other than `CUDNN_RNN_ALGO_STANDARD`.

## 7.2.47. cudnnSetRNNDataDescriptor()

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.

```

cudnnStatus_t cudnnSetRNNDataDescriptor(
    cudnnRNNDataDescriptor_t  RNNDataDesc,
    cudnnDataType_t          dataType,
    cudnnRNNDataLayout_t     layout,
    int                      maxSeqLength,
    int                      batchSize,
    int                      vectorSize,
    const int                seqLengthArray[],
    void                     *paddingFill);

```

## Parameters

### **RNNDataDesc**

*Input/Output.* A previously created RNN descriptor. For more information, refer to [cudnnRNNDataDescriptor\\_t](#).

### **dataType**

*Input.* The datatype of the RNN data tensor. For more information, refer to [cudnnDataType\\_t](#).

### **layout**

*Input.* The memory layout of the RNN data tensor.

### **maxSeqLength**

*Input.* The maximum sequence length within this RNN data tensor. In the unpacked (padded) layout, this should include the padding vectors in each sequence. In the packed (unpadded) layout, this should be equal to the greatest element in `seqLengthArray`.

### **batchSize**

*Input.* The number of sequences within the mini-batch.

### **vectorSize**

*Input.* The vector length (embedding size) of the input or output tensor at each time-step.

### **seqLengthArray**

*Input.* An integer array with `batchSize` number of elements. Describes the length (number of time-steps) of each sequence. Each element in `seqLengthArray` must be greater than or equal to 0 but less than or equal to `maxSeqLength`. In the packed layout, the elements should be sorted in descending order, similar to the layout required by the non-extended RNN compute functions.

### **paddingFill**

*Input.* A user-defined symbol for filling the padding position in RNN output. This is only effective when the descriptor is describing the RNN output, and the unpacked layout is specified. The symbol should be in the host memory, and is interpreted as the same data type as that of the RNN data tensor. If a `NULL` pointer is passed in, then the padding position in the output will be undefined.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

Any one of these have occurred:

- ▶ `dataType` is not one of `CUDNN_DATA_HALF`, `CUDNN_DATA_FLOAT`, `CUDNN_DATA_DOUBLE`.
- ▶ `maxSeqLength` is larger than 65535 (0xffff).

**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 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.


## 7.2.48. cudnnSetRNNDescriptor\_v6()

This function has been deprecated in cuDNN 8.0. Use [cudnnSetRNNDescriptor\\_v8\(\)](#) instead of cudnnSetRNNDescriptor\_v6().

```

cudnnStatus_t cudnnSetRNNDescriptor_v6(
    cudnnHandle_t      handle,
    cudnnRNNDescriptor_t  rnnDesc,
    const int          hiddenSize,
    const int          numLayers,
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnRNNInputMode_t  inputMode,
    cudnnDirectionMode_t direction,
    cudnnRNNMode_t      mode,
    cudnnRNNAlgo_t      algo,
    cudnnDataType_t     mathPrec)
    
```

This function initializes a previously created RNN descriptor object.



Note: Larger networks, for example, longer sequences or more layers, are expected to be more efficient than smaller networks.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN library descriptor.

**rnnDesc**

*Input/Output.* A previously created RNN descriptor.

**hiddenSize**

*Input.* Size of the internal hidden state for each layer.

**numLayers**

*Input.* Number of stacked layers.

**dropoutDesc**

*Input.* Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers, for example, a single layer network will have no dropout applied.

**inputMode**

*Input.* Specifies the behavior at the input to the first layer.

**direction**

*Input.* Specifies the recurrence pattern, for example, bidirectional.

**mode**

*Input.* Specifies the type of RNN to compute.

**algo**

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

**mathPrec**

*Input.* Math precision. This parameter is used for controlling the math precision in RNN. The following applies:

- ▶ For the input/output in FP16, the parameter `mathPrec` can be `CUDNN_DATA_HALF` or `CUDNN_DATA_FLOAT`.
- ▶ For the input/output in FP32, the parameter `mathPrec` can only be `CUDNN_DATA_FLOAT`.
- ▶ For the input/output in FP64, double type, the parameter `mathPrec` can only be `CUDNN_DATA_DOUBLE`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The object was set successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

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

## 7.2.49. cudnnSetRNNDescriptor\_v8()

This function initializes a previously created RNN descriptor object. The RNN descriptor configured by `cudnnSetRNNDescriptor_v8()` was enhanced to store all information needed to compute the total number of adjustable weights/biases in the RNN model.

```

cudnnStatus_t cudnnSetRNNDescriptor_v8(
    cudnnRNNDescriptor_t rnnDesc,
    cudnnRNNAlgo_t algo,
    cudnnRNNMode_t cellMode,
    cudnnRNNBiasMode_t biasMode,
    cudnnDirectionMode_t dirMode,
    cudnnRNNInputMode_t inputMode,
    cudnnDataType_t dataType,
    cudnnDataType_t mathPrec,
    cudnnMathType_t mathType,
    int32_t inputSize,
    int32_t hiddenSize,
    int32_t projSize,
    int32_t numLayers,
    cudnnDropoutDescriptor_t dropoutDesc,
    uint32_t auxFlags);
    
```

## Parameters

### rnnDesc

*Input.* A previously initialized RNN descriptor.

### algo

*Input.* RNN algo (CUDNN\_RNN\_ALGO\_STANDARD, CUDNN\_RNN\_ALGO\_PERSIST\_STATIC, or CUDNN\_RNN\_ALGO\_PERSIST\_DYNAMIC).

### cellMode

*Input.* Specifies the RNN cell type in the entire model (CUDNN\_RNN\_RELU, CUDNN\_RNN\_TANH, CUDNN\_RNN\_LSTM, CUDNN\_RNN\_GRU).

### biasMode

*Input.* Sets the number of bias vectors (CUDNN\_RNN\_NO\_BIAS, CUDNN\_RNN\_SINGLE\_INP\_BIAS, CUDNN\_RNN\_SINGLE\_REC\_BIAS, CUDNN\_RNN\_DOUBLE\_BIAS). The two single bias settings are functionally the same for RELU, TANH and LSTM cell types. For differences in GRU cells, see the description of CUDNN\_GRU in the [cudnnRNNMode\\_t](#) enumerated type.

### dirMode

*Input.* Specifies the recurrence pattern: CUDNN\_UNIDIRECTIONAL or CUDNN\_BIDIRECTIONAL. In bidirectional RNNs, the hidden states passed between physical layers are concatenations of forward and backward hidden states.

### inputMode

*Input.* Specifies how the input to the RNN model is processed by the first layer. When `inputMode` is CUDNN\_LINEAR\_INPUT, original input vectors of size `inputSize` are multiplied by the weight matrix to obtain vectors of `hiddenSize`. When `inputMode` is CUDNN\_SKIP\_INPUT, the original input vectors to the first layer are used as is without multiplying them by the weight matrix.

### dataType

*Input.* Specifies data type for RNN weights/biases and input and output data.

### mathPrec

*Input.* This parameter is used to control the compute math precision in the RNN model. The following applies:

- ▶ For the input/output in FP16, the parameter `mathPrec` can be CUDNN\_DATA\_HALF or CUDNN\_DATA\_FLOAT.
- ▶ For the input/output in FP32, the parameter `mathPrec` can only be CUDNN\_DATA\_FLOAT.
- ▶ For the input/output in FP64, double type, the parameter `mathPrec` can only be CUDNN\_DATA\_DOUBLE.

### mathType

*Input.* Sets the preferred option to use NVIDIA Tensor Cores accelerators on Volta (SM 7.0) or higher GPU-s).

- ▶ When `dataType` is `CUDNN_DATA_HALF`, the `mathType` parameter can be `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH`. The `ALLOW_CONVERSION` setting is treated the same as `CUDNN_TENSOR_OP_MATH` for this data type.
- ▶ When `dataType` is `CUDNN_DATA_FLOAT`, the `mathType` parameter can be `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`. When the latter settings are used, original weights and intermediate results will be down-converted to `CUDNN_DATA_HALF` before they are used in another recursive iteration.
- ▶ When `dataType` is `CUDNN_DATA_DOUBLE`, the `mathType` parameter can be `CUDNN_DEFAULT_MATH`.

This option has an advisory status meaning Tensor Cores may not be always utilized, for example, due to specific GEMM dimensions restrictions.

**inputSize**

*Input.* Size of the input vector in the RNN model. When the `inputMode=CUDNN_SKIP_INPUT`, the `inputSize` should match the `hiddenSize` value.

**hiddenSize**

*Input.* Size of the hidden state vector in the RNN model. The same hidden size is used in all RNN layers.

**projSize**

*Input.* The size of the LSTM cell output after the recurrent projection. This value should not be larger than `hiddenSize`. It is legal to set `projSize` equal to `hiddenSize`, however, in this case, the recurrent projection feature is disabled. 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 `projSize` 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 recurrent projection can be enabled for LSTM cells and `CUDNN_RNN_ALGO_STANDARD` only.

**numLayers**

*Input.* Number of stacked, physical layers in the deep RNN model. When `dirMode=CUDNN_BIDIRECTIONAL`, the physical layer consists of two pseudo-layers corresponding to forward and backward directions.

**dropoutDesc**

*Input.* Handle to a previously created and initialized dropout descriptor. Dropout operation will be applied between physical layers. A single layer network will have no dropout applied. Dropout is used in the training mode only.

**auxFlags**

*Input.* This argument is used to pass miscellaneous switches that do not require additional numerical values to configure the corresponding feature. In future cuDNN releases, this parameter will be used to extend the RNN functionality without adding new API functions (applicable options should be bitwise OR-ed). Currently, this parameter is used to enable or disable padded input/output (`CUDNN_RNN_PADDED_IO_DISABLED`, `CUDNN_RNN_PADDED_IO_ENABLED`). When the



padded I/O is enabled, layouts `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` and `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED` are permitted in RNN data descriptors.

## Returns

### `CUDNN_STATUS_SUCCESS`

The RNN descriptor was configured successfully.

### `CUDNN_STATUS_BAD_PARAM`

An invalid input argument was detected.

### `CUDNN_STATUS_NOT_SUPPORTED`

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

### `CUDNN_STATUS_EXECUTION_FAILED`

An incompatible or unsupported combination of input arguments was detected.

## 7.2.50. `cudaSetRNNMatrixMathType()`

This function has been deprecated in cuDNN 8.0. Use [`cudaSetRNNDescrptor\_v8\(\)`](#) instead of `cudaSetRNNMatrixMathType()`.

```
cudaStatus_t cudaSetRNNMatrixMathType(
    cudaRNNDescrptor_t  rnnDesc,
    cudaMathType_t      mType)
```

This function sets the preferred option to use NVIDIA Tensor Cores accelerators on Volta GPUs (SM 7.0 or higher). When the `mType` parameter is `CUDNN_TENSOR_OP_MATH`, inference and training RNN APIs will attempt use Tensor Cores when weights/biases are of type `CUDNN_DATA_HALF` or `CUDNN_DATA_FLOAT`. When RNN weights/biases are stored in the `CUDNN_DATA_FLOAT` format, the original weights and intermediate results will be down-converted to `CUDNN_DATA_HALF` before they are used in another recursive iteration.

## Parameters

### `rnnDesc`

*Input.* A previously created and initialized RNN descriptor.

### `mType`

*Input.* A preferred compute option when performing RNN GEMMs (general matrix-matrix multiplications). This option has an advisory status meaning that Tensor Cores may not be utilized, for example, due to specific GEMM dimensions.

## Returns

### `CUDNN_STATUS_SUCCESS`

The preferred compute option for the RNN network was set successfully.

### `CUDNN_STATUS_BAD_PARAM`

An invalid input parameter was detected.

## 7.2.51. cudnnSetRNNPaddingMode()

This function has been deprecated in cuDNN 8.0. Use [cudnnSetRNNDescriptor\\_v8\(\)](#) instead of `cudnnSetRNNPaddingMode()`.

```

cudnnStatus_t cudnnSetRNNPaddingMode(
    cudnnRNNDescriptor_t      rnnDesc,
    cudnnRNNPaddingMode_t    paddingMode)
    
```

This function enables or disables the padded RNN input/output for a previously created and initialized RNN descriptor. This information is required before calling the [cudnnGetRNNWorkspaceSize\(\)](#) and [cudnnGetRNNTrainingReserveSize\(\)](#) functions, to determine whether additional workspace and training reserve space is needed. By default, the padded RNN input/output is not enabled.

### Parameters

**rnnDesc**

*Input/Output.* A previously created RNN descriptor.

**paddingMode**

*Input.* Enables or disables the padded input/output. For more information, refer to [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.

## 7.2.52. cudnnSetRNNProjectionLayers()

This function has been deprecated in cuDNN 8.0. Use [cudnnSetRNNDescriptor\\_v8\(\)](#) instead of `cudnnSetRNNProjectionLayers()`.

```

cudnnStatus_t cudnnSetRNNProjectionLayers(
    cudnnHandle_t      handle,
    cudnnRNNDescriptor_t  rnnDesc,
    int                recProjSize,
    int                outProjSize)
    
```

The `cudnnSetRNNProjectionLayers()` function should be called 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 (refer to [cudnnRNNMode\\_t](#) type), such as  $W_i$  in hidden RNN layers or  $R_j$  in the entire network, become rectangular versus square in the canonical LSTM mode. Obviously, the result of  $R_j * W_r$  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. 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, refer to 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`.

### **outProjSize**

*Input.* This parameter should be zero.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

RNN projection parameters were set successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

An invalid input argument was detected (for example, `NULL` handles, negative values for projection parameters).

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

Projection applied to RNN algo other than `CUDNN_RNN_ALGO_STANDARD`, cell type other than `CUDNN_LSTM`, `recProjSize` larger than `hiddenSize`.

## 7.2.53. `cudaSetSeqDataDescriptor()`

This function initializes a previously created sequence data descriptor object. In the most simplified view, this descriptor defines dimensions (`dimA`) and the data layout (`axes`) of a four-dimensional tensor.

```
cudaStatus_t cudaSetSeqDataDescriptor(
    cudaSeqDataDescriptor_t seqDataDesc,
    cudaDataType_t dataType,
    int nbDims,
    const int dimA[],
    const cudaSeqDataAxis_t axes[],
    size_t seqLengthArraySize,
    const int seqLengthArray[],
```

```
void *paddingFill);
```

All four dimensions of the sequence data descriptor have unique identifiers that can be used to index the `dimA[]` array:

```
CUDNN_SEQDATA_TIME_DIM
CUDNN_SEQDATA_BATCH_DIM
CUDNN_SEQDATA_BEAM_DIM
CUDNN_SEQDATA_VECT_DIM
```

For example, to express information that vectors in our sequence data buffer are five elements long, we need to assign `dimA[CUDNN_SEQDATA_VECT_DIM]=5` in the `dimA[]` array.

The number of active dimensions in the `dimA[]` and `axes[]` arrays is defined by the `nbDims` argument. Currently, the value of this argument should be four. The actual size of the `dimA[]` and `axes[]` arrays should be declared using the `CUDNN_SEQDATA_DIM_COUNT` macro.

The `cudaSeqDataDescriptor_t` container is treated as a collection of fixed length vectors that form sequences, similarly to words (vectors of characters) constructing sentences. The `TIME` dimension spans the sequence length. Different sequences are bundled together in a batch. A `BATCH` may be a group of individual sequences or beams. A `BEAM` is a cluster of alternative sequences or candidates. When thinking about the beam, consider a translation task from one language to another. You may want to keep around and experiment with several translated versions of the original sentence before selecting the best one. The number of candidates kept around is the `BEAM` size.

Every sequence can have a different length, even within the same beam, so vectors toward the end of the sequence can be just padding. The `paddingFill` argument specifies how the padding vectors should be written in output sequence data buffers. The `paddingFill` argument points to one value of type `dataType` that should be copied to all elements in padding vectors. Currently, the only supported value for `paddingFill` is `NULL` which means this option should be ignored. In this case, elements of the padding vectors in output buffers will have undefined values.

It is assumed that a non-empty sequence always starts from the time index zero. The `seqLengthArray[]` must specify all sequence lengths in the container so the total size of this array should be `dimA[CUDNN_SEQDATA_BATCH_DIM] * dimA[CUDNN_SEQDATA_BEAM_DIM]`. Each element of the `seqLengthArray[]` array should have a non-negative value, less than or equal to `dimA[CUDNN_SEQDATA_TIME_DIM]`; the maximum sequence length. Elements in `seqLengthArray[]` are always arranged in the same batch-major order, meaning, when considering `BEAM` and `BATCH` dimensions, `BATCH` is the outer or the slower changing index when we traverse the array in ascending order of the addresses. Using a simple example, the `seqLengthArray[]` array should hold sequence lengths in the following order:

```
{batch_idx=0, beam_idx=0}
{batch_idx=0, beam_idx=1}
{batch_idx=1, beam_idx=0}
{batch_idx=1, beam_idx=1}
{batch_idx=2, beam_idx=0}
{batch_idx=2, beam_idx=1}
```

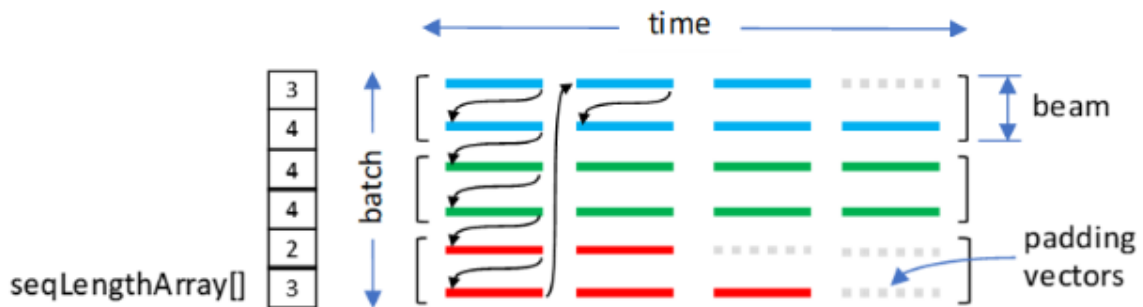
when `dimA[CUDNN_SEQDATA_BATCH_DIM]=3` and `dimA[CUDNN_SEQDATA_BEAM_DIM]=2`.

Data stored in the `cudaSeqDataDescriptor_t` container must comply with the following constraints:

- ▶ All data is fully packed. There are no unused spaces or gaps between individual vector elements or consecutive vectors.
- ▶ The most inner dimension of the container is the vector. In other words, the first contiguous group of `dimA[CUDNN_SEQDATA_VECT_DIM]` elements belongs to the first vector, followed by elements of the second vector, and so on.

The `axes` argument in the `cudaSetSeqDataDescriptor()` function is a bit more complicated. This array should have the same capacity as `dimA[]`. The `axes[]` array specifies the actual data layout in the GPU memory. In this function, the layout is described in the following way: as we move from one element of a vector to another in memory by incrementing the element pointer, what is the order of `VECT`, `TIME`, `BATCH`, and `BEAM` dimensions that we encounter. Let us assume that we want to define the following data layout:

Figure 5. Data Layout Example for `cudaSetSeqDataDescriptor()`



that corresponds to tensor dimensions:

```
int dimA[CUDNN_SEQDATA_DIM_COUNT];
dimA[CUDNN_SEQDATA_TIME_DIM] = 4;
dimA[CUDNN_SEQDATA_BATCH_DIM] = 3;
dimA[CUDNN_SEQDATA_BEAM_DIM] = 2;
dimA[CUDNN_SEQDATA_VECT_DIM] = 5;
```

Now, let's initialize the `axes[]` array. Note that the most inner dimension is described by the last active element of `axes[]`. There is only one valid configuration here as we always traverse a full vector first. Thus, we need to write `CUDNN_SEQDATA_VECT_DIM` in the last active element of `axes[]`.

```
cudaSeqDataAxis_t axes[CUDNN_SEQDATA_DIM_COUNT];
axes[3] = CUDNN_SEQDATA_VECT_DIM; // 3 = nbDims-1
```

Now, let's work on the remaining three elements of `axes[]`. When we reach the end of the first vector, we jump to the next beam, therefore:

```
axes[2] = CUDNN_SEQDATA_BEAM_DIM;
```

When we approach the end of the second vector, we move to the next batch, therefore:

```
axes[1] = CUDNN_SEQDATA_BATCH_DIM;
```

The last (outermost) dimension is `TIME`:

```
axes[0] = CUDNN_SEQDATA_TIME_DIM;
```

The four values of the `axes[]` array fully describe the data layout depicted in the figure.

The sequence data descriptor allows the user to select  $3! = 6$  different data layouts or permutations of BEAM, BATCH and TIME dimensions. The multi-head attention API supports all six layouts.

## Parameters

### **seqDataDesc**

*Output.* Pointer to a previously created sequence data descriptor.

### **dataType**

*Input.* Data type of the sequence data buffer (CUDA\_DATA\_HALF, CUDA\_DATA\_FLOAT or CUDA\_DATA\_DOUBLE).

### **nbDims**

*Input.* Must be 4. The number of active dimensions in dimA[] and axes[] arrays. Both arrays should be declared to contain at least CUDA\_SEQDATA\_DIM\_COUNT elements.

### **dimA[]**

*Input.* Integer array specifying sequence data dimensions. Use the [cudaSeqDataAxis\\_t](#) enumerated type to index all active dimA[] elements.

### **axes[]**

*Input.* Array of [cudaSeqDataAxis\\_t](#) that defines the layout of sequence data in memory. The first nbDims elements of axes[] should be initialized with the outermost dimension in axes[0] and the innermost dimension in axes[nbDims-1].

### **seqLengthArraySize**

*Input.* Number of elements in the sequence length array, seqLengthArray[].

### **seqLengthArray[]**

*Input.* An integer array that defines all sequence lengths of the container.

### **paddingFill**

*Input.* Must be NULL. Pointer to a value of dataType that is used to fill up output vectors beyond the valid length of each sequence or NULL to ignore this setting.

## Returns

### **CUDA\_STATUS\_SUCCESS**

All input arguments were validated and the sequence data descriptor was successfully updated.

### **CUDA\_STATUS\_BAD\_PARAM**

An invalid input argument was found. Some examples include:

- ▶ seqDataDesc=NULL
- ▶ dataType was not a valid type of [cudaDataType\\_t](#)
- ▶ nbDims was negative or zero
- ▶ seqLengthArraySize did not match the expected length
- ▶ some elements of seqLengthArray[] were invalid

### **CUDA\_STATUS\_NOT\_SUPPORTED**

An unsupported input argument was encountered. Some examples include:

- ▶ nbDims is not equal to 4
- ▶ paddingFill is not NULL

**CUDNN\_STATUS\_ALLOC\_FAILED**

Failed to allocate storage for the sequence data descriptor object.

---

# Chapter 8. `cudaAdvTrain` Library

This entity contains all the training counterparts of `cudaAdvInfer`. The `cudaAdvTrain` library depends on `cudaOpsInfer`, `cudaOpsTrain`, and `cudaAdvInfer`.

## 8.1. Data Type References

These are the data type references in the `cudaAdvTrain` library.

### 8.1.1. Enumeration Types

These are the enumeration types in the `cudaAdvTrain` library.

#### 8.1.1.1. `cudaLossNormalizationMode_t`

`cudaLossNormalizationMode_t` is an enumerated type that controls the input normalization mode for a loss function. This type can be used with [`cudaSetCTCLossDescriptorEx\(\)`](#).

#### Values

##### `CUDNN_LOSS_NORMALIZATION_NONE`

The input `probs` of the [`cudaCTCLoss\(\)`](#) function is expected to be the normalized probability, and the output `gradients` is the gradient of loss with respect to the unnormalized probability.

##### `CUDNN_LOSS_NORMALIZATION_SOFTMAX`

The input `probs` of the [`cudaCTCLoss\(\)`](#) function is expected to be the unnormalized activation from the previous layer, and the output `gradients` is the gradient with respect to the activation. Internally the probability is computed by softmax normalization.



### 8.1.1.2. `cudnnWgradMode_t`

`cudnnWgradMode_t` is an enumerated type that selects how buffers holding gradients of the loss function, computed with respect to trainable parameters, are updated.

Currently, this type is used by the `cudnnMultiHeadAttnBackwardWeights()` and `cudnnRNNBackwardWeights_v8()` functions only.

#### Values

##### **CUDNN\_WGRAD\_MODE\_ADD**

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

##### **CUDNN\_WGRAD\_MODE\_SET**

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

## 8.2. API Functions

These are the API functions in the `cudnn_adv_train.so` library.

### 8.2.1. `cudnnAdvTrainVersionCheck()`

This function checks whether the version of the `AdvTrain` subset of the library is consistent with the other sub-libraries.

```
cudnnStatus_t cudnnAdvTrainVersionCheck(void)
```

#### Returns

##### **CUDNN\_STATUS\_SUCCESS**

The version is consistent with other sub-libraries.

##### **CUDNN\_STATUS\_VERSION\_MISMATCH**

The version of `AdvTrain` is not consistent with other sub-libraries. Users should check the installation and make sure all sub-component versions are consistent.

### 8.2.2. `cudnnCreateCTCLossDescriptor()`

This function creates a CTC loss function descriptor.

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

## Parameters

### ctcLossDesc

*Output.* CTC loss descriptor to be set. For more information, refer to [cudaCTCLossDescriptor\\_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.

## 8.2.3. `cudaCTCLoss()`

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

```

cudaStatus_t cudaCTCLoss(
    cudaHandle_t          handle,
    const cudaTensorDescriptor_t  probsDesc,
    const void*          *probs,
    const int            hostLabels[],
    const int            hostLabelLengths[],
    const int            hostInputLengths[],
    void*                *costs,
    const cudaTensorDescriptor_t  gradientsDesc,
    const void*          *gradients,
    cudaCTCLossAlgo_t    algo,
    const cudaCTCLossDescriptor_t  ctcLossDesc,
    void*                *workspace,
    size_t               *workspaceSizeInBytes)
    
```



Note: This function can have an inconsistent interface depending on the [cudaLossNormalizationMode\\_t](#) chosen (bound to the [cudaCTCLossDescriptor\\_t](#) with [cudaSetCTCLossDescriptorEx\(\)](#)). For the `CUDNN_LOSS_NORMALIZATION_NONE`, this function has an inconsistent interface, for example, the probs input is probability normalized by softmax, but the gradients output is with respect to the unnormalized activation. However, for `CUDNN_LOSS_NORMALIZATION_SOFTMAX`, the function has a consistent interface; all values are normalized by softmax.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudaHandle\\_t](#).

### probsDesc

*Input.* Handle to the previously initialized probabilities tensor descriptor. For more information, refer to [cudaTensorDescriptor\\_t](#).

**probs**

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

**hostLabels**

*Input.* Pointer to a previously initialized labels list, in CPU memory.

**hostLabelLengths**

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

**hostInputLengths**

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

**costs**

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

**gradientsDesc**

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

**gradients**

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

**algo**

*Input.* Enumerant that specifies the chosen CTC loss algorithm. For more information, refer to [cudnnCTCLossAlgo\\_t](#).

**ctcLossDesc**

*Input.* Handle to the previously initialized CTC loss descriptor. For more information, refer to [cudnnCTCLossDescriptor\\_t](#).

**workspace**

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

**sizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions of `probsDesc` do not match the dimensions of `gradientsDesc`.
- ▶ The `inputLengths` do not agree with the first dimension of `probsDesc`.
- ▶ The `workSpaceSizeInBytes` is not sufficient.
- ▶ The `labelLengths` is greater than 255.

**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.

## 8.2.4. `cudaCTCLoss_v8()`

This function returns the CTC costs and gradients, given the probabilities and labels. Many CTC API functions were updated in v8 with the `_v8` suffix to support CUDA graphs. Label and input data is now passed in GPU memory, and `cudaCTCLossDescriptor_t` should be set using `cudaSetCTCLossDescriptor_v8()`.

```

cudaStatus_t cudaCTCLoss_v8(
    cudaHandle_t          handle,
    cudaCTCLossAlgo_t    algo,
    const cudaCTCLossDescriptor_t ctcLossDesc,
    const cudaTensorDescriptor_t probsDesc,
    const void           *probs,
    const int            labels[],
    const int            labelLengths[],
    const int            inputLengths[],
    void                 *costs,
    const cudaTensorDescriptor_t gradientsDesc,
    const void           *gradients,
    size_t               *workspaceSizeInBytes,
    void                 *workspace)
    
```

Note: This function can have an inconsistent interface depending on the `cudaLossNormalizationMode_t` chosen (bound to the `cudaCTCLossDescriptor_t` with `cudaSetCTCLossDescriptorEx()`). For the `CUDNN_LOSS_NORMALIZATION_NONE`, this function has an inconsistent interface, for example, the probs input is probability normalized by softmax, but the gradients output is with respect to the unnormalized activation. However, for `CUDNN_LOSS_NORMALIZATION_SOFTMAX`, the function has a consistent interface; all values are normalized by softmax.

### Parameters

**handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to `cudaHandle_t`.

**algo**

*Input.* Enumerant that specifies the chosen CTC loss algorithm. For more information, refer to `cudaCTCLossAlgo_t`.

**ctcLossDesc**

*Input.* Handle to the previously initialized CTC loss descriptor. To use this `_v8` function, this descriptor must be set using `cudaSetCTCLossDescriptor_v8()`. For more information, refer to `cudaCTCLossDescriptor_t`.

**probsDesc**

*Input.* Handle to the previously initialized probabilities tensor descriptor. For more information, refer to `cudaTensorDescriptor_t`.

**probs**

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

**labels**

*Input.* Pointer to a previously initialized labels list, in GPU memory.

**labelLengths**

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

**inputLengths**

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

**costs**

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

**gradientsDesc**

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

**gradients**

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

**workspace**

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

**sizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

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

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

## 8.2.5. cudnnDestroyCTCLossDescriptor()

This function destroys a CTC loss function descriptor object.

```
cudaStatus_t cudnnDestroyCTCLossDescriptor(
    cudnnCTCLossDescriptor_t ctcLossDesc)
```

### Parameters

#### ctcLossDesc

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

### Returns

#### CUDNN\_STATUS\_SUCCESS

The function returned successfully.

## 8.2.6. cudnnFindRNNBackwardDataAlgorithmEx()

This function has been deprecated in cuDNN 8.0.

This function attempts all available cuDNN algorithms for [cudnnRNNBackwardData\(\)](#), using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of `cudnnAlgorithmPerformance_t`. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

```
cudaStatus_t cudnnFindRNNBackwardDataAlgorithmEx(
    cudaHandle_t          handle,
    const cudnnRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudnnTensorDescriptor_t *yDesc,
    const void            *y,
    const cudnnTensorDescriptor_t *dyDesc,
    const void            *dy,
    const cudnnTensorDescriptor_t dhyDesc,
    const void            *dhy,
    const cudnnTensorDescriptor_t dcyDesc,
    const void            *dcy,
    const cudnnFilterDescriptor_t wDesc,
    const void            *w,
    const cudnnTensorDescriptor_t hxDesc,
    const void            *hx,
    const cudnnTensorDescriptor_t cxDesc,
    const void            *cx,
    const cudnnTensorDescriptor_t dxDesc,
    void                  *dx,
    const cudnnTensorDescriptor_t dhxDesc,
    void                  *dhx,
    const cudnnTensorDescriptor_t dcxDesc,
    void                  *dcx,
    const float           findIntensity,
    const int             requestedAlgoCount,
    int                   *returnedAlgoCount,
    cudnnAlgorithmPerformance_t *perfResults,
    void                  *workspace,
    size_t                workspaceSizeInBytes,
    const void            *reserveSpace,
    size_t                reserveSpaceSizeInBytes)
```

## 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 the `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). The second dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:



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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument.

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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `dxDesc`. The third dimension must match the `hiddenSize` argument 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 cuDNN 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- ▶ Setting `findIntensity` within the range  $(0, 1.]$  will set a percentage of the entire RNN search space to search. When `findIntensity` is set to 1.0, a full search is performed over all RNN parameters.
- ▶ When `findIntensity` is set to `0.0f`, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- ▶ Setting `findIntensity` within the range  $[-1., 0)$  sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been

heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.

- ▶ Values outside the range [-1,1] are truncated to the range [-1,1], and then interpreted as per the above.
- ▶ Setting `findIntensity` to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- ▶ This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

**returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

**perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**reserveSpace**

*Input/Output.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `dhxDesc`, `wDesc`, `hxDesc`, `cxDesc`, `dcxDesc`, `dhyDesc`, or `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`, or `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.

## 8.2.7. **cudaFindRNNBackwardWeightsAlgorithmEx()**

This function has been deprecated in cuDNN 8.0.

This function attempts all available cuDNN algorithms for [cudaRNNBackwardWeights\(\)](#), using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of `cudaAlgorithmPerformance_t`. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

```

cudaStatus_t cudaFindRNNBackwardWeightsAlgorithmEx(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_t *xDesc,
    const void            *x,
    const cudaTensorDescriptor_t hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t yDesc,
    const void            *y,
    const float           findIntensity,
    const int             requestedAlgoCount,
    int                   *returnedAlgoCount,
    cudaAlgorithmPerformance_t *perfResults,
    const void            *workspace,
    size_t                workspaceSizeInBytes,
    const cudaFilterDescriptor_t dwDesc,
    void                  *dw,
    const void            *reserveSpace,
    size_t                reserveSpaceSizeInBytes)

```

### 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 the [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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

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 cuDNN 7.2.0. It is used in cuDNN 7.2.0 and later versions to control the overall runtime of the RNN find algorithms, by selecting the percentage of a large Cartesian product space to be searched.

- ▶ Setting `findIntensity` within the range (0,1.] will set a percentage of the entire RNN search space to search. When `findIntensity` is set to 1.0, a full search is performed over all RNN parameters.

- ▶ When `findIntensity` is set to 0.0f, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- ▶ Setting `findIntensity` within the range  $[-1.,0)$  sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- ▶ Values outside the range  $[-1,1]$  are truncated to the range  $[-1,1]$ , and then interpreted as per the above.
- ▶ Setting `findIntensity` to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.
- ▶ This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

**returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

**perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**dwDesc**

*Input.* Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**dw**

*Input/Output.* Data pointer to GPU memory associated with the filter descriptor `dwDesc`.

**reserveSpace**

*Input.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

## Returns

### CUDNN\_STATUS\_SUCCESS

The function launched successfully.

### CUDNN\_STATUS\_NOT\_SUPPORTED

The function does not support the provided configuration.

### CUDNN\_STATUS\_BAD\_PARAM

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `dwDesc` or one of the descriptors in `xDesc`, `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `yDesc`, or `dwDesc` 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.

## 8.2.8. `cudaFindRNNTForwardTrainingAlgorithmEx()`

This function has been deprecated in cuDNN 8.0.

This function attempts all available cuDNN algorithms for `cudaRNNTForwardTraining()`, using user-allocated GPU memory. It outputs the parameters that influence the performance of the algorithm to a user-allocated array of `cudaAlgorithmPerformance_t`. These parameter metrics are written in sorted fashion where the first element has the lowest compute time.

```

cudaStatus_t cudaFindRNNTForwardTrainingAlgorithmEx(
    cudaHandle_t          handle,
    const cudaRNNTDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_t *xDesc,
    const void            *x,
    const cudaTensorDescriptor_t  hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t  cxDesc,
    const void            *cx,
    const cudaFilterDescriptor_t  wDesc,
    const void            *w,
    const cudaTensorDescriptor_t  yDesc,
    void                  *y,
    const cudaTensorDescriptor_t  hyDesc,
    void                  *hy,
    const cudaTensorDescriptor_t  cyDesc,
    void                  *cy,
    const float           findIntensity,
    const int             requestedAlgoCount,
    int                   *returnedAlgoCount,
    cudaAlgorithmPerformance_t *perfResults,

```

```
void          *workspace,
size_t       workSpaceSizeInBytes,
void         *reserveSpace,
size_t       reserveSpaceSizeInBytes)
```

## 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 the `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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the first dimension should match the `numLayers` argument.



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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 cuDNN 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 0.0, a quick, minimal search is performed. This setting has the best runtime. However, in this case the parameters returned by this function will not correspond to the best performance of the algorithm; a longer search might discover better parameters. This option will execute up to three instances of the configured RNN problem. Runtime will vary proportionally to RNN problem size, as it will in the other cases, hence no guarantee of an explicit time bound can be given.
- ▶ Setting `findIntensity` within the range  $[-1., 0)$  sets a percentage of a reduced Cartesian product space to be searched. This reduced search space has been heuristically selected to have good performance. The setting of -1.0 represents a full search over this reduced search space.
- ▶ Values outside the range  $[-1, 1]$  are truncated to the range  $[-1, 1]$ , and then interpreted as per the above.
- ▶ Setting `findIntensity` to 1.0 in cuDNN 7.2 and later versions is equivalent to the behavior of this function in versions prior to cuDNN 7.2.0.

- ▶ This function times the single RNN executions over large parameter spaces - one execution per parameter combination. The times returned by this function are latencies.

**requestedAlgoCount**

*Input.* The maximum number of elements to be stored in `perfResults`.

**returnedAlgoCount**

*Output.* The number of output elements stored in `perfResults`.

**perfResults**

*Output.* A user-allocated array to store performance metrics sorted ascending by compute time.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**reserveSpace**

*Input/Output.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `cxDesc`, `wDesc`, `hyDesc`, or `cyDesc` or one of the descriptors in `xDesc`, `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `cxDesc`, `wDesc`, `yDesc`, `hyDesc`, or `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.

## 8.2.9. cudnnGetCTCLossDescriptor()

This function returns the configuration of the passed CTC loss function descriptor.

```

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

## 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 `ctcLossDesc` descriptor passed is invalid.

## 8.2.10. cudnnGetCTCLossDescriptorEx()

This function returns the configuration of the passed CTC loss function descriptor.

```

cudnnStatus_t cudnnGetCTCLossDescriptorEx(
    cudnnCTCLossDescriptor_t      ctcLossDesc,
    cudnnDataType_t               *compType,
    cudnnLossNormalizationMode_t *normMode,
    cudnnNanPropagation_t         *gradMode)
    
```

## 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.

**normMode**

*Output.* Input normalization type for this CTC loss function descriptor. For more information, see [cudnnLossNormalizationMode\\_t](#).

**gradMode**

*Output.* NaN propagation type for this CTC loss function descriptor.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function returned successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

Input `ctcLossDesc` descriptor passed is invalid.

## 8.2.11. cudnnGetCTCLossDescriptor\_v8()

This function returns the configuration of the passed CTC loss function descriptor.

```

cudnnStatus_t cudnnGetCTCLossDescriptor_v8(
    cudnnCTCLossDescriptor_t      ctcLossDesc,
    cudnnDataType_t               *compType,
    cudnnLossNormalizationMode_t *normMode,
    )
    
```

```

    cudnnNanPropagation_t      *gradMode,
    int                       *maxLabelLength)

```

## 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.

### normMode

*Output.* Input normalization type for this CTC loss function descriptor. For more information, see [cudnnLossNormalizationMode\\_t](#).

### gradMode

*Output.* NaN propagation type for this CTC loss function descriptor.

### maxLabelLength

*Output.* The max label length for this CTC loss function descriptor.

## Returns

### CUDNN\_STATUS\_SUCCESS

The function returned successfully.

### CUDNN\_STATUS\_BAD\_PARAM

Input `ctcLossDesc` descriptor passed is invalid.

## 8.2.12. cudnnGetCTCLossWorkspaceSize()

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\(\)](#).

```

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)

```

## 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 gradient tensor descriptor.

### labels

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

**labelLengths**

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

**inputLengths**

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

**algo**

*Input.* Enumerant that specifies the chosen CTC loss algorithm.

**ctcLossDesc**

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

**sizeInBytes**

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

Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

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

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

### 8.2.13. `cudaAdvTrainGetCTCLossWorkspaceSize_v8()`

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call `cudaAdvTrainCTCLoss_v8()` with the specified algorithm. The workspace allocated will then be passed to the routine `cudaAdvTrainCTCLoss_v8()`.

```

cudaAdvTrainStatus_t cudaAdvTrainGetCTCLossWorkspaceSize_v8(
    cudaAdvTrainHandle_t      handle,
    cudaAdvTrainCTCLossAlgo_t  algo,
    const cudaAdvTrainCTCLossDescriptor_t  ctcLossDesc,
    const cudaAdvTrainTensorDescriptor_t   probsDesc,
    const cudaAdvTrainTensorDescriptor_t   gradientsDesc,
    size_t                      *sizeInBytes
)
    
```

Parameters

**handle**

*Input.* Handle to a previously created cuDNN context.

**algo**

*Input.* Enumerant that specifies the chosen CTC loss algorithm.

**ctcLossDesc**

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

**probsDesc**

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

**gradientsDesc**

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

**sizeInBytes**

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

## Returns

**CUDNN\_STATUS\_SUCCESS**

The query was successful.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The dimensions of `probsDesc` do not match the dimensions of `gradientsDesc`.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

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

## 8.2.14. `cudaGetRNNBackwardDataAlgorithmMaxCount()`

This function has been deprecated in cuDNN 8.0.

## 8.2.15. `cudaGetRNNForwardTrainingAlgorithmMaxCount()`

This function has been deprecated in cuDNN 8.0.

## 8.2.16. `cudaMultiHeadAttnBackwardData()`

This function computes exact, first-order derivatives of the multi-head attention block with respect to its inputs: Q, K, V. If  $y=F(x)$  is a vector-valued function that represents the multi-head attention layer and it takes some vector  $w \in R^n$  as an input (with all other parameters and inputs constant), and outputs vector  $y \in R^m$ , then `cudaMultiHeadAttnBackwardData()` computes the result of  $(\partial y_i / \partial x_j)^T \delta_{out}$  where  $\delta_{out}$  is the  $m \times 1$  gradient of the loss function with respect to multi-head attention outputs. The  $\delta_{out}$  gradient is back propagated through prior layers of the deep learning model.  $\partial y_i / \partial x_j$  is the  $m \times n$  Jacobian matrix of  $F(x)$ . The input is supplied via the `dout` argument and gradient results for Q, K, V are written to the `dqueries`, `dkeys`, and `dvalues` buffers.

```

cudaStatus_t cudaMultiHeadAttnBackwardData(
    cudaHandle_t handle,
    const cudaAttnDescriptor_t attnDesc,
    const int loWinIdx[],
    const int hiWinIdx[],
    const int devSeqLengthsDQDO[],

```

```

const int devSeqLengthsDKDV[],
const cudnnSeqDataDescriptor_t doDesc,
const void *dout,
const cudnnSeqDataDescriptor_t dqDesc,
void *dqueries,
const void *queries,
const cudnnSeqDataDescriptor_t dkDesc,
void *dkeys,
const void *keys,
const cudnnSeqDataDescriptor_t dvDesc,
void *dvalues,
const void *values,
size_t weightSizeInBytes,
const void *weights,
size_t workSpaceSizeInBytes,
void *workSpace,
size_t reserveSpaceSizeInBytes,
void *reserveSpace);

```

The `cudnnMultiHeadAttnBackwardData()` function does not output partial derivatives for residual connections because this result is equal to  $\delta_{out}$ . If the multi-head attention model enables residual connections sourced directly from Q, then the `dout` tensor needs to be added to `dqueries` to obtain the correct result of the latter. This operation is demonstrated in the `cuDNN multiHeadAttention` sample code.

The `cudnnMultiHeadAttnBackwardData()` function must be invoked after `cudnnMultiHeadAttnForward()`. The `loWinIdx[]`, `hiWinIdx[]`, `queries`, `keys`, `values`, `weights`, and `reserveSpace` arguments should be the same as in the `cudnnMultiHeadAttnForward()` call. `devSeqLengthsDQDO[]` and `devSeqLengthsDKDV[]` device arrays should contain the same start and end attention window indices as `devSeqLengthsQO[]` and `devSeqLengthsKV[]` arrays in the forward function invocation.



**Note:** `cudnnMultiHeadAttnBackwardData()` does not verify that sequence lengths stored in `devSeqLengthsDQDO[]` and `devSeqLengthsDKDV[]` contain the same settings as `seqLengthArray[]` in the corresponding sequence data descriptor.

## Parameters

### **handle**

*Input.* The current cuDNN context handle.

### **attnDesc**

*Input.* A previously initialized attention descriptor.

### **loWinIdx[], hiWinIdx[]**

*Input.* Two host integer arrays specifying the start and end indices of the attention window for each Q time-step. The start index in K, V sets is inclusive, and the end index is exclusive.

### **devSeqLengthsDQDO[]**

*Input.* Device array containing a copy of the sequence length array from the `dqDesc` or `doDesc` sequence data descriptor.

### **devSeqLengthsDKDV[]**

*Input.* Device array containing a copy of the sequence length array from the `dkDesc` or `dvDesc` sequence data descriptor.



**doDesc**

*Input.* Descriptor for the  $\delta_{out}$  gradients (vectors of partial derivatives of the loss function with respect to the multi-head attention outputs).

**dout**

Pointer to  $\delta_{out}$  gradient data in the device memory.

**dqDesc**

*Input.* Descriptor for `queries` and `dqueries` sequence data.

**dqueries**

*Output.* Device pointer to gradients of the loss function computed with respect to `queries` vectors.

**queries**

*Input.* Pointer to `queries` data in the device memory. This is the same input as in [cudaMultiHeadAttnForward\(\)](#).

**dkDesc**

*Input.* Descriptor for `keys` and `dkeys` sequence data.

**dkeys**

*Output.* Device pointer to gradients of the loss function computed with respect to `keys` vectors.

**keys**

*Input.* Pointer to `keys` data in the device memory. This is the same input as in [cudaMultiHeadAttnForward\(\)](#).

**dvDesc**

*Input.* Descriptor for `values` and `dvalues` sequence data.

**dvalues**

*Output.* Device pointer to gradients of the loss function computed with respect to `values` vectors.

**values**

*Input.* Pointer to `values` data in the device memory. This is the same input as in [cudaMultiHeadAttnForward\(\)](#).

**weightSizeInBytes**

*Input.* Size of the `weight` buffer in bytes where all multi-head attention trainable parameters are stored.

**weights**

*Input.* Address of the `weight` buffer in the device memory.

**workSpaceSizeInBytes**

*Input.* Size of the work-space buffer in bytes used for temporary API storage.

**workSpace**

*Input/Output.* Address of the work-space buffer in the device memory.

**reserveSpaceSizeInBytes**

*Input.* Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls.

**reserveSpace**

*Input/Output.* Address to the reserve-space buffer in the device memory.

## Returns

### CUDNN\_STATUS\_SUCCESS

No errors were detected while processing API input arguments and launching GPU kernels.

### CUDNN\_STATUS\_BAD\_PARAM

An invalid or incompatible input argument was encountered.

### CUDNN\_STATUS\_EXECUTION\_FAILED

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

### CUDNN\_STATUS\_INTERNAL\_ERROR

An inconsistent internal state was encountered.

### CUDNN\_STATUS\_NOT\_SUPPORTED

A requested option or a combination of input arguments is not supported.

### CUDNN\_STATUS\_ALLOC\_FAILED

Insufficient amount of shared memory to launch a GPU kernel.

## 8.2.17. cudnnMultiHeadAttnBackwardWeights ()

This function computes exact, first-order derivatives of the multi-head attention block with respect to its trainable parameters: projection weights and projection biases. If  $y=F(w)$  is a vector-valued function that represents the multi-head attention layer and it takes some vector  $x \in R^n$  of flatten weights or biases as an input (with all other parameters and inputs fixed), and outputs vector  $y \in R^m$ , then

`cudnnMultiHeadAttnBackwardWeights ()` computes the result of  $(\partial y_i / \partial x_j)^T \delta_{out}$  where  $\delta_{out}$  is the  $m \times 1$  gradient of the loss function with respect to multi-head attention outputs. The  $\delta_{out}$  gradient is back propagated through prior layers of the deep learning model.  $\partial y_i / \partial x_j$  is the  $m \times n$  Jacobian matrix of  $F(w)$ . The  $\delta_{out}$  input is supplied via the `dout` argument.

```

cudnnStatus_t cudnnMultiHeadAttnBackwardWeights(
    cudnnHandle_t handle,
    const cudnnAttnDescriptor_t attnDesc,
    cudnnWgradMode_t addGrad,
    const cudnnSeqDataDescriptor_t qDesc,
    const void *queries,
    const cudnnSeqDataDescriptor_t kDesc,
    const void *keys,
    const cudnnSeqDataDescriptor_t vDesc,
    const void *values,
    const cudnnSeqDataDescriptor_t doDesc,
    const void *dout,
    size_t weightSizeInBytes,
    const void *weights,
    void *dweights,
    size_t workSpaceSizeInBytes,
    void *workSpace,
    size_t reserveSpaceSizeInBytes,
    void *reserveSpace);

```

All gradient results with respect to weights and biases are written to the `dweights` buffer. The size and the organization of the `dweights` buffer is the same as the

`weights` buffer that holds multi-head attention weights and biases. The cuDNN `multiHeadAttention` sample code demonstrates how to access those weights.

Gradient of the loss function with respect to weights or biases is typically computed over multiple batches. In such a case, partial results computed for each batch should be summed together. The `addGrad` argument specifies if the gradients from the current batch should be added to previously computed results or the `dweights` buffer should be overwritten with the new results.

The `cudaMultiHeadAttnBackwardWeights()` function should be invoked after `cudaMultiHeadAttnBackwardData()`. The `queries`, `keys`, `values`, `weights`, and `reserveSpace` arguments should be the same as in `cudaMultiHeadAttnForward()` and `cudaMultiHeadAttnBackwardData()` calls. The `dout` argument should be the same as in `cudaMultiHeadAttnBackwardData()`.

## Parameters

**handle**

*Input.* The current cuDNN context handle.

**attnDesc**

*Input.* A previously initialized attention descriptor.

**addGrad**

*Input.* Weight gradient output mode.

**qDesc**

*Input.* Descriptor for the `query` sequence data.

**queries**

*Input.* Pointer to `queries` sequence data in the device memory.

**kDesc**

*Input.* Descriptor for the `keys` sequence data.

**keys**

*Input.* Pointer to `keys` sequence data in the device memory.

**vDesc**

*Input.* Descriptor for the `values` sequence data.

**values**

*Input.* Pointer to `values` sequence data in the device memory.

**doDesc**

*Input.* Descriptor for the  $\delta_{out}$  gradients (vectors of partial derivatives of the loss function with respect to the multi-head attention outputs).

**dout**

*Input.* Pointer to  $\delta_{out}$  gradient data in the device memory.

**weightSizeInBytes**

*Input.* Size of the `weights` and `dweights` buffers in bytes.

**weights**

*Input.* Address of the `weight` buffer in the device memory.

**dweights**

*Output.* Address of the weight gradient buffer in the device memory.

**workspaceSizeInBytes**

*Input.* Size of the work-space buffer in bytes used for temporary API storage.

**workspace**

*Input/Output.* Address of the work-space buffer in the device memory.

**reserveSpaceSizeInBytes**

*Input.* Size of the reserve-space buffer in bytes used for data exchange between forward and backward (gradient) API calls.

**reserveSpace**

*Input/Output.* Address to the reserve-space buffer in the device memory.

Returns

**CUDNN\_STATUS\_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

An inconsistent internal state was encountered.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

A requested option or a combination of input arguments is not supported.

8.2.18. **cudaRNNBackwardData()**

This function has been deprecated in cuDNN 8.0. Use [cudaRNNBackwardData\\_v8\(\)](#) instead of `cudaRNNBackwardData()`.

```

cudaStatus_t cudaRNNBackwardData(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_t *yDesc,
    const void            *y,
    const cudaTensorDescriptor_t *dyDesc,
    const void            *dy,
    const cudaTensorDescriptor_t dhyDesc,
    const void            *dhy,
    const cudaTensorDescriptor_t dcyDesc,
    const void            *dcy,
    const cudaFilterDescriptor_t wDesc,
    const void            *w,
    const cudaTensorDescriptor_t hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t cxDesc,
    const void            *cx,
    const cudaTensorDescriptor_t *dxDesc,
    void                  *dx,
    const cudaTensorDescriptor_t dhxDesc,
    void                  *dhx,
    const cudaTensorDescriptor_t dcxDesc,
    void                  *dcx,

```

```

void                                     *workspace,
size_t                                  workSpaceSizeInBytes,
const void                               *reserveSpace,
size_t                                   reserveSpaceSizeInBytes)

```

This routine executes the recurrent neural network described by `rnnDesc` with output gradients `dy`, `dhy`, and `dhc`, weights `w` and input gradients `dx`, `dhx`, and `dcx`. `workspace` is required for intermediate storage. The data in `reserveSpace` must have previously been generated by `cudnnRNNForwardTraining()`. The same `reserveSpace` data must be used for future calls to `cudnnRNNBackwardWeights()` if they execute on the same input data.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context. For more information, refer to [cudnnHandle\\_t](#).

### **rnnDesc**

*Input.* A previously initialized RNN descriptor. For more information, refer to [cudnnRNNDescriptor\\_t](#).

### **seqLength**

*Input.* Number of iterations to unroll over. The value of this `seqLength` must not exceed the value that was used in the `cudnnGetRNNWorkspaceSize()` function for querying the workspace size required to execute the RNN.

### **yDesc**

*Input.* An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). For more information, refer to [cudnnTensorDescriptor\\_t](#). The second dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`:

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

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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the second dimension should match the `hiddenSize` argument.

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

**dcy**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `dcyDesc`. If a `NULL` pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

**wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN. For more information, refer to [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 used to initialize `rnnDesc`:

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

The second dimension must match the second dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the second dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

**dcx**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `dcxDesc`. If a `NULL` pointer is passed, the gradient at the cell input of the network will not be set.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**reserveSpace**

*Input/Output.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.



## Returns

### **CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `dhxDesc`, `wDesc`, `hxDesc`, `cxDesc`, `dcxDesc`, `dhyDesc`, or `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`, or `dcyDesc` has incorrect strides or dimensions.
- ▶ `workSpaceSizeInBytes` is too small.
- ▶ `reserveSpaceSizeInBytes` is too small.

### **CUDNN\_STATUS\_INVALID\_VALUE**

[`cudaSetPersistentRNNPlan\(\)`](#) was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

### **CUDNN\_STATUS\_MAPPING\_ERROR**

A GPU/CUDA resource, such as a texture object, shared memory, or zero-copy memory is not available in the required size or there is a mismatch between the user resource and cuDNN internal resources. A resource mismatch may occur, for example, when calling `cudaSetStream()`. There could be a mismatch between the user provided CUDA stream and the internal CUDA events instantiated in the cuDNN handle when `cudaCreate()` was invoked.

This error status may not be correctable when it is related to texture dimensions, shared memory size, or zero-copy memory availability. If `CUDNN_STATUS_MAPPING_ERROR` is returned by `cudaSetStream()`, then it is typically correctable, however, it means that the cuDNN handle was created on one GPU and the user stream passed to this function is associated with another GPU.

### **CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

### **CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 8.2.19. `cudaRNNBackwardData_v8()`

This function computes exact, first-order derivatives of the RNN model with respect to its inputs:  $x$ ,  $h_x$  and for the LSTM cell type also  $c_x$ . If  $o = [y, h_y, c_y] = F(x, h_x, c_x) = F(z)$  is a vector-valued function that represents the entire RNN model and it takes vectors  $x$  (for all time-steps) and vectors  $h_x, c_x$  (for all layers) as inputs, concatenated into  $z \in R^n$

(network weights and biases are assumed constant), and outputs vectors  $y, h_y, c_y$  concatenated into a vector  $o \in R^m$ , then `cudaRNNBackwardData_v8()` computes the result of  $(\partial o_i / \partial z_j)^T \delta_{out}$  where  $\delta_{out}$  is the  $m \times 1$  gradient of the loss function with respect to all RNN outputs. The  $\delta_{out}$  gradient is back propagated through prior layers of the deep learning model, starting from the model output.  $\partial o_i / \partial z_j$  is the  $m \times n$  Jacobian matrix of  $F(z)$ . The  $\delta_{out}$  input is supplied via the `dy, dh_y,` and `dc_y` arguments and gradient results  $(\partial o_i / \partial z_j)^T \delta_{out}$  are written to the `dx, dh_x,` and `dc_x` buffers.

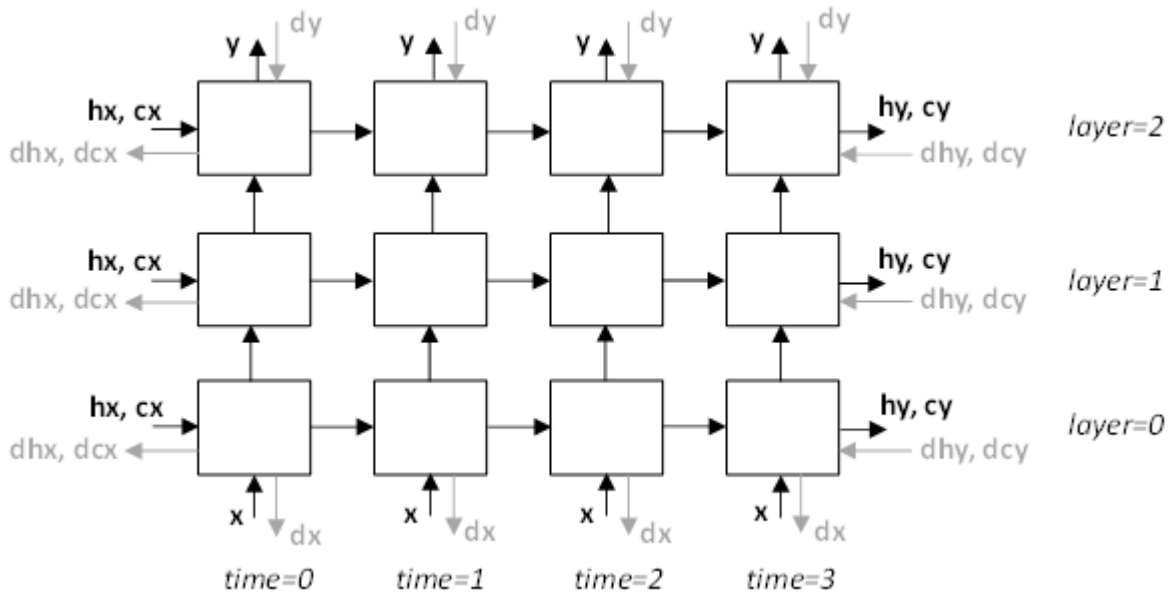
```

cudaStatus_t cudaRNNBackwardData_v8(
    cudaHandle_t handle,
    cudaRNNDescriptor_t rnnDesc,
    const int32_t devSeqLengths[],
    cudaRNNDataDescriptor_t yDesc,
    const void *y,
    const void *dy,
    cudaRNNDataDescriptor_t xDesc,
    void *dx,
    cudaTensorDescriptor_t hDesc,
    const void *hx,
    const void *dh_y,
    void *dh_x,
    cudaTensorDescriptor_t cDesc,
    const void *cx,
    const void *dc_y,
    void *dc_x,
    size_t weightSpaceSize,
    const void *weightSpace,
    size_t workSpaceSize,
    void *workSpace,
    size_t reserveSpaceSize,
    void *reserveSpace);

```

Locations of `x, y, hx, cx, hy, cy, dx, dy, dh_x, dc_x, dh_y,` and `dc_y` signals a multi-layer RNN model are shown in the following figure. Note that internal RNN signals (between time-steps and between layers) are not exposed by the `cudaRNNBackwardData_v8()` function.

Figure 6. Locations of  $x$ ,  $y$ ,  $h_x$ ,  $c_x$ ,  $h_y$ ,  $c_y$ ,  $dx$ ,  $dy$ ,  $dh_x$ ,  $dc_x$ ,  $dh_y$ , and  $dc_y$  Signals a Multi-Layer RNN Model



Memory addresses to the primary RNN output  $y$ , the initial hidden state  $h_x$ , and the initial cell state  $c_x$  (for LSTM only) should point to the same data as in the preceding `cudaRNNForward()` call. The  $dy$  and  $dx$  pointers cannot be NULL.

The `cudaRNNBackwardData_v8()` function accepts any combination of  $dh_y$ ,  $dh_x$ ,  $dc_y$ ,  $dc_x$  buffer addresses being NULL. When  $dh_y$  or  $dc_y$  are NULL, it is assumed that those inputs are zero. When  $dh_x$  or  $dc_x$  pointers are NULL then the corresponding results are not written by `cudaRNNBackwardData_v8()`.

When all  $h_x$ ,  $dh_y$ ,  $dh_x$  pointers are NULL, then the corresponding tensor descriptor `hDesc` can be NULL too. The same rule applies to the  $c_x$ ,  $dc_y$ ,  $dc_x$  pointers and the `cDesc` tensor descriptor.

The `cudaRNNBackwardData_v8()` function allows the user to use padded layouts for inputs  $y$ ,  $dy$ , and output  $dx$ . In padded or unpacked layouts (`CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED`, `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED`) each sequence of vectors in a mini-batch has a fixed length defined by the `maxSeqLength` argument in the `cudaSetRNNDataDescriptor()` function. The term "unpacked" refers here to the presence of padding vectors, and not unused address ranges between contiguous vectors.

Each padded, fixed-length sequence starts from a segment of valid vectors. The valid vector count is stored in `seqLengthArray` passed to `cudaSetRNNDataDescriptor()`, such that  $0 < seqLengthArray[i] \leq maxSeqLength$  for all sequences in a mini-batch, that is, for  $i=0..batchSize-1$ . The remaining padding vectors make the combined sequence length equal to `maxSeqLength`. Both sequence-major and batch-major padded layouts are supported.

In addition, a packed sequence-major layout:

`CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_PACKED` can be selected by the user. In the latter layout, sequences of vectors in a mini-batch are sorted in the descending order according to the sequence lengths. First, all vectors for time step zero are stored. They are followed by vectors for time step one, and so on. This layout uses no padding vectors.

The same layout type must be specified in `xDesc` and `yDesc` descriptors.

Two host arrays named `seqLengthArray` in `xDesc` and `yDesc` RNN data descriptors must be the same. In addition, a copy of `seqLengthArray` in the device memory must be passed via the `devSeqLengths` argument. This array is supplied directly to GPU kernels. Starting in cuDNN 8.9.1, the `devSeqLengths` parameter is no longer required and can be set to `NULL`. The variable sequence length array is transferred automatically to GPU memory by the `cudaRNNBackwardData_v8()` function.

The `cudaRNNBackwardData_v8()` function does not verify that sequence lengths stored in `devSeqLengths` in GPU memory are the same as in `xDesc` and `yDesc` descriptors in CPU memory. Sequence length arrays from `xDesc` and `yDesc` descriptors are checked for consistency, however.

The `cudaRNNBackwardData_v8()` function must be called after [`cudaRNNForward\(\)`](#). The [`cudaRNNForward\(\)`](#) function should be invoked with the  `fwdMode`  argument of type [`cudaRNNForward\(\)`](#) set to `CUDNN_FWD_MODE_TRAINING`.

## Parameters

### **handle**

*Input.* The current cuDNN context handle.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **devSeqLengths**

*Input.* A copy of `seqLengthArray` from `xDesc` or `yDesc` RNN data descriptors. The `devSeqLengths` array must be stored in GPU memory as it is accessed asynchronously by GPU kernels, possibly after the `cudaRNNBackwardData_v8()` function exists. In cuDNN 8.9.1 and later versions, `devSeqLengths` should be `NULL`.

### **yDesc**

*Input.* A previously initialized descriptor corresponding to the RNN model primary output. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` need to match that of `xDesc`.

### **y, dy**

*Input.* Data pointers to GPU buffers holding the RNN model primary output and gradient deltas (gradient of the loss function with respect to `y`). The `y` output should be produced by the preceding [`cudaRNNForward\(\)`](#) call. The `y` and `dy` vectors are expected to be laid out in memory according to the layout specified by `yDesc`. The elements in the tensor (including elements in padding vectors) must be densely packed. The `y` and `dy` arguments cannot be `NULL`.

**xDesc**

*Input.* A previously initialized RNN data descriptor corresponding to the gradient of the loss function with respect to the RNN primary model input. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` must match that of `yDesc`. The parameter `vectorSize` must match the `inputSize` argument passed to the [`cudaSetRNNDesc\_v8\(\)`](#) function.

**dx**

*Output.* Data pointer to GPU memory where back-propagated gradients of the loss function with respect to the RNN primary input `x` should be stored. The vectors are expected to be arranged in memory according to the layout specified by `xDesc`. The elements in the tensor (including padding vectors) must be densely packed. This argument cannot be `NULL`.

**hDesc**

*Input.* A tensor descriptor describing the initial RNN hidden state `hx` and gradients of the loss function with respect to the initial of the final hidden state. Hidden state data and the corresponding gradients are fully packed. The first dimension of the tensor depends on the `dirMode` argument passed to the [`cudaSetRNNDesc\_v8\(\)`](#) function.

- ▶ If `dirMode` is `CUDNN_UNIDIRECTIONAL`, then the first dimension should match the `numLayers` argument passed to [`cudaSetRNNDesc\_v8\(\)`](#).
- ▶ If `dirMode` is `CUDNN_BIDIRECTIONAL`, then the first dimension should be double the `numLayers` argument passed to [`cudaSetRNNDesc\_v8\(\)`](#).

The second dimension must match the `batchSize` parameter described in `xDesc`. The third dimension depends on whether RNN mode is `CUDNN_LSTM` and whether the LSTM projection is enabled. Specifically:

- ▶ If RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the third dimension must match the `projSize` argument passed to the [`cudaSetRNNDesc\_v8\(\)`](#) call.
- ▶ Otherwise, the third dimension must match the `hiddenSize` argument passed to the [`cudaSetRNNDesc\_v8\(\)`](#) call used to initialize `rnnDesc`.

**hx, dhx**

*Input.* Addresses of GPU buffers with the RNN initial hidden state `hx` and gradient deltas `dhx`. Data dimensions are described by the `hDesc` tensor descriptor. If a `NULL` pointer is passed in `hx` or `dhx` arguments, the corresponding buffer is assumed to contain all zeros.

**dhx**

*Output.* Pointer to the GPU buffer where first-order derivatives corresponding to initial hidden state variables should be stored. Data dimensions are described by the `hDesc` tensor descriptor. If a `NULL` pointer is assigned to `dhx`, the back-propagated derivatives are not saved.

**cDesc**

*Input.* For LSTM networks only. A tensor descriptor describing the initial cell state `cx` and gradients of the loss function with respect to the initial of the final cell state. Cell state data are fully packed. The first dimension of the tensor depends on the `dirMode` argument passed to the `cudnnSetRNNDescriptor_v8()` call.

- ▶ If `dirMode` is `CUDNN_UNIDIRECTIONAL`, then the first dimension should match the `numLayers` argument passed to `cudnnSetRNNDescriptor_v8()`.
- ▶ If `dirMode` is `CUDNN_BIDIRECTIONAL`, then the first dimension should be double the `numLayers` argument passed to `cudnnSetRNNDescriptor_v8()`.

The second tensor dimension must match the `batchSize` parameter in `xDesc`. The third dimension must match the `hiddenSize` argument passed to the `cudnnSetRNNDescriptor_v8()` call.

**cx, dcy**

*Input.* For LSTM networks only. Addresses of GPU buffers with the initial LSTM state data and gradient deltas `dcy`. Data dimensions are described by the `cDesc` tensor descriptor. If a `NULL` pointer is passed in `cx` or `dcy` arguments, the corresponding buffer is assumed to contain all zeros.

**dcx**

*Output.* For LSTM networks only. Pointer to the GPU buffer where first-order derivatives corresponding to initial LSTM state variables should be stored. Data dimensions are described by the `cDesc` tensor descriptor. If a `NULL` pointer is assigned to `dcx`, the back-propagated derivatives are not saved.

**weightSpaceSize**

*Input.* Specifies the size in bytes of the provided weight-space buffer.

**weightSpace**

*Input.* Address of the weight space buffer in GPU memory.

**workSpaceSize**

*Input.* Specifies the size in bytes of the provided workspace buffer.

**workSpace**

*Input/Output.* Address of the workspace buffer in GPU memory to store temporary data.

**reserveSpaceSize**

*Input.* Specifies the size in bytes of the reserve-space buffer.

**reserveSpace**

*Input/Output.* Address of the reserve-space buffer in GPU memory.

**Returns****CUDNN\_STATUS\_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

At least one of the following conditions are met:

- ▶ variable sequence length input is passed while `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is specified
- ▶ `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is requested on pre-Pascal devices
- ▶ the 'double' floating point type is used for input/output and the `CUDNN_RNN_ALGO_PERSIST_STATIC` algo

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered. For example:

- ▶ some input descriptors are `NULL`
- ▶ settings in `rnnDesc`, `xDesc`, `yDesc`, `hDesc`, or `cDesc` descriptors are invalid
- ▶ `weightSpaceSize`, `workSpaceSize`, or `reserveSpaceSize` is too small

**CUDNN\_STATUS\_MAPPING\_ERROR**

A GPU/CUDA resource, such as a texture object, shared memory, or zero-copy memory is not available in the required size or there is a mismatch between the user resource and cuDNN internal resources. A resource mismatch may occur, for example, when calling `cudaSetStream()`. There could be a mismatch between the user provided CUDA stream and the internal CUDA events instantiated in the cuDNN handle when `cudaCreate()` was invoked.

This error status may not be correctable when it is related to texture dimensions, shared memory size, or zero-copy memory availability. If `CUDNN_STATUS_MAPPING_ERROR` is returned by `cudaSetStream()`, then it is typically correctable, however, it means that the cuDNN handle was created on one GPU and the user stream passed to this function is associated with another GPU.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate CPU memory.

## 8.2.20. `cudaRNNSBackwardDataEx()`

This function has been deprecated in cuDNN 8.0. Use [cudaRNNSBackwardData\\_v8](#) instead of `cudaRNNSBackwardDataEx()`.

```

cudaStatus_t cudaRNNSBackwardDataEx(
    cudaHandle_t          handle,
    const cudaRNNDescr_t  rnnDesc,
    const cudaRNNDDataDescr_t  yDesc,
    const void            *y,
    const cudaRNNDDataDescr_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 an 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 (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function [cudnnRNNBackwardData\(\)](#), the sequences in the mini-batch need to be sorted in descending order according to length.

## Parameters

### **handle**

*Input.* Handle to a previously created This function is deprecated starting in cuDNN 8.0.0. context.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **yDesc**

*Input.* A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

### **y**

*Input.* Data pointer to the GPU memory associated with the RNN data descriptor `yDesc`. The vectors are expected to be laid out in memory according to the layout specified by `yDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported. Must contain the exact same data previously produced by [cudnnRNNForwardTrainingEx\(\)](#).



**dyDesc**

*Input.* A previously initialized RNN data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, `vectorSize`, and `seqLengthArray` need to match the `yDesc` previously passed to [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 used to initialize `rnnDesc`. Additionally:

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

The second dimension must match the `batchSize` parameter in `xDesc`. The third dimension depends on whether the RNN mode is `CUDNN_LSTM` and whether LSTM projection is enabled. Additionally:

- ▶ If the RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the third dimension must match the `recProjSize` argument passed to [cudaSetRNNProjectionLayers\(\)](#) call used to set `rnnDesc`.
- ▶ Otherwise, the third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`. Additionally:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

**dcy**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `dcyDesc`. If a `NULL` pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

**wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN.

**w**

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

**hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

**hx**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `hxDesc`. If a `NULL` pointer is passed, the initial hidden state of the network will be initialized to zero. Must contain the exact same data previously passed into [cudnnRNNForwardTrainingEx\(\)](#), or be `NULL` if `NULL` was previously passed to [cudnnRNNForwardTrainingEx\(\)](#).

**cxDesc**

*Input.* A fully packed tensor descriptor describing the initial cell state for LSTM networks. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

**cx**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `cxDesc`. If a `NULL` pointer is passed, the initial cell state of the network will be initialized to zero. Must contain the exact same data previously passed into [cudnnRNNForwardTrainingEx\(\)](#), or be `NULL` if `NULL` was previously passed to [cudnnRNNForwardTrainingEx\(\)](#).

**dxDesc**

*Input.* A previously initialized RNN data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, `vectorSize` and `seqLengthArray` need to match that of `xDesc` previously passed to [cudnnRNNForwardTrainingEx\(\)](#).

**dx**

*Output.* Data pointer to the GPU memory associated with the RNN data descriptor `dxDesc`. The vectors are expected to be laid out in memory according to the layout specified by `dxDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**dhxDesc**

*Input.* A fully packed tensor descriptor describing the gradient at the initial hidden state of the RNN. The descriptor must be set exactly the same way as `dhYDesc`.

**dhx**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `dhxDesc`. If a `NULL` pointer is passed, the gradient at the hidden input of the network will not be set.

**dcxDesc**

*Input.* A fully packed tensor descriptor describing the gradient at the initial cell state of the RNN. The descriptor must be set exactly the same way as `dcyDesc`.

**dcx**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `dcxDesc`. If a `NULL` pointer is passed, the gradient at the cell input of the network will not be set.

**dkDesc**

Reserved. Users may pass in `NULL`.

**dkeys**

Reserved. Users 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`, or `dcyDesc` is invalid or has incorrect strides or dimensions.
- ▶ `workSpaceSizeInBytes` is too small.
- ▶ `reserveSpaceSizeInBytes` is too small.

**CUDNN\_STATUS\_INVALID\_VALUE**

`cudaSetPersistentRNNPlan()` was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

**CUDNN\_STATUS\_MAPPING\_ERROR**

A GPU/CUDA resource, such as a texture object, shared memory, or zero-copy memory is not available in the required size or there is a mismatch between the user resource and cuDNN internal resources. A resource mismatch may occur, for example, when calling `cudaSetStream()`. There could be a mismatch between the user provided CUDA stream and the internal CUDA events instantiated in the cuDNN handle when `cudaCreate()` was invoked.

This error status may not be correctable when it is related to texture dimensions, shared memory size, or zero-copy memory availability. If `CUDNN_STATUS_MAPPING_ERROR` is returned by `cudaSetStream()`, then it is typically correctable, however, it means that the cuDNN handle was created on one GPU and the user stream passed to this function is associated with another GPU.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 8.2.21. `cudaRNNBackwardWeights()`

This function has been deprecated in cuDNN 8.0. Use `cudaRNNBackwardWeights_v8()` instead of `cudaRNNBackwardWeights()`.

```

cudaStatus_t cudaRNNBackwardWeights(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t  rnnDesc,
    const int             seqLength,
    const cudaTensorDescriptor_t *xDesc,
    const void            *x,
    const cudaTensorDescriptor_t  hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t  yDesc,
    const void            *y,
    const void            *workspace,
    size_t                workspaceSizeInBytes,
    const cudaFilterDescriptor_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 [`cudaRNNBackwardData\(\)`](#).

## 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 the [`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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the second dimension should match the `hiddenSize` argument.

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

The first dimension of the tensor `n` must match the first dimension of the tensor `n` in `dyDesc`.

**y**

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

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**dwDesc**

*Input.* Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**dw**

*Input/Output.* Data pointer to GPU memory associated with the filter descriptor `dwDesc`.

**reserveSpace**

*Input.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `dwDesc` or one of the descriptors in `xDesc`, `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `yDesc`, `dwDesc` 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.

## 8.2.22. cudnnRNNBackwardWeights\_v8()

This function computes exact, first-order derivatives of the RNN model with respect to all trainable parameters: weights and biases. If  $o = [y, h_y, c_y] = F(w)$  is a vector-valued function that represents the multi-layer RNN model and it takes some vector  $w \in R^n$  of "flatten" weights or biases as input (with all other data inputs constant), and outputs vector  $o \in R^m$ , then `cudnnRNNBackwardWeights_v8()` computes the result of  $(\partial o_i / \partial w_j)^T \delta_{out}$  where  $\delta_{out}$  is the  $m \times 1$  gradient of the loss function with respect to all RNN outputs. The  $\delta_{out}$  gradient is back propagated through prior layers of the deep learning model, starting from the model output.  $\partial o_i / \partial w_j$  is the  $m \times n$  Jacobian matrix of  $F(w)$ . The  $\delta_{out}$  input is supplied via the `dy`, `dhy`, and `dcy` arguments in the `cudnnRNNBackwardData_v8()` function.

```
cudnnStatus_t cudnnRNNBackwardWeights_v8(
    cudnnHandle_t handle,
    cudnnRNNDescriptor_t rnnDesc,
    cudnnWgradMode_t addGrad,
    const int32_t devSeqLengths[],
    cudnnRNNDataDescriptor_t xDesc,
    const void *x,
    cudnnTensorDescriptor_t hDesc,
    const void *hx,
    cudnnRNNDataDescriptor_t yDesc,
    const void *y,
    size_t weightSpaceSize,
    void *dweightSpace,
    size_t workSpaceSize,
    void *workSpace,
    size_t reserveSpaceSize,
    void *reserveSpace);
```

All gradient results  $(\partial o_i / \partial w_j)^T \delta_{out}$  with respect to weights and biases are written to the `dweightSpace` buffer. The size and the organization of the `dweightSpace` buffer is the same as the `weightSpace` buffer that holds RNN weights and biases.

Gradient of the loss function with respect to weights and biases is typically computed over multiple mini-batches. In such a case, partial results computed for each mini-batch should be aggregated. The `addGrad` argument specifies if gradients from the current mini-batch should be added to previously computed results (`CUDNN_WGRAD_MODE_ADD`) or the `dweightSpace` buffer should be overwritten with the new results (`CUDNN_WGRAD_MODE_SET`). Currently, the `cudnnRNNBackwardWeights_v8()` function supports the `CUDNN_WGRAD_MODE_ADD` mode only so the `dweightSpace` buffer should be zeroed by the user before invoking the routine for the first time.

The same sequence lengths must be specified in the `xDesc` descriptor and in the device array `devSeqLengths`. Starting in cuDNN 8.9.1, the `devSeqLengths` parameter is no longer required and can be set to `NULL`. The variable sequence length array is transferred automatically to GPU memory by the `cudnnRNNBackwardWeights_v8()` function.

The `cudnnRNNBackwardWeights_v8()` function should be invoked after [cudnnRNNBackwardData\(\)](#).

## Parameters

### **handle**

*Input.* The current cuDNN context handle.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **addGrad**

*Input.* Weight gradient output mode. For more details, see the description of the [cudnnWgradMode\\_t](#) enumerated type. Currently, only the `CUDNN_WGRAD_MODE_ADD` mode is supported by the `cudnnRNNBackwardWeights_v8()` function.

### **devSeqLengths**

*Input.* A copy of `seqLengthArray` from the `xDesc` RNN data descriptor. The `devSeqLengths` array must be stored in GPU memory as it is accessed asynchronously by GPU kernels, possibly after the `cudnnRNNBackwardWeights_v8()` function exists. In cuDNN 8.9.1 and later versions, `devSeqLengths` should be `NULL`.

### **xDesc**

*Input.* A previously initialized descriptor corresponding to the RNN model input data. This is the same RNN data descriptor as used in the preceding [cudnnRNNForward\(\)](#) and [cudnnRNNBackwardData\\_v8\(\)](#) calls.

### **x**

*Input.* Pointer to the GPU buffer with the primary RNN input. The same buffer address `x` should be provided in prior [cudnnRNNForward\(\)](#) and [cudnnRNNBackwardData\\_v8\(\)](#) calls.

### **hDesc**

*Input.* A tensor descriptor describing the initial RNN hidden state. Hidden state data are fully packed. This is the same tensor descriptor as used in prior [cudnnRNNForward\(\)](#) and [cudnnRNNBackwardData\\_v8\(\)](#) calls.

### **hx**

*Input.* Pointer to the GPU buffer with the RNN initial hidden state. The same buffer address `hx` should be provided in prior [cudnnRNNForward\(\)](#) and [cudnnRNNBackwardData\\_v8\(\)](#) calls.

### **yDesc**

*Input.* A previously initialized descriptor corresponding to the RNN model output data. This is the same RNN data descriptor as used in prior [cudnnRNNForward\(\)](#) and [cudnnRNNBackwardData\\_v8\(\)](#) calls.

### **y**

*Output.* Pointer to the GPU buffer with the primary RNN output as generated by the prior [cudnnRNNForward\(\)](#) call. Data in the `y` buffer are described by the `yDesc` descriptor. Elements in the `y` tensor (including elements in padding vectors) must be densely packed.

### **weightSpaceSize**

*Input.* Specifies the size in bytes of the provided weight-space buffer.



**dweightSpace**

*Output.* Address of the weight space buffer in GPU memory.

**workSpaceSize**

*Input.* Specifies the size in bytes of the provided workspace buffer.

**workSpace**

*Input/Output.* Address of the workspace buffer in GPU memory to store temporary data.

**reserveSpaceSize**

*Input.* Specifies the size in bytes of the reserve-space buffer.

**reserveSpace**

*Input/Output.* Address of the reserve-space buffer in GPU memory.

## Returns

**CUDNN\_STATUS\_SUCCESS**

No errors were detected while processing API input arguments and launching GPU kernels.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid or incompatible input argument was encountered. For example:

- ▶ some input descriptors are `NULL`
- ▶ settings in `rnnDesc`, `xDesc`, `yDesc`, or `hDesc` descriptors are invalid
- ▶ `weightSpaceSize`, `workSpaceSize`, or `reserveSpaceSize` values are too small
- ▶ the `addGrad` argument is not equal to `CUDNN_WGRAD_MODE_ADD`

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The process of launching a GPU kernel returned an error, or an earlier kernel did not complete successfully.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate CPU memory.

## 8.2.23. `cudnnRNNBackwardWeightsEx()`

This function has been deprecated in cuDNN 8.0. Use [`cudnnRNNBackwardWeights\_v8\(\)`](#) instead of `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 an 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 (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function [cudnnRNNBackwardWeights\(\)](#), the sequences in the mini-batch need to be sorted in descending order according to length.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **xDesc**

*Input.* A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

### **x**

*Input.* Data pointer to GPU memory associated with the tensor descriptors in the array `xDesc`. Must contain the exact same data previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

### **hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

### **hx**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `hxDesc`. If a `NULL` pointer is passed, the initial hidden state of the network will be initialized to zero. Must contain the exact same data previously passed into [cudnnRNNForwardTrainingEx\(\)](#), or be `NULL` if `NULL` was previously passed to [cudnnRNNForwardTrainingEx\(\)](#).

### **yDesc**

*Input.* A previously initialized RNN data descriptor. Must match or be the exact same descriptor previously passed into [cudnnRNNForwardTrainingEx\(\)](#).

**y**

*Input.* Data pointer to GPU memory associated with the output tensor descriptor `yDesc`. Must contain the exact same data previously produced by [cudnnRNNForwardTrainingEx\(\)](#).

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**dwDesc**

*Input.* Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

**dw**

*Input/Output.* Data pointer to GPU memory associated with the filter descriptor `dwDesc`.

**reserveSpace**

*Input.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The function does not support the provided configuration.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `xDesc`, `yDesc`, `hxDesc`, `dwDesc` is invalid, or has incorrect strides or dimensions.
- ▶ `workspaceSizeInBytes` is too small.
- ▶ `reserveSpaceSizeInBytes` is too small.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 8.2.24. [cudnnRNNForwardTraining\(\)](#)

This function is deprecated starting in cuDNN 8.0.0. Use [cudnnRNNForward\(\)](#) instead of `cudnnRNNForwardTraining()`.

```
cudnnStatus_t cudnnRNNForwardTraining(
    cudnnHandle_t          handle,
    const cudnnRNNDescriptor_t  rnnDesc,
```

```

const int          seqLength,
const cudnnTensorDescriptor_t *xDesc,
const void        *x,
const cudnnTensorDescriptor_t hxDesc,
const void        *hx,
const cudnnTensorDescriptor_t cxDesc,
const void        *cx,
const cudnnFilterDescriptor_t wDesc,
const void        *w,
const cudnnTensorDescriptor_t *yDesc,
void              *y,
const cudnnTensorDescriptor_t hyDesc,
void              *hy,
const cudnnTensorDescriptor_t cyDesc,
void              *cy,
void              *workspace,
size_t            workspaceSizeInBytes,
void              *reserveSpace,
size_t            reserveSpaceSizeInBytes)

```

This routine executes the recurrent neural network described by `rnnDesc` with inputs `x`, `hx`, and `cx`, weights `w` and outputs `y`, `hy`, and `cy`. `workspace` is required for intermediate storage. `reserveSpace` stores data required for training. The same `reserveSpace` data must be used for future calls to [cudnnRNNBackwardData\(\)](#) and [cudnnRNNBackwardWeights\(\)](#) if these execute on the same input data.

## Parameters

### handle

*Input.* Handle to a previously created cuDNN context.

### rnnDesc

*Input.* A previously initialized RNN descriptor.

### seqLength

*Input.* Number of iterations to unroll over. The value of this `seqLength` must not exceed the value that was used in the [cudnnGetRNNWorkspaceSize\(\)](#) function for querying the workspace size required to execute the RNN.

### xDesc

*Input.* An array of `seqLength` fully packed tensor descriptors. Each descriptor in the array should have three dimensions that describe the input data format to one recurrent iteration (one descriptor per RNN time-step). The first dimension (batch size) of the tensors may decrease from iteration element `n` to iteration element `n+1` but may not increase. Each tensor descriptor must have the same second dimension (RNN input vector length, `inputSize`). The third dimension of each tensor should be 1. Input vectors are expected to be arranged in the column-major order so strides in `xDesc` should be set as follows:

```
strideA[0]=inputSize, strideA[1]=1, strideA[2]=1
```

### x

*Input.* Data pointer to GPU memory associated with the array of tensor descriptors `xDesc`. The input vectors are expected to be packed contiguously with the first vector of iterations (time-step) `n+1` following directly the last vector of iteration `n`. In other words, input vectors for all RNN time-steps should be packed in the contiguous block of GPU memory with no gaps between the vectors.

**hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` the second dimension should match the `hiddenSize` argument.

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

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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument 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 used to initialize `rnnDesc`:

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

**cy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `cyDesc`. If a `NULL` pointer is passed, the final cell state of the network will not be saved.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**reserveSpace**

*Input/Output.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `hxDesc`, `cxDesc`, `wDesc`, `hyDesc`, `cyDesc` or one of the descriptors in `xDesc`, `yDesc` is invalid.
- ▶ The descriptors in one of `xDesc`, `hxDesc`, `cxDesc`, `wDesc`, `yDesc`, `hyDesc`, `cyDesc` have incorrect strides or dimensions.
- ▶ `workSpaceSizeInBytes` is too small.
- ▶ `reserveSpaceSizeInBytes` is too small.

**CUDNN\_STATUS\_INVALID\_VALUE**

[`cudaSetPersistentRNNPlan\(\)`](#) was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 8.2.25. `cudaRNNForwardTrainingEx()`

This function has been deprecated starting in cuDNN 8.0. Use [`cudaRNNForward\(\)`](#) instead of `cudaRNNForwardTrainingEx()`.

```

cudaStatus_t cudaRNNForwardTrainingEx(
    cudaHandle_t          handle,
    const cudaRNNDescriptor_t    rnnDesc,
    const cudaRNNDataDescriptor_t xDesc,
    const void            *x,
    const cudaTensorDescriptor_t hxDesc,
    const void            *hx,
    const cudaTensorDescriptor_t cxDesc,
    const void            *cx,
    const cudaFilterDescriptor_t wDesc,
    const void            *w,
    const cudaRNNDataDescriptor_t yDesc,
    void                *y,
    const cudaTensorDescriptor_t hyDesc,
    void                *hy,
    const cudaTensorDescriptor_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 (meaning, time major) and batch major are supported. For backward compatibility, the packed sequence major layout is supported. However, similar to the non-extended function [cudnnRNNForwardTraining\(\)](#), the sequences in the mini-batch need to be sorted in descending order according to length.

## Parameters

### **handle**

*Input.* Handle to a previously created cuDNN context.

### **rnnDesc**

*Input.* A previously initialized RNN descriptor.

### **xDesc**

*Input.* A previously initialized RNN Data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` need to match that of `yDesc`.

### **x**

*Input.* Data pointer to the GPU memory associated with the RNN data descriptor `xDesc`. The input vectors are expected to be laid out in memory according to the layout specified by `xDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

### **hxDesc**

*Input.* A fully packed tensor descriptor describing the initial hidden state of the RNN.

The first dimension of the tensor depends on the `direction` argument used to initialize `rnnDesc`. Moreover:

- ▶ If `direction` is `CUDNN_UNIDIRECTIONAL` then the first dimension should match the `numLayers` argument.



- ▶ If `direction` is `CUDNN_BIDIRECTIONAL` then the first dimension should match double the `numLayers` argument.

The second dimension must match the `batchSize` parameter in `xDesc`. The third dimension depends on whether RNN mode is `CUDNN_LSTM` and whether LSTM projection is enabled. Additionally:

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

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

The second dimension must match the first dimension of the tensors described in `xDesc`. The third dimension must match the `hiddenSize` argument used to initialize `rnnDesc`. The tensor must be fully packed.

#### **cx**

*Input.* Data pointer to GPU memory associated with the tensor descriptor `cxDesc`. If a `NULL` pointer is passed, the initial cell state of the network will be initialized to zero.

#### **wDesc**

*Input.* Handle to a previously initialized filter descriptor describing the weights for the RNN.

#### **w**

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

#### **yDesc**

*Input.* A previously initialized RNN data descriptor. The `dataType`, `layout`, `maxSeqLength`, `batchSize`, and `seqLengthArray` need to match that of `dyDesc` and `dxDesc`. The parameter `vectorSize` depends on whether the RNN mode is `CUDNN_LSTM` and whether LSTM projection is enabled and whether the network is bidirectional. Specifically:

- ▶ For a unidirectional network, if the RNN mode is `CUDNN_LSTM` and LSTM projection is enabled, the parameter `vectorSize` must match the `recProjSize` argument passed to `cudaAdvTrainSetRNNProjectionLayers()` call used to set `rnnDesc`. If the network is bidirectional, then multiply the value by 2.
- ▶ Otherwise, for unidirectional network, the parameter `vectorSize` must match the `hiddenSize` argument used to initialize `rnnDesc`. If the network is bidirectional, then multiply the value by 2.

**y**

*Output.* Data pointer to GPU memory associated with the RNN data descriptor `yDesc`. The input vectors are expected to be laid out in memory according to the layout specified by `yDesc`. The elements in the tensor (including elements in the padding vector) must be densely packed, and no strides are supported.

**hyDesc**

*Input.* A fully packed tensor descriptor describing the final hidden state of the RNN. The descriptor must be set exactly the same as `hxDesc`.

**hy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `hyDesc`. If a `NULL` pointer is passed, the final hidden state of the network will not be saved.

**cyDesc**

*Input.* A fully packed tensor descriptor describing the final cell state for LSTM networks. The descriptor must be set exactly the same as `cxDesc`.

**cy**

*Output.* Data pointer to GPU memory associated with the tensor descriptor `cyDesc`. If a `NULL` pointer is passed, the final cell state of the network will not be saved.

**kDesc**

Reserved. Users may pass in `NULL`.

**keys**

Reserved. Users may pass in `NULL`.

**cDesc**

Reserved. Users may pass in `NULL`.

**cAttn**

Reserved. Users may pass in `NULL`.

**iDesc**

Reserved. Users may pass in `NULL`.

**iAttn**

Reserved. Users may pass in `NULL`.

**qDesc**

Reserved. Users may pass in `NULL`.

**queries**

Reserved. Users may pass in `NULL`.

**workspace**

*Input.* Data pointer to GPU memory to be used as a workspace for this call.

**workspaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `workspace`.

**reserveSpace**

*Input/Output.* Data pointer to GPU memory to be used as a reserve space for this call.

**reserveSpaceSizeInBytes**

*Input.* Specifies the size in bytes of the provided `reserveSpace`.

**Returns**

**CUDNN\_STATUS\_SUCCESS**

The function launched successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

At least one of the following conditions are met:

- ▶ Variable sequence length input is passed in while `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is used.
- ▶ `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` is used on pre-Pascal devices.
- ▶ Double input/output is used for `CUDNN_RNN_ALGO_PERSIST_STATIC`.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of the following conditions are met:

- ▶ The descriptor `rnnDesc` is invalid.
- ▶ At least one of the descriptors `xDesc`, `yDesc`, `hxDesc`, `cxDesc`, `wDesc`, `hyDesc`, and `cyDesc` is invalid, or have incorrect strides or dimensions.
- ▶ `workspaceSizeInBytes` is too small.
- ▶ `reserveSpaceSizeInBytes` is too small.

**CUDNN\_STATUS\_INVALID\_VALUE**

[`cudaSetPersistentRNNPlan\(\)`](#) was not called prior to the current function when `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` was selected in the RNN descriptor.

**CUDNN\_STATUS\_EXECUTION\_FAILED**

The function failed to launch on the GPU.

**CUDNN\_STATUS\_ALLOC\_FAILED**

The function was unable to allocate memory.

## 8.2.26. `cudaSetCTCLossDescriptor()`

This function sets a CTC loss function descriptor. See also the extended version [`cudaSetCTCLossDescriptorEx\(\)`](#) to set the input normalization mode.

```

cudaStatus_t cudaSetCTCLossDescriptor(
    cudaCTCLossDescriptor_t    ctcLossDesc,
    cudaDataType_t             compType)
    
```

When the extended version [`cudaSetCTCLossDescriptorEx\(\)`](#) is used with `normMode` set to `CUDNN_LOSS_NORMALIZATION_NONE` and the `gradMode` set

to `CUDNN_NOT_PROPAGATE_NAN`, then it is the same as the current function `cudaSetCtCLossDescriptor()`, meaning:

```
cudaSetCtCLossDescriptor(*) = cudaSetCtCLossDescriptorEx(*,
normMode=CUDNN_LOSS_NORMALIZATION_NONE, gradMode=CUDNN_NOT_PROPAGATE_NAN)
```

## Parameters

### **ctcLossDesc**

*Output.* CTC loss descriptor to be set.

### **compType**

*Input.* Compute type for this CTC loss function.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The function returned successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

At least one of the input parameters passed is invalid.

## 8.2.27. `cudaSetCtCLossDescriptorEx()`

This function is an extension of `cudaSetCtCLossDescriptor()`. This function provides an additional interface `normMode` to set the input normalization mode for the CTC loss function, and `gradMode` to control the NaN propagation type.

```
cudaStatus_t cudaSetCtCLossDescriptorEx(
    cudaCtCLossDescriptor_t    ctcLossDesc,
    cudaDataType_t             compType,
    cudaLossNormalizationMode_t normMode,
    cudaNanPropagation_t       gradMode)
```

When this function `cudaSetCtCLossDescriptorEx()` is used with `normMode` set to `CUDNN_LOSS_NORMALIZATION_NONE` and the `gradMode` set to `CUDNN_NOT_PROPAGATE_NAN`, then it is the same as `cudaSetCtCLossDescriptor()`, meaning:

```
cudaSetCtCLossDescriptor(*) = cudaSetCtCLossDescriptorEx(*,
normMode=CUDNN_LOSS_NORMALIZATION_NONE, gradMode=CUDNN_NOT_PROPAGATE_NAN)
```

## Parameters

### **ctcLossDesc**

*Output.* CTC loss descriptor to be set.

### **compType**

*Input.* Compute type for this CTC loss function.

### **normMode**

*Input.* Input normalization type for this CTC loss function. For more information, refer to `cudaLossNormalizationMode_t`.

### **gradMode**

*Input.* NaN propagation type for this CTC loss function. For  $L$  the sequence length,  $R$  the number of repeated letters in the sequence, and  $T$  the length of sequential data, the following applies: when a sample with  $L+R > T$  is encountered during the gradient calculation, if `gradMode` is set to `CUDNN_PROPAGATE_NAN` (refer to

[`cudaNanPropagation\_t`](#)), then the CTC loss function does not write to the gradient buffer for that sample. Instead, the current values, even not finite, are retained. If `gradMode` is set to `CUDNN_NOT_PROPAGATE_NAN`, then the gradient for that sample is set to zero. This guarantees a finite gradient.

## Returns

### `CUDNN_STATUS_SUCCESS`

The function returned successfully.

### `CUDNN_STATUS_BAD_PARAM`

At least one of the input parameters passed is invalid.

## 8.2.28. `cudaSetCTCLossDescriptor_v8()`

Many CTC API functions are updated in cuDNN version 8.0.0 to support CUDA graphs. In order to do so, a new parameter is needed, `maxLabelLength`. Now that label and input data are assumed to be in GPU memory, this information is not otherwise readily available.

```
cudaStatus_t cudaSetCTCLossDescriptorEx(
    cudaCTCLossDescriptor_t    ctcLossDesc,
    cudaDataType_t             compType,
    cudaLossNormalizationMode_t normMode,
    cudaNanPropagation_t       gradMode,
    int                        maxLabelLength)
```

## Parameters

### `ctcLossDesc`

*Output.* CTC loss descriptor to be set.

### `compType`

*Input.* Compute type for this CTC loss function.

### `normMode`

*Input.* Input normalization type for this CTC loss function. For more information, refer to [`cudaLossNormalizationMode\_t`](#).

### `gradMode`

*Input.* NaN propagation type for this CTC loss function. For  $L$  the sequence length,  $R$  the number of repeated letters in the sequence, and  $T$  the length of sequential data, the following applies: when a sample with  $L+R > T$  is encountered during the gradient calculation, if `gradMode` is set to `CUDNN_PROPAGATE_NAN` (refer to [`cudaNanPropagation\_t`](#)), then the CTC loss function does not write to the gradient buffer for that sample. Instead, the current values, even not finite, are retained. If `gradMode` is set to `CUDNN_NOT_PROPAGATE_NAN`, then the gradient for that sample is set to zero. This guarantees a finite gradient.

### `maxLabelLength`

*Input.* The maximum label length from the labels data.

## Returns

**CUDNN\_STATUS\_SUCCESS**

The function returned successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

At least one of input parameters passed is invalid.

---

# Chapter 9. cuDNN Backend API

This chapter documents the current implemented behavior of the `cudaBackend*` API introduced in cuDNN version 8.x. Users specify the computational case, set up an execution plan for it, and execute the computation via numerous descriptors. The typical use pattern for a descriptor with attributes consists of the following sequence of API calls:

1. `cudaBackendCreateDescriptor()` creates a descriptor of a specified type.
2. `cudaBackendSetAttribute()` sets the values of a settable attribute for the descriptor. All required attributes must be set before the next step.
3. `cudaBackendFinalize()` finalizes the descriptor.
4. `cudaBackendGetAttribute()` gets the values of an attribute from a finalized descriptor.

The enumeration type `cudaBackendDescriptorType_t` enumerates the list of valid cuDNN backend descriptor types. The enumeration type `cudaBackendAttributeName_t` enumerates the list of valid attributes. Each descriptor type in `cudaBackendDescriptorType_t` has a disjoint subset of valid attribute values of `cudaBackendAttributeName_t`. The full description of each descriptor type and their attributes are specified in the [Backend Descriptor Types](#) section.

## 9.1. Data Type References

These are the data type references for the cuDNN Backend API.

### 9.1.1. Enumeration Types

These are the enumeration types for the cuDNN Backend API.

#### 9.1.1.1. `cudaBackendAttributeName_t`

`cudaBackendAttributeName_t` is an enumerated type that indicates the backend descriptor attributes that can be set or get using `cudaBackendSetAttribute()` and `cudaBackendGetAttribute()` functions. The backend descriptor to which an attribute belongs is identified by the prefix of the attribute name.

```
typedef enum {
    CUDA_ATTR_POINTWISE_MODE           = 0,
    CUDA_ATTR_POINTWISE_MATH_PREC     = 1,
```

```

CUDNN_ATTR_POINTWISE_NAN_PROPAGATION      = 2,
CUDNN_ATTR_POINTWISE_RELU_LOWER_CLIP      = 3,
CUDNN_ATTR_POINTWISE_RELU_UPPER_CLIP      = 4,
CUDNN_ATTR_POINTWISE_RELU_LOWER_CLIP_SLOPE = 5,
CUDNN_ATTR_POINTWISE_ELU_ALPHA            = 6,
CUDNN_ATTR_POINTWISE_SOFTPLUS_BETA        = 7,
CUDNN_ATTR_POINTWISE_SWISH_BETA           = 8,
CUDNN_ATTR_POINTWISE_AXIS                 = 9,

CUDNN_ATTR_CONVOLUTION_COMP_TYPE          = 100,
CUDNN_ATTR_CONVOLUTION_CONV_MODE          = 101,
CUDNN_ATTR_CONVOLUTION_DILATIONS          = 102,
CUDNN_ATTR_CONVOLUTION_FILTER_STRIDES    = 103,
CUDNN_ATTR_CONVOLUTION_POST_PADDING      = 104,
CUDNN_ATTR_CONVOLUTION_PRE_PADDING        = 105,
CUDNN_ATTR_CONVOLUTION_SPATIAL_DIMS      = 106,

CUDNN_ATTR_ENGINEHEUR_MODE                = 200,
CUDNN_ATTR_ENGINEHEUR_OPERATION_GRAPH    = 201,
CUDNN_ATTR_ENGINEHEUR_RESULTS            = 202,

CUDNN_ATTR_ENGINECFG_ENGINE               = 300,
CUDNN_ATTR_ENGINECFG_INTERMEDIATE_INFO    = 301,
CUDNN_ATTR_ENGINECFG_KNOB_CHOICES        = 302,

CUDNN_ATTR_EXECUTION_PLAN_HANDLE          = 400,
CUDNN_ATTR_EXECUTION_PLAN_ENGINE_CONFIG   = 401,
CUDNN_ATTR_EXECUTION_PLAN_WORKSPACE_SIZE = 402,
CUDNN_ATTR_EXECUTION_PLAN_COMPUTED_INTERMEDIATE_UIDS = 403,
CUDNN_ATTR_EXECUTION_PLAN_RUN_ONLY_INTERMEDIATE_UIDS = 404,

CUDNN_ATTR_INTERMEDIATE_INFO_UNIQUE_ID    = 500,
CUDNN_ATTR_INTERMEDIATE_INFO_SIZE         = 501,
CUDNN_ATTR_INTERMEDIATE_INFO_DEPENDENT_DATA_UIDS = 502,
CUDNN_ATTR_INTERMEDIATE_INFO_DEPENDENT_ATTRIBUTES = 503,

CUDNN_ATTR_KNOB_CHOICE_KNOB_TYPE          = 600,
CUDNN_ATTR_KNOB_CHOICE_KNOB_VALUE        = 601,

CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_ALPHA = 700,
CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_BETA = 701,
CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_CONV_DESC = 702,
CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_W = 703,
CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_X = 704,
CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_Y = 705,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_ALPHA = 706,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_BETA = 707,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_CONV_DESC = 708,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_W = 709,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_DX = 710,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_DY = 711,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_ALPHA = 712,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_BETA = 713,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_CONV_DESC = 714,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_DW = 715,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_X = 716,
CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_DY = 717,
CUDNN_ATTR_OPERATION_POINTWISE_PW_DESCRIPTOR = 750,
CUDNN_ATTR_OPERATION_POINTWISE_XDESC     = 751,
CUDNN_ATTR_OPERATION_POINTWISE_BDESC     = 752,
CUDNN_ATTR_OPERATION_POINTWISE_YDESC     = 753,
CUDNN_ATTR_OPERATION_POINTWISE_ALPHA1    = 754,
CUDNN_ATTR_OPERATION_POINTWISE_ALPHA2    = 755,
CUDNN_ATTR_OPERATION_POINTWISE_DXDESC    = 756,
CUDNN_ATTR_OPERATION_POINTWISE_DYDESC    = 757,
CUDNN_ATTR_OPERATION_POINTWISE_TDESC     = 758,

```



```

CUDNN_ATTR_OPERATION_GENSTATS_MODE = 770,
CUDNN_ATTR_OPERATION_GENSTATS_MATH_PREC = 771,
CUDNN_ATTR_OPERATION_GENSTATS_XDESC = 772,
CUDNN_ATTR_OPERATION_GENSTATS_SUMDESC = 773,
CUDNN_ATTR_OPERATION_GENSTATS_SQSUMDESC = 774,

CUDNN_ATTR_OPERATION_BN_FINALIZE_STATS_MODE = 780,
CUDNN_ATTR_OPERATION_BN_FINALIZE_MATH_PREC = 781,
CUDNN_ATTR_OPERATION_BN_FINALIZE_Y_SUM_DESC = 782,
CUDNN_ATTR_OPERATION_BN_FINALIZE_Y_SQ_SUM_DESC = 783,
CUDNN_ATTR_OPERATION_BN_FINALIZE_SCALE_DESC = 784,
CUDNN_ATTR_OPERATION_BN_FINALIZE_BIAS_DESC = 785,
CUDNN_ATTR_OPERATION_BN_FINALIZE_PREV_RUNNING_MEAN_DESC = 786,
CUDNN_ATTR_OPERATION_BN_FINALIZE_PREV_RUNNING_VAR_DESC = 787,
CUDNN_ATTR_OPERATION_BN_FINALIZE_UPDATED_RUNNING_MEAN_DESC = 788,
CUDNN_ATTR_OPERATION_BN_FINALIZE_UPDATED_RUNNING_VAR_DESC = 789,
CUDNN_ATTR_OPERATION_BN_FINALIZE_SAVED_MEAN_DESC = 790,
CUDNN_ATTR_OPERATION_BN_FINALIZE_SAVED_INV_STD_DESC = 791,
CUDNN_ATTR_OPERATION_BN_FINALIZE_EQ_SCALE_DESC = 792,
CUDNN_ATTR_OPERATION_BN_FINALIZE_EQ_BIAS_DESC = 793,
CUDNN_ATTR_OPERATION_BN_FINALIZE_ACCUM_COUNT_DESC = 794,
CUDNN_ATTR_OPERATION_BN_FINALIZE_EPSILON_DESC = 795,
CUDNN_ATTR_OPERATION_BN_FINALIZE_EXP_AVERAGE_FACTOR_DESC = 796,

CUDNN_ATTR_OPERATIONGRAPH_HANDLE = 800,
CUDNN_ATTR_OPERATIONGRAPH_OPS = 801,
CUDNN_ATTR_OPERATIONGRAPH_ENGINE_GLOBAL_COUNT = 802,

CUDNN_ATTR_TENSOR_BYTE_ALIGNMENT = 900,
CUDNN_ATTR_TENSOR_DATA_TYPE = 901,
CUDNN_ATTR_TENSOR_DIMENSIONS = 902,
CUDNN_ATTR_TENSOR_STRIDES = 903,
CUDNN_ATTR_TENSOR_VECTOR_COUNT = 904,
CUDNN_ATTR_TENSOR_VECTORIZED_DIMENSION = 905,
CUDNN_ATTR_TENSOR_UNIQUE_ID = 906,
CUDNN_ATTR_TENSOR_IS_VIRTUAL = 907,
CUDNN_ATTR_TENSOR_IS_BY_VALUE = 908,
CUDNN_ATTR_TENSOR_REORDERING_MODE = 909,
CUDNN_ATTR_TENSOR_RAGGED_OFFSET_DESC = 910,

CUDNN_ATTR_VARIANT_PACK_UNIQUE_IDS = 1000,
CUDNN_ATTR_VARIANT_PACK_DATA_POINTERS = 1001,
CUDNN_ATTR_VARIANT_PACK_INTERMEDIATES = 1002,
CUDNN_ATTR_VARIANT_PACK_WORKSPACE = 1003,

CUDNN_ATTR_LAYOUT_INFO_TENSOR_UID = 1100,
CUDNN_ATTR_LAYOUT_INFO_TYPES = 1101,

CUDNN_ATTR_KNOB_INFO_TYPE = 1200,
CUDNN_ATTR_KNOB_INFO_MAXIMUM_VALUE = 1201,
CUDNN_ATTR_KNOB_INFO_MINIMUM_VALUE = 1202,
CUDNN_ATTR_KNOB_INFO_STRIDE = 1203,

CUDNN_ATTR_ENGINE_OPERATION_GRAPH = 1300,
CUDNN_ATTR_ENGINE_GLOBAL_INDEX = 1301,
CUDNN_ATTR_ENGINE_KNOB_INFO = 1302,
CUDNN_ATTR_ENGINE_NUMERICAL_NOTE = 1303,
CUDNN_ATTR_ENGINE_LAYOUT_INFO = 1304,
CUDNN_ATTR_ENGINE_BEHAVIOR_NOTE = 1305,

CUDNN_ATTR_MATMUL_COMP_TYPE = 1500,
CUDNN_ATTR_MATMUL_PADDING_VALUE = 1501,

CUDNN_ATTR_OPERATION_MATMUL_ADESC = 1520,
CUDNN_ATTR_OPERATION_MATMUL_BDESC = 1521,
CUDNN_ATTR_OPERATION_MATMUL_CDESC = 1522,
CUDNN_ATTR_OPERATION_MATMUL_DESC = 1523,

```

```

CUDNN_ATTR_OPERATION_MATMUL_IRREGULARLY_STRIDED_BATCH_COUNT = 1524,
CUDNN_ATTR_OPERATION_MATMUL_GEMM_M_OVERRIDE_DESC = 1525,
CUDNN_ATTR_OPERATION_MATMUL_GEMM_N_OVERRIDE_DESC = 1526,
CUDNN_ATTR_OPERATION_MATMUL_GEMM_K_OVERRIDE_DESC = 1527,

CUDNN_ATTR_REDUCTION_OPERATOR = 1600,
CUDNN_ATTR_REDUCTION_COMP_TYPE = 1601,

CUDNN_ATTR_OPERATION_REDUCTION_XDESC = 1610,
CUDNN_ATTR_OPERATION_REDUCTION_YDESC = 1611,
CUDNN_ATTR_OPERATION_REDUCTION_DESC = 1612,

CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_MATH_PREC = 1620,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_MEAN_DESC = 1621,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_INVSTD_DESC = 1622,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_BN_SCALE_DESC = 1623,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_X_DESC = 1624,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_DY_DESC = 1625,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_DBN_SCALE_DESC = 1626,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_DBN_BIAS_DESC = 1627,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_EQ_DY_SCALE_DESC = 1628,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_EQ_X_SCALE_DESC = 1629,
CUDNN_ATTR_OPERATION_BN_BWD_WEIGHTS_EQ_BIAS = 1630,

CUDNN_ATTR_RESAMPLE_MODE = 1700,
CUDNN_ATTR_RESAMPLE_COMP_TYPE = 1701,
CUDNN_ATTR_RESAMPLE_SPATIAL_DIMS = 1702,
CUDNN_ATTR_RESAMPLE_POST_PADDINGS = 1703,
CUDNN_ATTR_RESAMPLE_PRE_PADDINGS = 1704,
CUDNN_ATTR_RESAMPLE_STRIDES = 1705,
CUDNN_ATTR_RESAMPLE_WINDOW_DIMS = 1706,
CUDNN_ATTR_RESAMPLE_NAN_PROPAGATION = 1707,
CUDNN_ATTR_RESAMPLE_PADDING_MODE = 1708,

CUDNN_ATTR_OPERATION_RESAMPLE_FWD_XDESC = 1710,
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_YDESC = 1711,
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_IDXDESC = 1712,
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_ALPHA = 1713,
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_BETA = 1714,
CUDNN_ATTR_OPERATION_RESAMPLE_FWD_DESC = 1716,

CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DXDESC = 1720,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DYDESC = 1721,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_IDXDESC = 1722,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_ALPHA = 1723,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_BETA = 1724,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DESC = 1725,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_XDESC = 1726,
CUDNN_ATTR_OPERATION_RESAMPLE_BWD_YDESC = 1727,

CUDNN_ATTR_OPERATION_CONCAT_AXIS = 1800,
CUDNN_ATTR_OPERATION_CONCAT_INPUT_DESCS = 1801,
CUDNN_ATTR_OPERATION_CONCAT_INPLACE_INDEX = 1802,
CUDNN_ATTR_OPERATION_CONCAT_OUTPUT_DESC = 1803,

CUDNN_ATTR_OPERATION_SIGNAL_MODE = 1900,
CUDNN_ATTR_OPERATION_SIGNAL_FLAGDESC = 1901,
CUDNN_ATTR_OPERATION_SIGNAL_VALUE = 1902,
CUDNN_ATTR_OPERATION_SIGNAL_XDESC = 1903,
CUDNN_ATTR_OPERATION_SIGNAL_YDESC = 1904,

CUDNN_ATTR_OPERATION_NORM_FWD_MODE = 2000,
CUDNN_ATTR_OPERATION_NORM_FWD_PHASE = 2001,
CUDNN_ATTR_OPERATION_NORM_FWD_XDESC = 2002,
CUDNN_ATTR_OPERATION_NORM_FWD_MEAN_DESC = 2003,
CUDNN_ATTR_OPERATION_NORM_FWD_INV_VARIANCE_DESC = 2004,

```

```

CUDNN_ATTR_OPERATION_NORM_FWD_SCALE_DESC           = 2005,
CUDNN_ATTR_OPERATION_NORM_FWD_BIAS_DESC            = 2006,
CUDNN_ATTR_OPERATION_NORM_FWD_EPSILON_DESC        = 2007,
CUDNN_ATTR_OPERATION_NORM_FWD_EXP_AVG_FACTOR_DESC  = 2008,
CUDNN_ATTR_OPERATION_NORM_FWD_INPUT_RUNNING_MEAN_DESC = 2009,
CUDNN_ATTR_OPERATION_NORM_FWD_INPUT_RUNNING_VAR_DESC = 2010,
CUDNN_ATTR_OPERATION_NORM_FWD_OUTPUT_RUNNING_MEAN_DESC = 2011,
CUDNN_ATTR_OPERATION_NORM_FWD_OUTPUT_RUNNING_VAR_DESC = 2012,
CUDNN_ATTR_OPERATION_NORM_FWD_YDESC               = 2013,
CUDNN_ATTR_OPERATION_NORM_FWD_PEER_STAT_DESCS      = 2014,

CUDNN_ATTR_OPERATION_NORM_BWD_MODE                 = 2100,
CUDNN_ATTR_OPERATION_NORM_BWD_XDESC                = 2101,
CUDNN_ATTR_OPERATION_NORM_BWD_MEAN_DESC            = 2102,
CUDNN_ATTR_OPERATION_NORM_BWD_INV_VARIANCE_DESC    = 2103,
CUDNN_ATTR_OPERATION_NORM_BWD_DYDESC              = 2104,
CUDNN_ATTR_OPERATION_NORM_BWD_SCALE_DESC           = 2105,
CUDNN_ATTR_OPERATION_NORM_BWD_EPSILON_DESC        = 2106,
CUDNN_ATTR_OPERATION_NORM_BWD_DSCALE_DESC          = 2107,
CUDNN_ATTR_OPERATION_NORM_BWD_DBIAS_DESC           = 2108,
CUDNN_ATTR_OPERATION_NORM_BWD_DXDESC              = 2109,
CUDNN_ATTR_OPERATION_NORM_BWD_PEER_STAT_DESCS      = 2110,

CUDNN_ATTR_OPERATION_RESHAPE_XDESC = 2200,
CUDNN_ATTR_OPERATION_RESHAPE_YDESC = 2201,

CUDNN_ATTR_RNG_DISTRIBUTION                    = 2300,
CUDNN_ATTR_RNG_NORMAL_DIST_MEAN                = 2301,
CUDNN_ATTR_RNG_NORMAL_DIST_STANDARD_DEVIATION = 2302,
CUDNN_ATTR_RNG_UNIFORM_DIST_MAXIMUM            = 2303,
CUDNN_ATTR_RNG_UNIFORM_DIST_MINIMUM           = 2304,
CUDNN_ATTR_RNG_BERNOULLI_DIST_PROBABILITY      = 2305,

CUDNN_ATTR_OPERATION_RNG_YDESC                 = 2310,
CUDNN_ATTR_OPERATION_RNG_SEED                  = 2311,
CUDNN_ATTR_OPERATION_RNG_DESC                  = 2312,
CUDNN_ATTR_OPERATION_RNG_OFFSET_DESC           = 2313,
} cudnnBackendAttributeName_t;

```

### 9.1.1.2. `cudnnBackendAttributeType_t`

The enumeration type `cudnnBackendAttributeType_t` specifies the data type of an attribute of a cuDNN backend descriptor. It is used to specify the type of data pointed to by the `void *arrayOfElements` argument of `cudnnBackendSetAttribute()` and `cudnnBackendGetAttribute()`.

```

typedef enum {
    CUDNN_TYPE_HANDLE = 0,
    CUDNN_TYPE_DATA_TYPE,
    CUDNN_TYPE_BOOLEAN,
    CUDNN_TYPE_INT64,
    CUDNN_TYPE_FLOAT,
    CUDNN_TYPE_DOUBLE,
    CUDNN_TYPE_VOID_PTR,
    CUDNN_TYPE_CONVOLUTION_MODE,
    CUDNN_TYPE_HEUR_MODE,
    CUDNN_TYPE_KNOB_TYPE,
    CUDNN_TYPE_NAN_PROPOGATION,
    CUDNN_TYPE_NUMERICAL_NOTE,
    CUDNN_TYPE_LAYOUT_TYPE,
    CUDNN_TYPE_ATTRIB_NAME,
    CUDNN_TYPE_POINTWISE_MODE,
    CUDNN_TYPE_BACKEND_DESCRIPTOR,
    CUDNN_TYPE_GENSTATS_MODE,
    CUDNN_TYPE_BN_FINALIZE_STATS_MODE,

```

```

    CUDNN_TYPE_REDUCTION_OPERATOR_TYPE,
    CUDNN_TYPE_BEHAVIOR_NOTE,
    CUDNN_TYPE_TENSOR_REORDERING_MODE,
    CUDNN_TYPE_RESAMPLE_MODE,
    CUDNN_TYPE_PADDING_MODE,
    CUDNN_TYPE_INT32,
    CUDNN_TYPE_CHAR,
    CUDNN_TYPE_SIGNAL_MODE,
    CUDNN_TYPE_FRACTION,
    CUDNN_TYPE_NORM_MODE,
    CUDNN_TYPE_NORM_FWD_PHASE,
    CUDNN_TYPE_RNG_DISTRIBUTION
} cudnnBackendAttributeType_t;

```

Table 51. Attribute Types for `cudnnBackendAttributeType_t`

<code>cudnnBackendAttributeType_t</code>	Attribute type
<code>CUDNN_TYPE_HANDLE</code>	<a href="#">cudnnHandle_t</a>
<code>CUDNN_TYPE_DATA_TYPE</code>	<a href="#">cudnnDataType_t</a>
<code>CUDNN_TYPE_BOOLEAN</code>	<code>bool</code>
<code>CUDNN_TYPE_INT64</code>	<code>int64_t</code>
<code>CUDNN_TYPE_FLOAT</code>	<code>float</code>
<code>CUDNN_TYPE_DOUBLE</code>	<code>double</code>
<code>CUDNN_TYPE_VOID_PTR</code>	<code>void *</code>
<code>CUDNN_TYPE_CONVOLUTION_MODE</code>	<a href="#">cudnnConvolutionMode_t</a>
<code>CUDNN_TYPE_HEUR_MODE</code>	<a href="#">cudnnBackendHeurMode_t</a>
<code>CUDNN_TYPE_KNOB_TYPE</code>	<a href="#">cudnnBackendKnobType_t</a>
<code>CUDNN_TYPE_NAN_PROPOGATION</code>	<a href="#">cudnnNanPropagation_t</a>
<code>CUDNN_TYPE_NUMERICAL_NOTE</code>	<a href="#">cudnnBackendNumericalNote_t</a>
<code>CUDNN_TYPE_LAYOUT_TYPE</code>	<a href="#">cudnnBackendLayoutType_t</a>
<code>CUDNN_TYPE_ATTRIB_NAME</code>	<a href="#">cudnnBackendAttributeName_t</a>
<code>CUDNN_TYPE_POINTWISE_MODE</code>	<a href="#">cudnnPointwiseMode_t</a>
<code>CUDNN_TYPE_BACKEND_DESCRIPTOR</code>	<a href="#">cudnnBackendDescriptor_t</a>
<code>CUDNN_TYPE_GENSTATS_MODE</code>	<a href="#">cudnnGenStatsMode_t</a>
<code>CUDNN_TYPE_BN_FINALIZE_STATS_MODE</code>	<a href="#">cudnnBnFinalizeStatsMode_t</a>
<code>CUDNN_TYPE_REDUCTION_OPERATOR_TYPE</code>	<a href="#">cudnnReduceTensorOp_t</a>
<code>CUDNN_TYPE_BEHAVIOR_NOTE</code>	<a href="#">cudnnBackendBehaviorNote_t</a>
<code>CUDNN_TYPE_TENSOR_REORDERING_MODE</code>	<a href="#">cudnnBackendTensorReordering_t</a>
<code>CUDNN_TYPE_RESAMPLE_MODE</code>	<a href="#">cudnnResampleMode_t</a>
<code>CUDNN_TYPE_PADDING_MODE</code>	<a href="#">cudnnPaddingMode_t</a>
<code>CUDNN_TYPE_INT32</code>	<code>int32_t</code>
<code>CUDNN_TYPE_CHAR</code>	<code>char</code>
<code>CUDNN_TYPE_SIGNAL_MODE</code>	<a href="#">cudnnSignalMode_t</a>
<code>CUDNN_TYPE_FRACTION</code>	<a href="#">cudnnFraction_t</a>

<code>cudaBackendAttributeType_t</code>	Attribute type
<code>CUDNN_TYPE_NORM_MODE</code>	<a href="#"><code>cudaBackendNormMode_t</code></a>
<code>CUDNN_TYPE_NORM_FWD_PHASE</code>	<a href="#"><code>cudaBackendNormFwdPhase_t</code></a>
<code>CUDNN_TYPE_RNG_DISTRIBUTION</code>	<a href="#"><code>cudaRngDistribution_t</code></a>

### 9.1.1.3. `cudaBackendBehaviorNote_t`

`cudaBackendBehaviorNote_t` is an enumerated type that indicates queryable behavior notes of an engine. Users can query for an array of behavior notes from an `CUDNN_BACKEND_ENGINE_DESC` using the [`cudaBackendGetAttribute\(\)`](#) function.

```
typedef enum {
    CUDNN_BEHAVIOR_NOTE_RUNTIME_COMPILATION = 0,
    CUDNN_BEHAVIOR_NOTE_REQUIRES_FILTER_INT8x32_REORDER = 1,
    CUDNN_BEHAVIOR_NOTE_REQUIRES_BIAS_INT8x32_REORDER = 2,
    CUDNN_BEHAVIOR_NOTE_TYPE_COUNT,
} cudaBackendBehaviorNote_t;
```

### 9.1.1.4. `cudaBackendDescriptorType_t`

`cudaBackendDescriptor_t` is an enumerated type that indicates the type of backend descriptors. Users create a backend descriptor of a particular type by passing a value from this enumerate to the [`cudaBackendCreateDescriptor\(\)`](#) function.

```
typedef enum {
    CUDNN_BACKEND_POINTWISE_DESCRIPTOR = 0,
    CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR,
    CUDNN_BACKEND_ENGINE_DESCRIPTOR,
    CUDNN_BACKEND_ENGINECFG_DESCRIPTOR,
    CUDNN_BACKEND_ENGINEHEUR_DESCRIPTOR,
    CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR,
    CUDNN_BACKEND_INTERMEDIATE_INFO_DESCRIPTOR,
    CUDNN_BACKEND_KNOB_CHOICE_DESCRIPTOR,
    CUDNN_BACKEND_KNOB_INFO_DESCRIPTOR,
    CUDNN_BACKEND_LAYOUT_INFO_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_FILTER_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_DATA_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR,
    CUDNN_BACKEND_OPERATIONGRAPH_DESCRIPTOR,
    CUDNN_BACKEND_VARIANT_PACK_DESCRIPTOR,
    CUDNN_BACKEND_TENSOR_DESCRIPTOR,
    CUDNN_BACKEND_MATMUL_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_BN_FINALIZE_STATISTICS_DESCRIPTOR,
    CUDNN_BACKEND_REDUCTION_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_BN_BWD_WEIGHTS_DESCRIPTOR,
    CUDNN_BACKEND_RESAMPLE_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_CONCAT_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_SIGNAL_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_NORM_FORWARD_DESCRIPTOR,
    CUDNN_BACKEND_OPERATION_NORM_BACKWARD_DESCRIPTOR,
} cudaBackendDescriptorType_t;
```

### 9.1.1.5. `cudaBackendHeurMode_t`

`cudaBackendHeurMode_t` is an enumerated type that indicates the operation mode of a `CUDNN_BACKEND_ENGINEHEUR_DESCRIPTOR`.

```
typedef enum {
    CUDNN_HEUR_MODE_INSTANT = 0,
    CUDNN_HEUR_MODE_B       = 1,
    CUDNN_HEUR_MODE_FALLBACK = 2,
    CUDNN_HEUR_MODE_A       = 3
}
```

#### Values

##### **CUDNN\_HEUR\_MODE\_A & CUDNN\_HEUR\_MODE\_INSTANT**

`CUDNN_HEUR_MODE_A` provides the exact same functionality as `CUDNN_HEUR_MODE_INSTANT`. The purpose of this renaming is to better match the naming of `CUDNN_HEUR_MODE_B`. Consider the use of `CUDNN_HEUR_MODE_INSTANT` as deprecated; instead, use `CUDNN_HEUR_MODE_A`.

`CUDNN_HEUR_MODE_A` utilizes a decision tree heuristic which provides optimal inference time on the CPU in comparison to `CUDNN_HEUR_MODE_B`.

`CUDNN_HEUR_MODE_A` and `CUDNN_HEUR_MODE_INSTANT` support the following operation node or operation graph:

- ▶ [ConvolutionFwd](#)
- ▶ [ConvolutionBwFilter](#)
- ▶ [ConvolutionBwData](#)
- ▶ [ConvBNfprop](#)
- ▶ [ConvBNwgrad](#)
- ▶ [ConvBiasAct](#)
- ▶ [ConvScaleBiasAct](#)
- ▶ [DgradDreluBNBwdWeight](#)
- ▶ [patterns supported by the runtime fusion engine](#)

All other operation graphs are not supported.

##### **CUDNN\_HEUR\_MODE\_B**

Can utilize the neural net based heuristics to improve generalization performance compared to `CUDNN_HEUR_MODE_INSTANT`. In cases where the neural net is utilized, inference time on the CPU will be increased by 10-100x compared to `CUDNN_HEUR_MODE_INSTANT`. These neural net heuristics are not supported for any of the following cases:

- ▶ 3-D convolutions
- ▶ Grouped convolutions (groupCount larger than 1)

- Dilated convolutions (any dilation for any spatial dimension larger than 1)

Further, the neural net is only enabled on x86 platforms when cuDNN is run on an A100 GPU. In cases where the neural net is not supported, `CUDNN_HEUR_MODE_B` will also fall back to `CUDNN_HEUR_MODE_INSTANT`. `CUDNN_HEUR_MODE_B` will fall back to `CUDNN_HEUR_MODE_INSTANT` in cases where the overhead of `CUDNN_HEUR_MODE_B` is projected to reduce overall network performance.

#### **CUDNN\_HEUR\_MODE\_FALLBACK**

This heuristic mode is intended to be used for finding fallback options which provide functional support (without any expectation of providing optimal GPU performance).

### 9.1.1.6. `cudaBackendKnobType_t`

`cudaBackendKnobType_t` is an enumerated type that indicates the type of performance knobs. Performance knobs are runtime settings to an engine that will affect its performance. Users can query for an array of performance knobs and their valid value range from a `CUDNN_BACKEND_ENGINE_DESCRIPTOR` using the `cudaBackendGetAttribute()` function. Users can set the choice for each knob using the `cudaBackendSetAttribute()` function with a `CUDNN_BACKEND_KNOB_CHOICE_DESCRIPTOR` descriptor.

```
typedef enum {
    CUDNN_KNOB_TYPE_SPLIT_K           = 0,
    CUDNN_KNOB_TYPE_SWIZZLE           = 1,
    CUDNN_KNOB_TYPE_TILE_SIZE         = 2,
    CUDNN_KNOB_TYPE_USE_TEX           = 3,
    CUDNN_KNOB_TYPE_EDGE               = 4,
    CUDNN_KNOB_TYPE_KBLOCK            = 5,
    CUDNN_KNOB_TYPE_LDGA              = 6,
    CUDNN_KNOB_TYPE_LDGB              = 7,
    CUDNN_KNOB_TYPE_CHUNK_K           = 8,
    CUDNN_KNOB_TYPE_SPLIT_H           = 9,
    CUDNN_KNOB_TYPE_WINO_TILE         = 10,
    CUDNN_KNOB_TYPE_MULTIPLY          = 11,
    CUDNN_KNOB_TYPE_SPLIT_K_BUF       = 12,
    CUDNN_KNOB_TYPE_TILEK             = 13,
    CUDNN_KNOB_TYPE_STAGES            = 14,
    CUDNN_KNOB_TYPE_REDUCTION_MODE    = 15,
    CUDNN_KNOB_TYPE_CTA_SPLIT_K_MODE  = 16,
    CUDNN_KNOB_TYPE_SPLIT_K_SLC       = 17,
    CUDNN_KNOB_TYPE_IDX_MODE          = 18,
    CUDNN_KNOB_TYPE_SLICED            = 19,
    CUDNN_KNOB_TYPE_SPLIT_RS          = 20,
    CUDNN_KNOB_TYPE_SINGLEBUFFER      = 21,
    CUDNN_KNOB_TYPE_LDGC              = 22,
    CUDNN_KNOB_TYPE_SPECFILT          = 23,
    CUDNN_KNOB_TYPE_KERNEL_CFG        = 24,
    CUDNN_KNOB_TYPE_WORKSPACE         = 25,

    CUDNN_KNOB_TYPE_COUNTS = 26,
} cudaBackendKnobType_t;
```

### 9.1.1.7. `cudaBackendLayoutType_t`

`cudaBackendLayoutType_t` is an enumerated type that indicates queryable layout requirements of an engine. Users can query for layout requirements from a

CUDNN\_BACKEND\_ENGINE\_DESC descriptor using the [cudnnBackendGetAttribute\(\)](#) function.

```
typedef enum {
    CUDNN_LAYOUT_TYPE_PREFERRED_NCHW = 0,
    CUDNN_LAYOUT_TYPE_PREFERRED_NHWC = 1,
    CUDNN_LAYOUT_TYPE_PREFERRED_PAD4CK = 2,
    CUDNN_LAYOUT_TYPE_PREFERRED_PAD8CK = 3,
    CUDNN_LAYOUT_TYPE_COUNT = 4,
} cudnnBackendLayoutType_t;
```

### 9.1.1.8. cudnnBackendNormFwdPhase\_t

cudnnBackendNormFwdPhase\_t is an enumerated type used to distinguish the inference and training phase of the normalization forward operation.

```
typedef enum {
    CUDNN_NORM_FWD_INFERENCE = 0,
    CUDNN_NORM_FWD_TRAINING = 1,
} cudnnBackendNormFwdPhase_t;
```

### 9.1.1.9. cudnnBackendNormMode\_t

cudnnBackendNormMode\_t is an enumerated type to indicate the normalization mode in the backend normalization forward and normalization backward operations.

For reference:

- ▶ The definition of layer normalization can be found in the [Layer Normalization](#) paper.
- ▶ The definition of instance normalization can be found in the [Instance Normalization: The Missing Ingredient for Fast Stylization](#) paper.
- ▶ The definition of batch normalization can be found in the [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#) paper.

CUDNN\_GROUP\_NORM is not yet supported. If you try to use it, cuDNN returns a CUDNN\_STATUS\_INTERNAL\_ERROR error.

```
typedef enum {
    CUDNN_LAYER_NORM = 0,
    CUDNN_INSTANCE_NORM = 1,
    CUDNN_BATCH_NORM = 2,
    CUDNN_GROUP_NORM = 3,
} cudnnBackendNormMode_t;
```

### 9.1.1.10. cudnnBackendNumericalNote\_t

cudnnBackendNumericalNote\_t is an enumerated type that indicates queryable numerical properties of an engine. Users can query for an array of numerical notes from an CUDNN\_BACKEND\_ENGINE\_DESC using the [cudnnBackendGetAttribute\(\)](#) function.

```
typedef enum {
    CUDNN_NUMERICAL_NOTE_TENSOR_CORE = 0,
    CUDNN_NUMERICAL_NOTE_DOWN_CONVERT_INPUTS,
    CUDNN_NUMERICAL_NOTE_REDUCED_PRECISION_REDUCTION,
    CUDNN_NUMERICAL_NOTE_FFT,
    CUDNN_NUMERICAL_NOTE_NONDETERMINISTIC,
    CUDNN_NUMERICAL_NOTE_WINOGRAD,
    CUDNN_NUMERICAL_NOTE_WINOGRAD_TILE_4x4,
    CUDNN_NUMERICAL_NOTE_WINOGRAD_TILE_6x6,
    CUDNN_NUMERICAL_NOTE_WINOGRAD_TILE_13x13,
    CUDNN_NUMERICAL_NOTE_TYPE_COUNT,
```



```
} cudnnBackendNumericalNote_t;
```

### 9.1.1.11. `cudnnBackendTensorReordering_t`

`cudnnBackendTensorReordering_t` is an enumerated type that indicates tensor reordering as a property of the tensor descriptor. Users can get and set this property in a `CUDNN_BACKEND_TENSOR_DESCRIPTOR` via `cudnnBackendSetAttribute()` and `cudnnBackendGetAttribute()` functions.

```
typedef enum {
    CUDNN_TENSOR_REORDERING_NONE      = 0,
    CUDNN_TENSOR_REORDERING_INT8x32  = 1,
    CUDNN_TENSOR_REORDERING_F16x16   = 2,
} cudnnBackendTensorReordering_t;
```

### 9.1.1.12. `cudnnBnFinalizeStatsMode_t`

`cudnnBnFinalizeStatsMode_t` is an enumerated type that exposes the different mathematical operation modes that converts batchnorm statistics and the trained scale and bias to the equivalent scale and bias to be applied in the next normalization stage for inference and training use cases.

```
typedef enum {
    CUDNN_BN_FINALIZE_STATISTICS_TRAINING = 0,
    CUDNN_BN_FINALIZE_STATISTICS_INFERENCE = 1,
} cudnnBnFinalizeStatsMode_t;
```

Table 52. BN Statistics for `cudnnBnFinalizeStatsMode_t`

BN Statistics Mode	Description
<code>CUDNN_BN_FINALIZE_STATISTICS_TRAINING</code>	<p>Computes the equivalent scale and bias from <code>ySum</code>, <code>ySqSum</code> and learned <code>scale</code>, <code>bias</code>.</p> <p>Optionally, update running statistics and generate saved stats for interoperability with <code>cudnnBatchNormalizationBackward()</code>, <code>cudnnBatchNormalizationBackwardEx()</code>, or <code>cudnnNormalizationBackward()</code>.</p>
<code>CUDNN_BN_FINALIZE_STATISTICS_INFERENCE</code>	<p>Computes the equivalent scale and bias from the learned running statistics and the learned <code>scale</code>, <code>bias</code>.</p>

### 9.1.1.13. `cudnnFraction_t`

`cudnnFraction_t` is a structure that allows a user to define `int64_t` fractions.

```
typedef struct cudnnFractionStruct {
    int64_t numerator;
    int64_t denominator;
} cudnnFraction_t;
```

### 9.1.1.14. `cudnnGenStatsMode_t`

`cudnnGenStatsMode_t` is an enumerated type to indicate the statistics mode in the backend statistics generation operation.

## Values

### CUDNN\_GENSTATS\_SUM\_SQSUM

In this mode, the sum and sum of squares of the input tensor along the specified dimensions are computed and written out. The reduction dimensions currently supported are limited per channel, however additional support may be added upon request.

### 9.1.1.15. cudnnPaddingMode\_t

`cudnnPaddingMode_t` is an enumerated type to indicate the padding mode in the backend resample operations.

```
typedef enum {
    CUDNN_ZERO_PAD      = 0,
    CUDNN_NEG_INF_PAD   = 1,
    CUDNN_EDGE_VAL_PAD  = 2,
} cudnnPaddingMode_t;
```

### 9.1.1.16. cudnnPointwiseMode\_t

`cudnnPointwiseMode_t` is an enumerated type to indicate the intended pointwise math operation in the backend pointwise operation descriptor.

## Values

### CUDNN\_POINTWISE\_ADD

In this mode, a pointwise addition between two tensors is computed.

### CUDNN\_POINTWISE\_ADD\_SQUARE

In this mode, a pointwise addition between the first tensor and the square of the second tensor is computed.

### CUDNN\_POINTWISE\_DIV

In this mode, a pointwise true division of the first tensor by second tensor is computed.

### CUDNN\_POINTWISE\_MAX

In this mode, a pointwise maximum is taken between two tensors.

### CUDNN\_POINTWISE\_MIN

In this mode, a pointwise minimum is taken between two tensors.

### CUDNN\_POINTWISE\_MOD

In this mode, a pointwise floating-point remainder of the first tensor's division by the second tensor is computed.

### CUDNN\_POINTWISE\_MUL

In this mode, a pointwise multiplication between two tensors is computed.

### CUDNN\_POINTWISE\_POW

In this mode, a pointwise value from the first tensor to the power of the second tensor is computed.

### CUDNN\_POINTWISE\_SUB

In this mode, a pointwise subtraction between two tensors is computed.

**CUDNN\_POINTWISE\_ABS**

In this mode, a pointwise absolute value of the input tensor is computed.

**CUDNN\_POINTWISE\_CEIL**

In this mode, a pointwise ceiling of the input tensor is computed.

**CUDNN\_POINTWISE\_COS**

In this mode, a pointwise trigonometric cosine of the input tensor is computed.

**CUDNN\_POINTWISE\_EXP**

In this mode, a pointwise exponential of the input tensor is computed.

**CUDNN\_POINTWISE\_FLOOR**

In this mode, a pointwise floor of the input tensor is computed.

**CUDNN\_POINTWISE\_LOG**

In this mode, a pointwise natural logarithm of the input tensor is computed.

**CUDNN\_POINTWISE\_NEG**

In this mode, a pointwise numerical negative of the input tensor is computed.

**CUDNN\_POINTWISE\_RSQRT**

In this mode, a pointwise reciprocal of the square root of the input tensor is computed.

**CUDNN\_POINTWISE\_SIN**

In this mode, a pointwise trigonometric sine of the input tensor is computed.

**CUDNN\_POINTWISE\_SQRT**

In this mode, a pointwise square root of the input tensor is computed.

**CUDNN\_POINTWISE\_TAN**

In this mode, a pointwise trigonometric tangent of the input tensor is computed.

**CUDNN\_POINTWISE\_ERF**

In this mode, a pointwise Error Function is computed.

**CUDNN\_POINTWISE\_IDENTITY**

In this mode, no computation is performed. As with other pointwise modes, this mode provides implicit conversions by specifying the data type of the input tensor as one type, and the data type of the output tensor as another.

**CUDNN\_POINTWISE\_RELU\_FWD**

In this mode, a pointwise rectified linear activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_TANH\_FWD**

In this mode, a pointwise tanh activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_SIGMOID\_FWD**

In this mode, a pointwise sigmoid activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_ELU\_FWD**

In this mode, a pointwise Exponential Linear Unit activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_GELU\_FWD**

In this mode, a pointwise Gaussian Error Linear Unit activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_SOFTPLUS\_FWD**

In this mode, a pointwise softplus activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_SWISH\_FWD**

In this mode, a pointwise swish activation function of the input tensor is computed.

**CUDNN\_POINTWISE\_GELU\_APPROX\_TANH\_FWD**

In this mode, a pointwise tanh approximation of the Gaussian Error Linear Unit activation function of the input tensor is computed. The tanh GELU approximation is computed as  $0.5x \left( 1 + \tanh \left[ \sqrt{\frac{2}{\pi}} \left( x + 0.044715x^3 \right) \right] \right)$

For more information, refer to the [GAUSSIAN ERROR LINEAR UNIT \(GELUS\)](#) paper.

**CUDNN\_POINTWISE\_RELU\_BWD**

In this mode, a pointwise first derivative of rectified linear activation of the input tensor is computed.

**CUDNN\_POINTWISE\_TANH\_BWD**

In this mode, a pointwise first derivative of tanh activation of the input tensor is computed.

**CUDNN\_POINTWISE\_SIGMOID\_BWD**

In this mode, a pointwise first derivative of sigmoid activation of the input tensor is computed.

**CUDNN\_POINTWISE\_ELU\_BWD**

In this mode, a pointwise first derivative of Exponential Linear Unit activation of the input tensor is computed.

**CUDNN\_POINTWISE\_GELU\_BWD**

In this mode, a pointwise first derivative of Gaussian Error Linear Unit activation of the input tensor is computed.

**CUDNN\_POINTWISE\_SOFTPLUS\_BWD**

In this mode, a pointwise first derivative of softplus activation of the input tensor is computed.

**CUDNN\_POINTWISE\_SWISH\_BWD**

In this mode, a pointwise first derivative of swish activation of the input tensor is computed.

**CUDNN\_POINTWISE\_GELU\_APPROX\_TANH\_BWD**

In this mode, a pointwise first derivative of the tanh approximation of the Gaussian Error Linear Unit activation of the input tensor is computed. This is computed as

$0.5 \left( 1 + \tanh \left( b \left( x + cx^3 \right) \right) + bx \operatorname{sech}^2 \left( b \left( cx^3 + x \right) \right) \left( 3cx^2 + 1 \right) \right) dx$  where  $b$  is  $\sqrt{\frac{2}{\pi}}$  and  $c$  is 0.044715.

**CUDNN\_POINTWISE\_CMP\_EQ**

In this mode, a pointwise truth value of the first tensor equal to the second tensor is computed.

**CUDNN\_POINTWISE\_CMP\_NEQ**

In this mode, a pointwise truth value of the first tensor not equal to the second tensor is computed.

**CUDNN\_POINTWISE\_CMP\_GT**

In this mode, a pointwise truth value of the first tensor greater than the second tensor is computed.

**CUDNN\_POINTWISE\_CMP\_GE**

In this mode, a pointwise truth value of the first tensor greater than equal to the second tensor is computed.

**CUDNN\_POINTWISE\_CMP\_LT**

In this mode, a pointwise truth value of the first tensor less than the second tensor is computed.

**CUDNN\_POINTWISE\_CMP\_LE**

In this mode, a pointwise truth value of the first tensor less than equal to the second tensor is computed.

**CUDNN\_POINTWISE\_LOGICAL\_AND**

In this mode, a pointwise truth value of the first tensor logical AND second tensor is computed.

**CUDNN\_POINTWISE\_LOGICAL\_OR**

In this mode, a pointwise truth value of the first tensor logical OR second tensor is computed.

**CUDNN\_POINTWISE\_LOGICAL\_NOT**

In this mode, a pointwise truth value of input tensor's logical NOT is computed.

**CUDNN\_POINTWISE\_GEN\_INDEX**

In this mode, a pointwise index value of the input tensor is generated along a given axis.

**CUDNN\_POINTWISE\_BINARY\_SELECT**

In this mode, a pointwise value is selected amongst two input tensors based on a given predicate tensor.

**CUDNN\_POINTWISE\_RECIRPOCAL**

In this mode, a pointwise reciprocal of the input tensor is computed. In other words, for every element  $x$  in the input tensor,  $1/x$  is computed.

### 9.1.1.17. `cudaResampleMode_t`

`cudaResampleMode_t` is an enumerated type to indicate the resample mode in the backend resample operations.

```
typedef enum {
    CUDNN_RESAMPLE_NEAREST           = 0,
    CUDNN_RESAMPLE_BILINEAR          = 1,
    CUDNN_RESAMPLE_AVGPOOL           = 2,
    CUDNN_RESAMPLE_AVGPOOL_INCLUDE_PADDING = 2,
    CUDNN_RESAMPLE_AVGPOOL_EXCLUDE_PADDING = 4,
    CUDNN_RESAMPLE_MAXPOOL           = 3,
} cudaResampleMode_t;
```

### 9.1.1.18. `cudaRngDistribution_t`

`cudaRngDistribution_t` is an enumerated type to indicate the distribution to be used in the backend Rng (random number generator) operation.

```
typedef enum {
    CUDNN_RNG_DISTRIBUTION_BERNOULLI,
```

```
CUDNN_RNG_DISTRIBUTION_UNIFORM,
CUDNN_RNG_DISTRIBUTION_NORMAL,
} cudnnRngDistribution_t;
```

## Values

### CUDNN\_RNG\_DISTRIBUTION\_BERNOULLI

In this mode, the bernoulli distribution is used for the random number generation. The attribute `CUDNN_ATTR_RNG_BERNOULLI_DIST_PROBABILITY` can be used to specify the probability of generating 1's.

### CUDNN\_RNG\_DISTRIBUTION\_UNIFORM

In this mode, the normal distribution is used for the random number generation. The attribute `CUDNN_ATTR_RNG_NORMAL_DIST_MEAN` and `CUDNN_ATTR_RNG_NORMAL_DIST_STANDARD_DEVIATION` can be used to specify the mean and standard deviation of the random number generator.

## 9.1.1.19. `cudnnSignalMode_t`

`cudnnSignalMode_t` is an enumerated type to indicate the signaling mode in the backend signal operation.

```
typedef enum {
    CUDNN_SIGNAL_SET = 0,
    CUDNN_SIGNAL_WAIT = 1,
} cudnnSignalMode_t;
```

## Values

### CUDNN\_SIGNAL\_SET

In this mode, the flag variable is updated with the provided signal value atomically.

### CUDNN\_SIGNAL\_WAIT

In this mode, the operation blocks until the flag variable keeps comparing equal to the provided signal value.

## 9.1.2. Data Types Found In `cudnn_backend.h`

These are the data types found in `cudnn_backend.h`.

### 9.1.2.1. `cudnnBackendDescriptor_t`

`cudnnBackendDescriptor_t` is a typedef void pointer to one of many opaque descriptor structures. The type of structure that it points to is determined by the argument when allocating the memory for the opaque structure using [cudnnBackendCreateDescriptor\(\)](#).

Attributes of a descriptor can be set using [cudnnBackendSetAttribute\(\)](#). After all required attributes of a descriptor are set, the descriptor can be finalized by [cudnnBackendFinalize\(\)](#). From a finalized descriptor, one can query its queryable attributes using [cudnnBackendGetAttribute\(\)](#). Finally, the memory allocated for a descriptor can be freed using [cudnnBackendDestroyDescriptor\(\)](#).

## 9.2. API Functions

These are the API functions in the cuDNN Backend API.

### 9.2.1. `cudaBackendCreateDescriptor()`

This function allocates memory in the `descriptor` for a given `descriptorType` type and at the location pointed by the `descriptor`.

```
cudaStatus_t cudaBackendCreateDescriptor(cudaBackendDescriptorType_t
descriptorType, cudaBackendDescriptor_t *descriptor)
```



Note: The `cudaBackendDescriptor_t` is a pointer to void \*.

#### Parameters

##### **descriptorType**

*Input.* One among the enumerated `cudaBackendDescriptorType_t`.

##### **descriptor**

*Input.* Pointer to an instance of `cudaBackendDescriptor_t` to be created.

#### Returns

##### **CUDA\_STATUS\_SUCCESS**

The creation was successful.

##### **CUDA\_STATUS\_NOT\_SUPPORTED**

Creating a descriptor of a given type is not supported.

##### **CUDA\_STATUS\_ALLOC\_FAILED**

The memory allocation failed.

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

### 9.2.2. `cudaBackendDestroyDescriptor()`

This function destroys instances of `cudaBackendDescriptor_t` that were previously created using `cudaBackendCreateDescriptor()`.

```
cudaStatus_t cudaBackendDestroyDescriptor(cudaBackendDescriptor_t descriptor)
```

#### Parameters

##### **descriptor**

*Input.* Instance of `cudaBackendDescriptor_t` previously created by `cudaBackendCreateDescriptor()`.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The memory was destroyed successfully.

### **CUDNN\_STATUS\_ALLOC\_FAILED**

The destruction of memory failed.

### **Undefined Behavior**

The descriptor was altered between the `Create` and `Destroy` Descriptor.

### **Undefined**

The value pointed by the `descriptor` will be `Undefined` after the memory is free and done.

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.2.3. `cudaBackendExecute()`

This function executes the given `Engine Configuration Plan` on the `VariantPack` and the finalized `ExecutionPlan` on the data. The data and the working space are encapsulated in the `VariantPack`.

```
cudaStatus_t cudaBackendExecute(cudaHandle_t handle, cudaBackendDescriptor_t
    executionPlan, cudaBackendDescriptor_t variantPack)
```

## Parameters

`executionPlan`

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

`variantPack`

*Input.* Pointer to the finalized `VariantPack` consisting of:

- ▶ Data pointer for each non-virtual pointer of the operation set in the execution plan.
- ▶ Pointer to user-allocated workspace in global memory at least as large as the size queried from `CUDNN_BACKEND_`.

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The `ExecutionPlan` was executed successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

An incorrect or inconsistent value is encountered. Some examples:

- ▶ A required data pointer is invalid.

### **CUDNN\_STATUS\_INTERNAL\_ERROR**

Some internal errors were encountered.



**CUDNN\_STATUS\_EXECUTION\_FAILED**

An error was encountered executing the plan with the variant pack.

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.2.4. `cudaBackendFinalize()`

This function finalizes the memory pointed to by the `descriptor`. The type of finalization is done depending on the `descriptorType` argument with which the `descriptor` was created using [cudaBackendCreateDescriptor\(\)](#) or initialized using [cudaBackendInitialize\(\)](#).

```
cudaStatus_t cudaBackendFinalize(cudaBackendDescriptor descriptor)
```

`cudaBackendFinalize()` also checks all the attributes set between the create/initialization and finalize phase. If successful, `cudaBackendFinalize()` returns `CUDNN_STATUS_SUCCESS` and the finalized state of the `descriptor` is set to `true`. In this state, setting attributes using [cudaBackendSetAttribute\(\)](#) is not allowed. Getting attributes using [cudaBackendGetAttribute\(\)](#) is only allowed when the finalized state of the `descriptor` is `true`.

### Parameters

**descriptor**

*Input.* Instance of [cudaBackendDescriptor\\_t](#) to finalize.

### Returns

**CUDNN\_STATUS\_SUCCESS**

The `descriptor` was finalized successfully.

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid `descriptor` attribute values or combination thereof is encountered.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

Descriptor attribute values or combinations therefore not supported by the current version of cuDNN are encountered.

**CUDNN\_STATUS\_INTERNAL\_ERROR**

Some internal errors are encountered.

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.2.5. `cudaBackendGetAttribute()`

This function retrieves the value(s) of an attribute of a `descriptor`. `attributeName` is the name of the attribute whose value is requested. The `attributeType` is the type of attribute. `requestedElementCount` is the number of elements to be potentially retrieved. The number of elements for the requested attribute is stored in `elementCount`. The retrieved values are stored in `arrayOfElements`. When the attribute is expected to have a single value, `arrayOfElements` can be pointer to the output value. This function

will return `CUDNN_STATUS_NOT_INITIALIZED` if the *descriptor* has not been successfully finalized using [`cudaBackendFinalize\(\)`](#).

```

cudaStatus_t cudaBackendGetAttribute(
    cudaBackendDescriptor_t descriptor,
    cudaBackendAttributeName_t attributeName,
    cudaBackendAttributeType_t attributeType,
    int64_t requestedElementCount,
    int64_t *elementCount,
    void *arrayOfElements);

```

## Parameters

### **descriptor**

*Input.* Instance of [`cudaBackendDescriptor\_t`](#) whose attribute the user wants to retrieve.

### **attributeName**

*Input.* The name of the attribute being get from the on the descriptor.

### **attributeType**

*Input.* The type of attribute.

### **requestedElementCount**

*Input.* Number of elements to output to `arrayOfElements`.

### **elementCount**

*Input.* Output pointer for the number of elements the `descriptor` attribute has. Note that `cudaBackendGetAttribute()` will only write the least of this and `requestedElementCount` elements to `arrayOfElements`.

### **arrayOfElements**

*Input.* Array of elements of the datatype of the `attributeType`. The datatype of the `attributeType` is listed in the mapping table of [`cudaBackendAttributeType\_t`](#).

## Returns

### **CUDNN\_STATUS\_SUCCESS**

The `attributeName` was given to the descriptor successfully.

### **CUDNN\_STATUS\_BAD\_PARAM**

One or more invalid or inconsistent argument values were encountered. Some examples:

- ▶ `attributeName` is not a valid attribute for the descriptor.
- ▶ `attributeType` is not one of the valid types for the attribute.

### **CUDNN\_STATUS\_NOT\_INITIALIZED**

The descriptor has not been successfully finalized using [`cudaBackendFinalize\(\)`](#).

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.2.6. `cudaBackendInitialize()`

This function repurposes a pre-allocated memory pointed to by a `descriptor` of size `sizeInByte` to a backend descriptor of type `descriptorType`. The finalized state of the descriptor is set to `false`.

```
cudaStatus_t cudaBackendInitialize(cudaBackendDescriptor_t descriptor,
    cudaBackendDescriptorType_t descriptorType, size_t sizeInBytes)
```

### Parameters

#### **descriptor**

*Input.* Instance of `cudaBackendDescriptor_t` to be initialized.

#### **descriptorType**

*Input.* Enumerated value for the type of cuDNN backend descriptor.

#### **sizeInBytes**

*Input.* Size of memory pointed to by descriptor.

### Returns

#### **CUDA\_STATUS\_SUCCESS**

The memory was initialized successfully.

#### **CUDA\_STATUS\_BAD\_PARAM**

An invalid or inconsistent argument value is encountered. For example:

- ▶ `descriptor` is a `nullptr`
- ▶ `sizeInBytes` is less than the size required by the `descriptor` type

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.2.7. `cudaBackendSetAttribute()`

This function sets an attribute of a descriptor to value(s) provided as a pointer. `descriptor` is the descriptor to be set. `attributeName` is the name of the attribute to be set. `attributeType` is the type of attribute. The value to which the attribute is set, is pointed by the `arrayOfElements`. The number of elements is given by `elementCount`. This function will return `CUDA_STATUS_NOT_INITIALIZED` if the descriptor is already successfully finalized using `cudaBackendFinalize()`.

```
cudaStatus_t cudaBackendSetAttribute(
    cudaBackendDescriptor_t descriptor,
    cudaBackendAttributeName_t attributeName,
    cudaBackendAttributeType_t attributeType,
    int64_t elementCount,
    void *arrayOfElements);
```

### Parameters

#### **descriptor**

*Input.* Instance of `cudaBackendDescriptor_t` whose attribute is being set.

**attributeName**

*Input.* The name of the attribute being set on the `descriptor`.

**attributeType**

*Input.* The type of attribute.

**elementCount**

*Input.* Number of elements being set.

**arrayOfElements**

*Input.* The starting location for an array from where to read the values from. The elements of the array are expected to be of the datatype of the `attributeType`. The datatype of the `attributeType` is listed in the mapping table of [cudnnBackendAttributeType\\_t](#).

## Returns

**CUDNN\_STATUS\_SUCCESS**

The `attributeName` was set to the `descriptor`.

**CUDNN\_STATUS\_NOT\_INITIALIZED**

The backend `descriptor` pointed to by the `descriptor` is already in the finalized state.

**CUDNN\_STATUS\_BAD\_PARAM**

The function is called with arguments that correspond to invalid values. Some possible causes are:

- ▶ `attributeName` is not a settable attribute of `descriptor`
- ▶ `attributeType` is incorrect for this `attributeName`.
- ▶ `elemCount` value is unexpected.
- ▶ `arrayOfElements` contains values invalid for the `attributeType`.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The value(s) to which the attributes are being set is not supported by the current version of cuDNN.

Additional return values depend on the arguments used as explained in the [cuDNN Backend API](#).

## 9.3. Backend Descriptor Types

This section enumerates all valid attributes of various descriptors.

### 9.3.1. CUDNN\_BACKEND\_CONVOLUTION\_DESCRIPTOR

Created with

`cudnnBackendCreateDescriptor(CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR, &desc);`  
the cuDNN backend convolution descriptor specifies the parameters for a convolution

operator for both forward and backward propagation: compute data type, convolution mode, filter dilation and stride, and padding on both sides.

## Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type `cudaDnnBackendAttributeName_t` with prefix `CUDNN_ATTR_CONVOLUTION_`:

### **CUDNN\_ATTR\_CONVOLUTION\_COMP\_TYPE**

The compute type of the convolution operator.

- ▶ `CUDNN_TYPE_DATA_TYPE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_CONV\_MODE**

Convolution or cross-correlation mode.

- ▶ `CUDNN_TYPE_CONVOLUTION_MODE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_DILATIONS**

Filter dilation.

- ▶ `CUDNN_TYPE_INT64`; one or more, but at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_FILTER\_STRIDES**

Filter stride.

- ▶ `CUDNN_TYPE_INT64`; one or more, but at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_PRE\_PADDINGS**

Padding at the beginning of each spatial dimension.

- ▶ `CUDNN_TYPE_INT64`; one or more, but at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_POST\_PADDINGS**

Padding at the end of each spatial dimension.

- ▶ `CUDNN_TYPE_INT64`; one or more, but at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_CONVOLUTION\_SPATIAL\_DIMS**

The number of spatial dimensions in the convolution.

- ▶ `CUDNN_TYPE_INT64`, one element.
- ▶ Required attribute.

## Finalization

`cudaBackendFinalize()` with a `CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR` can have the following return values:

### **CUDNN\_STATUS\_BAD\_PARAM**

An `elemCount` argument for setting `CUDNN_ATTR_CONVOLUTION_DILATIONS`, `CUDNN_ATTR_CONVOLUTION_FILTER_STRIDES`, `CUDNN_ATTR_CONVOLUTION_PRE_PADDINGS`, and `CUDNN_ATTR_CONVOLUTION_POST_PADDINGS` is not equal to the value set for `CUDNN_ATTR_CONVOLUTION_SPATIAL_DIMS`.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.2. CUDNN\_BACKEND\_ENGINE\_DESCRIPTOR

Created with descriptor type value `CUDNN_BACKEND_ENGINE_DESCRIPTOR`, cuDNN backend engine descriptor describes an engine to compute an operation graph. An engine is a grouping of kernels with similar compute and numerical attributes.

### Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type

`cudaBackendAttributeName_t` with prefix `CUDNN_ATTR_ENGINE_`:

#### **CUDNN\_ATTR\_ENGINE\_OPERATION\_GRAPH**

The operation graph to compute.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_OPERATIONGRAPH_DESCRIPTOR`.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_ENGINE\_GLOBAL\_INDEX**

The index for the engine.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Valid values are between 0 and `CUDNN_ATTR_OPERATIONGRAPH_ENGINE_GLOBAL_COUNT-1`.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_ENGINE\_KNOB\_INFO**

The descriptors of performance knobs of the engine.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_KNOB_INFO_DESCRIPTOR`.
- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_ENGINE\_NUMERICAL\_NOTE**

The numerical attributes of the engine.

- ▶ `CUDNN_TYPE_NUMERICAL_NOTE`; zero or more elements.
- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_ENGINE\_LAYOUT\_INFO**

The preferred tensor layouts of the engine.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_LAYOUT_INFO_DESCRIPTOR`.
- ▶ Read-only attribute.

## Finalization

#### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

#### **CUDNN\_STATUS\_NOT\_SUPPORTED**

The descriptor attribute set is not supported by the current version of cuDNN. Some examples include:

- ▶ The value of `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` is not in a valid range.

#### **CUDNN\_STATUS\_BAD\_PARAM**

The descriptor attribute set is inconsistent or in an unexpected state. Some examples include:

- ▶ The operation graph descriptor set is not already finalized.

## 9.3.3. **CUDNN\_BACKEND\_ENGINECFG\_DESCRIPTOR**

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_ENGINECFG_DESCRIPTOR, &desc)`; the cuDNN backend engine configuration descriptor consists of an engine descriptor and an array of knob choice descriptors. Users can query from engine config information about intermediates: computational intermediate results that can be reused between executions.

### Attributes

#### **CUDNN\_ATTR\_ENGINECFG\_ENGINE**

The backend engine.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`: one element, a backend descriptor of type `CUDNN_BACKEND_ENGINE_DESCRIPTOR`.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_ENGINECFG\_KNOB\_CHOICES**

The engine tuning knobs and choices.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`: zero or more elements, backend descriptors of type `CUDNN_BACKEND_KNOB_CHOICE_DESCRIPTOR`.

**CUDNN\_ATTR\_ENGINECFG\_INTERMEDIATE\_INFO**

Information of the computational intermediate of this engine config.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR: one element, a backend descriptor of type CUDNN\_BACKEND\_INTERMEDIATE\_INFO\_DESCRIPTOR.
- ▶ Read-only attribute.
- ▶ Currently unsupported. Placeholder for future implementation.

## Finalization

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

The descriptor attribute set is not supported by the current version of cuDNN. Some examples include:

- ▶ The value knob.

## 9.3.4. CUDNN\_BACKEND\_ENGINEHEUR\_DESCRIPTOR

Created with

```
cudaBackendCreateDescriptor(CUDNN_BACKEND_ENGINEHEUR_DESCRIPTOR, &desc);
```

the cuDNN backend engine heuristics descriptor allows users to obtain for an operation graph engine configuration descriptors ranked by performance according to cuDNN's heuristics.

### Attributes

**CUDNN\_ATTR\_ENGINEHEUR\_OPERATION\_GRAPH**

The operation graph for which heuristics result in a query.

**CUDNN\_TYPE\_BACKEND\_DESCRIPTOR**

One element.

- ▶ Required attribute.

**CUDNN\_ATTR\_ENGINEHEUR\_MODE**

The heuristic mode to query the result.

- ▶ CUDNN\_TYPE\_HEUR\_MODE; one element.
- ▶ Required attribute.

**CUDNN\_ATTR\_ENGINEHEUR\_RESULTS**

The result of the heuristics query.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; zero or more elements of descriptor type CUDNN\_BACKEND\_ENGINECFG\_DESCRIPTOR.
- ▶ Get-only attribute.



## Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is a cuDNN backend engine heuristics descriptor:

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.5. CUDNN\_BACKEND\_EXECUTION\_PLAN\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR, &desc)`; the cuDNN backend execution plan descriptor allows the user to specify an execution plan, consists of a cuDNN handle, an engine configuration, and optionally an array of intermediates to compute.

## Attributes

### **CUDNN\_ATTR\_EXECUTION\_PLAN\_HANDLE**

A cuDNN handle.

- ▶ `CUDNN_TYPE_HANDLE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_EXECUTION\_PLAN\_ENGINE\_CONFIG**

An engine configuration to execute.

- ▶ `CUDNN_BACKEND_ENGINECFG_DESCRIPTOR`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_EXECUTION\_PLAN\_RUN\_ONLY\_INTERMEDIATE\_UIDS**

Unique identifiers of intermediates to compute.

- ▶ `CUDNN_TYPE_INT64`; zero or more elements.
- ▶ Optional attribute. If set, the execution plan will only compute the specified intermediate and not any of the output tensors on the operation graph in the engine configuration.

### **CUDNN\_ATTR\_EXECUTION\_PLAN\_COMPUTED\_INTERMEDIATE\_UIDS**

Unique identifiers of precomputed intermediates.

- ▶ `CUDNN_TYPE_INT64`; zero or more elements.
- ▶ Optional attribute. If set, the plan will expect and use pointers for each intermediate in the variant pack descriptor during execution.
- ▶ Not supported currently: placeholder for future implementation.

### **CUDNN\_ATTR\_EXECUTION\_PLAN\_WORKSPACE\_SIZE**

The size of the workspace buffer required to execute this plan.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_EXECUTION\_PLAN\_JSON\_REPRESENTATION**

The JSON representation of the serialized execution plan. Serialization and deserialization can be done by getting and setting this attribute, respectively.

- ▶ `CUDNN_TYPE_CHAR`; many elements, the same amount as the size of a null-terminated string of the json representation of the execution plan.

### Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is a cuDNN backend execution plan descriptor:

#### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.6. CUDNN\_BACKEND\_INTERMEDIATE\_INFO\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_INTERMEDIATE_INFO_DESCRIPTOR, &desc)`; the cuDNN backend intermediate descriptor is a read-only descriptor that contains information about an execution intermediate. An execution intermediate is some intermediate computation for an engine config in device memory that can be reused between plan execution to amortize the kernel. Each intermediate is identified by a unique ID. Users can query for the device memory size of the intermediate. An intermediate can depend on the data of one or more tensors identified by the tensor UIDs or one more attribute of the operation graph.

This is a read-only descriptor. Users cannot set the descriptor attributes or finalize the descriptor. User query for a finalized descriptor from an engine config descriptor.

### Attributes

#### **CUDNN\_ATTR\_INTERMEDIATE\_INFO\_UNIQUE\_ID**

A unique identifier of the intermediate.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_INTERMEDIATE\_INFO\_SIZE**

The required device memory size for the intermediate.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_INTERMEDIATE\_INFO\_DEPENDENT\_DATA\_UIDS**

UID of tensors on which the intermediate depends.

- ▶ `CUDNN_TYPE_INT64`; zero or more elements.

- ▶ Read-only attribute.

#### **CUDNN\_ATTR\_INTERMEDIATE\_INFO\_DEPENDENT\_ATTRIBUTES**

Placeholder for future implementation.

### Finalization

User does not finalize this descriptor. `cudaBackendFinalize(desc)` with a backend intermediate descriptor returns `CUDNN_STATUS_NOT_SUPPORTED`.

## 9.3.7. CUDNN\_BACKEND\_KNOB\_CHOICE\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_KNOB_CHOICE_DESCRIPTOR, &desc);` the cuDNN backend knob choice descriptor consists of the type of knobs to be set and the value to which the knob is set.

### Attributes

#### **CUDNN\_ATTR\_KNOB\_CHOICE\_KNOB\_TYPE**

The type of knobs to be set.

- ▶ `CUDNN_TYPE_KNOB_TYPE`: one element.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_KNOB\_CHOICE\_KNOB\_VALUE**

- ▶ `CUDNN_TYPE_INT64`: one element.
- ▶ Required attribute.

### Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is a cuDNN backend knob choice descriptor:

#### **CUDNN\_STATUS\_SUCCESS**

The knob choice descriptor was finalized successfully.

## 9.3.8. CUDNN\_BACKEND\_KNOB\_INFO\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_INFO_DESCRIPTOR, &desc);` the cuDNN backend knob info descriptor consists of the type and valid value range of an engine performance knob. Valid value range is given in terms of minimum, maximum, and stride of valid values. This is a purely informative descriptor type. Setting descriptor attributes is not supported. User obtains an array of finalized descriptors, one for each knob type, from a finalized backend descriptor.

## Attributes

### **CUDNN\_ATTR\_KNOB\_INFO\_TYPE**

The type of the performance knob.

- ▶ CUDNN\_TYPE\_KNOB\_TYPE: one element.
- ▶ Read-only attribute.

### **CUDNN\_ATTR\_KNOB\_INFO\_MAXIMUM\_VALUE**

The smallest valid value choice value for this knob.

- ▶ CUDNN\_TYPE\_INT64: one element.
- ▶ Read-only attribute.

### **CUDNN\_ATTR\_KNOB\_INFO\_MINIMUM\_VALUE**

The largest valid choice value for this knob.

- ▶ CUDNN\_TYPE\_INT64: one element.
- ▶ Read-only attribute.

### **CUDNN\_ATTR\_KNOB\_INFO\_STRIDE**

The stride of valid choice values for this knob.

- ▶ CUDNN\_TYPE\_INT64: one element.
- ▶ Read-only attribute.

## Finalization

This descriptor is read-only; it is retrieved and finalized from a cuDNN backend engine configuration descriptor. Users cannot set or finalize.

## 9.3.9. CUDNN\_BACKEND\_LAYOUT\_INFO\_DESCRIPTOR

Created with descriptor type value CUDNN\_BACKEND\_LAYOUT\_INFO\_DESCRIPTOR, cuDNN backend layout info descriptor provides information on the preferred layout for a tensor.

## Attributes

### **CUDNN\_ATTR\_LAYOUT\_INFO\_TENSOR\_UID**

The UID of the tensor.

- ▶ CUDNN\_TYPE\_INT64; one element.
- ▶ Read-only attribute.

### **CUDNN\_ATTR\_LAYOUT\_INFO\_TYPES**

The preferred layout of the tensor.

- ▶ CUDNN\_TYPE\_LAYOUT\_TYPE: zero or more element [`cudaBackendLayoutType\_t`](#).
- ▶ Read-only attribute.

## Finalization

This descriptor is read-only; it is retrieved and finalized from a cuDNN backend engine configuration descriptor. Users cannot set its attribute or finalize it.

### 9.3.10. CUDNN\_BACKEND\_MATMUL\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_MATMUL_DESCRIPTOR, &desc)`; the cuDNN backend `matmul` descriptor specifies any metadata needed for the `matmul` operation.

## Attributes

### CUDNN\_ATTR\_MATMUL\_COMP\_TYPE

The compute precision used for the `matmul` operation.

- ▶ `CUDNN_TYPE_DATA_TYPE`; one element.
- ▶ Required attribute.

## Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is a cuDNN backend `matmul` descriptor:

### CUDNN\_STATUS\_SUCCESS

The descriptor was finalized successfully.

### 9.3.11. CUDNN\_BACKEND\_OPERATION\_CONCAT\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_CONCAT_DESCRIPTOR, &desc)`; the cuDNN backend concatenation operation descriptor specifies an operation node for concatenating a given vector of tensors along a given concatenation axis.

This operation also supports an in-place mode, where one of the input tensors is already assumed to be at the correct location in the output tensor, that is, they share the same device buffer.

## Attributes

Attributes of a cuDNN backend `concat` operation descriptor are values of enumeration type `cudaBackendAttributeName_t` with prefix `CUDNN_ATTR_OPERATION_CONCAT_`:

### CUDNN\_ATTR\_OPERATION\_CONCAT\_AXIS

The dimension which tensors are being concatenated over.

- ▶ Type: `CUDNN_TYPE_INT64`
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONCAT\_INPUT\_DESCS**

A vector of input tensor descriptors, which are concatenated in the same order as provided in this vector.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one or more elements of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONCAT\_INPLACE\_INDEX**

The index of input tensor in the vector of input tensor descriptors that is already present in-place in the output tensor.

- ▶ Type: CUDNN\_TYPE\_INT64
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_CONCAT\_OUTPUT\_DESC**

The output tensor descriptor for the result from concatenation of input tensors.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

## Finalization

[`cudaBackendFinalize\(\)`](#) with a `CUDNN_BACKEND_OPERATION_CONCAT_DESCRIPTOR()` can have the following return values:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes:

- ▶ The tensors involved in the operation should have the same shape in all dimensions except the dimension that they are being concatenated over.
- ▶ The output tensor shape in the concatenating dimension should equal the sum of tensor shape of all input tensors in that same dimension.
- ▶ Concatenation axis should be a valid tensor dimension.
- ▶ If provided, the in-place input tensor index should be a valid index in the vector of input tensor descriptors.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.12. CUDNN\_BACKEND\_OPERATION\_CONVOLUTION\_BACKWARD

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_DATA_DESCRIPTOR &desc);` the cuDNN backend convolution backward data operation descriptor specifies an operation node for convolution backward data to compute the gradient of input data  $dx$  with filter tensor  $w$  and gradient of response  $dy$  with output  $\alpha$  scaling and residue

add with  $\beta$  scaling. That is, the equation  $dx = \alpha (w^*dy) + \beta dx$ , where  $*$  denotes the convolution backward data operator.

## Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type `cudaDnnBackendAttributeName_t` with prefix

`CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_DATA_`:

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_ALPHA**

The alpha value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_BETA**

The beta value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_CONV\_DESC**

The convolution operator descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_W**

The convolution filter tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_DX**

The image gradient tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_DATA\_DY**

The response gradient tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

## Finalization

In finalizing the convolution operation, the tensor dimensions of the tensor  $DX$ ,  $W$ , and  $DY$  are bound based on the same interpretations as the  $X$ ,  $W$ , and  $Y$  tensor dimensions described in the [CUDNN\\_BACKEND\\_OPERATION\\_CONVOLUTION\\_FORWARD\\_DESCRIPTOR](#) section.

`cudaBackendFinalize()` with a

`CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_DATA_DESCRIPTOR()` can have the following return values:

### **CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible cause:

- ▶ The  $DX$ ,  $W$ , and  $DY$  tensors do not constitute a valid convolution operation under the convolution operator.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.13. CUDNN\_BACKEND\_OPERATION\_CONVOLUTION\_BACKWARD\_FILTER\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_FILTER_DESCRIPTOR &desc);` the cuDNN backend convolution backward filter operation descriptor specifies an operation node for convolution backward filter to compute the gradient of filter  $dw$  with image tensor  $x$  and gradient of response  $dy$  with output  $\alpha$  scaling and residue add with  $\beta$  scaling. That is, the equation:  $dw = \alpha(x \tilde{*} dy) + \beta dw$ , where  $\tilde{*}$  denotes the convolution backward filter operator.

## Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type `cudaBackendAttributeName_t` with prefix

`CUDNN_ATTR_OPERATION_CONVOLUTION_BWD_FILTER_:`

### **CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_ALPHA**

The alpha value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required attribute. Required to be set before finalization.

### **CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_BETA**

The beta value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required attribute. Required to be set before finalization.

### **CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_CONV\_DESC**

The convolution operator descriptor.



- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR`.
- ▶ Required attribute. Required to be set before finalization.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_DW**

The convolution filter tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute. Required to be set before finalization.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_X**

The image gradient tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute. Required to be set before finalization.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_BWD\_FILTER\_DY**

The response gradient tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute. Required to be set before finalization.

## Finalization

In finalizing the convolution operation, the tensor dimensions of the tensor `X`, `DW`, and `DY` are bound based on the same interpretations as the `X`, `W`, and `Y` tensor dimensions described in the [CUDNN\\_BACKEND\\_OPERATION\\_CONVOLUTION\\_FORWARD\\_DESCRIPTOR](#) section.

[`cudaBackendFinalize\(\)`](#) with a

`CUDNN_BACKEND_OPERATION_CONVOLUTION_BACKWARD_FILTER_DESCRIPTOR()` can have the following return values:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible cause:

- ▶ The `X`, `DW`, and `DY` tensors do not constitute a valid convolution operation under the convolution operator.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.14. CUDNN\_BACKEND\_OPERATION\_CONVOLUTION\_FORWARD\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR, &desc)`; the cuDNN backend convolution forward operation descriptor specifies an operation node for forward convolution to compute the response tensor `y` of image

tensor  $x$  convoluted with filter tensor  $w$  with output scaling  $\alpha$  and residual add with  $\beta$  scaling. That is, the equation  $y = \alpha(w*x) + \beta y$ , where  $*$  is the convolution operator in the forward direction.

## Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type `cudaBackendAttribute_t` with prefix

`CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_`:

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_ALPHA**

The alpha value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required to be set before finalization.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_BETA**

The beta value.

- ▶ `CUDNN_TYPE_FLOAT` or `CUDNN_TYPE_DOUBLE`; one or more elements.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_CONV\_DESC**

The convolution operator descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_W**

The convolution filter tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_X**

The image tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_CONVOLUTION\_FORWARD\_Y**

The response tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_CONVOLUTION\_SPATIAL\_DIMS**

The number of spatial dimensions in the convolution.

- ▶ CUDNN\_TYPE\_INT64, one element.
- ▶ Required attribute.

## Finalization

In finalizing the convolution operation, the tensor dimensions of the tensor  $x$ ,  $w$ , and  $y$  are bound based on the following interpretations:

The `CUDNN_ATTR_CONVOLUTION_SPATIAL_DIMS` attribute of `CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_CONV_DESC` is the number of spatial dimension of the convolution. The number of dimensions for tensor  $x$ ,  $w$ , and  $y$  must be larger than the number of spatial dimensions by 2 or 3 depending on how users choose to specify the convolution tensors.

If the number of tensor dimension is the number of spatial dimensions plus 2:

- ▶  $x$  tensor dimension and stride arrays are  $[N, G, \dots]$
- ▶  $w$  tensor dimension and stride arrays are  $[K, C, \dots]$
- ▶  $y$  tensor dimension and stride arrays are  $[N, G, \dots]$

where the ellipsis  $\dots$  are shorthand for spatial dimensions of each tensor,  $G$  is the number of convolution groups, and  $C$  and  $K$  are the number of input and output feature maps per group. In this interpretation, it is assumed that the memory layout for each group is packed. `cudaBackendFinalize()` asserts the tensors dimensions and strides are consistent with this interpretation or it returns `CUDNN_STATUS_BAD_PARAM`.

If the number of tensor dimension is the number of spatial dimensions plus 3:

- ▶  $x$  tensor dimension and stride arrays are  $[N, G, C, \dots]$
- ▶  $w$  tensor dimension and stride arrays are  $[G, K, C, \dots]$
- ▶  $y$  tensor dimension and stride arrays are  $[N, G, K, \dots]$

where the ellipsis  $\dots$  are shorthand for spatial dimensions of each tensor,  $G$  is the number of convolution groups, and  $C$  and  $K$  are the number of input and output feature maps per group. In this interpretation, users can specify an unpacked group stride. `cudaBackendFinalize()` asserts the tensors dimensions and strides are consistent with this interpretation or it returns `CUDNN_STATUS_BAD_PARAM`.

`cudaBackendFinalize()` with a

`CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR` can have the following return values:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible cause:

- ▶ The  $x$ ,  $w$ , and  $y$  tensors do not constitute a valid convolution operation under the convolution operator.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.15. CUDNN\_BACKEND\_OPERATION\_GEN\_STATS\_DESCRIPTOR

Represents an operation that will generate per-channel statistics. The specific statistics that will be generated depends on the `CUDNN_ATTR_OPERATION_GENSTATS_MODE` attribute in the descriptor. Currently, only `CUDNN_GENSTATS_SUM_SQSUM` is supported for the `CUDNN_ATTR_OPERATION_GENSTATS_MODE`. It will generate the sum and quadratic sum of per-channel elements of the input tensor  $x$ . The output dimension should be all 1 except the  $c$  dimension. Also, the  $c$  dimension of outputs should equal the  $c$  dimension of the input. This opaque struct can be created with `cudaBackendCreateDescriptor()` (`CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR`).

#### Attributes

**CUDNN\_ATTR\_OPERATION\_GENSTATS\_MODE**

Sets the `CUDNN_TYPE_GENSTATS_MODE` of the operation. This attribute is required.

**CUDNN\_ATTR\_OPERATION\_GENSTATS\_MATH\_PREC**

The math precision of the computation. This attribute is required.

**CUDNN\_ATTR\_OPERATION\_GENSTATS\_XDESC**

Sets the descriptor for the input tensor  $x$ . This attribute is required.

**CUDNN\_ATTR\_OPERATION\_GENSTATS\_SUMDESC**

Sets the descriptor for the output tensor `sum`. This attribute is required.

**CUDNN\_ATTR\_OPERATION\_GENSTATS\_SQSUMDESC**

Sets the descriptor for the output tensor `quadraticsum`. This attribute is required.

#### Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes are:

- ▶ The number of dimensions do not match between the input and output tensors.
- ▶ The input/output tensor dimensions do not agree with the above description.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.16. CUDNN\_BACKEND\_OPERATION\_MATMUL\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR, &desc);` the cuDNN backend `matmul` operation descriptor specifies an operation node for `matmul` to compute the matrix product  $C$  by multiplying matrix  $A$  and matrix  $B$ , as shown in the following equation:  $C = AB$

When using the `matmul` operation, the matrices are expected to be at least rank-2 tensors. The last two dimensions are expected to correspond to either M, K or N. All the preceding dimensions are interpreted as batch dimensions. If there are zero batch dimensions then the requirements are as follows:

Table 53. `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` for Zero Batch Dimensions

Case	Matrix A	Matrix B	Matrix C
Single matmul	$M \times K$	$K \times N$	$M \times N$

For a single batch dimension we have the following requirements:

Table 54. `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` for a Single Batch Dimension

Case	Matrix A	Matrix B	Matrix C
Single matmul	$1 \times M \times K$	$1 \times K \times N$	$1 \times M \times N$
Batch matmul	$B \times M \times K$	$B \times K \times N$	$B \times M \times N$
Broadcast A	$1 \times M \times K$	$B \times K \times N$	
Broadcast B	$B \times M \times K$	$1 \times K \times N$	

where:

- ▶ B indicates the batch size
- ▶ M is the number of rows of the matrix A
- ▶ K is the number of columns of the input matrix A (which is the same as the number of rows as the input matrix B)
- ▶ N is the number of columns of the input matrix B

If either the batch size of matrix A or B is set to 1, this indicates that the matrix will be broadcasted in the batch matmul. The resulting output matrix C will be a tensor of  $B \times M \times N$ .

The above broadcasting convention is extended to all the batch dimensions. Concretely, for tensors with three batch dimensions:

Table 55. `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` for a Three Batch Dimension

Case	Matrix A	Matrix B	Matrix C
Multiple batched matmul	$B1 \times 1 \times B3 \times M \times K$	$1 \times B2 \times B3 \times K \times N$	$B1 \times B2 \times B3 \times M \times N$

The functionality of having multiple batch dimensions allows you to have layouts where the batch is not packed at a single stride. This case is especially seen in multi-head attention.

The addressing of the matrix elements from a given tensor can be specified using strides in the tensor descriptor. The strides represent the spacing between elements for each tensor dimension. Considering a matrix tensor A ( $B \times M \times N$ ) with strides [BS, MS, NS], it indicates that the actual matrix element  $A[x, y, z]$  is found at  $(A\_base\_address + x * BS + y * MS + z * NS)$  from the linear memory space allocated for tensor A. With our current support, the innermost dimension must be packed, which requires either  $MS=1$  or  $NS=1$ . Otherwise, there are no other technical constraints with regard to how the strides can be specified in a tensor descriptor as it should follow the aforementioned addressing formula and the strides as specified by the user.

This representation provides support for some common usages, such as leading dimension and matrix transpose as we will explain through the following examples.

1. The most basic case is a fully packed row-major batch matrix, without any consideration of leading dimension or transpose. In this case,  $BS = M*N$ ,  $MS = N$  and  $NS = 1$ .
2. Matrix transpose can be achieved by exchanging the inner and outer dimensions using strides. Namely:
  - a). To specify a non-transposed matrix:  $BS = M*N$ ,  $MS = N$  and  $NS = 1$ .
  - b). To specify matrix transpose:  $BS = M*N$ ,  $MS = 1$  and  $NS = M$ .
3. Leading dimension, a widely used concept in BLAS-like APIs, describes the inner dimension of the 2D array memory allocation (as opposed to the conceptual matrix dimension). It resembles the stride in a way that it defines the spacing between elements in the outer dimension. The most typical use cases where it shows difference from the matrix inner dimension is when the matrix is only part of the data in the allocated memory, addressing submatrices, or addressing matrices from an aligned memory allocation. Therefore, the leading dimension LDA in a column-major matrix A must satisfy  $LDA \geq M$ , whereas in a row-major matrix A, it must satisfy  $LDA \geq N$ . To transition from the leading dimension concept to using strides, this entails  $MS \geq N$  and  $NS = 1$  or  $MS = 1$  and  $NS \geq M$ . Keep in mind that, while these are some practical use cases, these inequalities do not impose technical constraints with respect to an acceptable specification of the strides.

Other commonly used GEMM features, such as alpha/beta output blending, can also be achieved using this `matmul` operation along with other pointwise operations.

## Attributes

The commonly used GEMM operation can also be achieved using this `matmul` operation along with other pointwise operations for output blending.

Attributes of a cuDNN backend `matmul` descriptor are values of enumeration type `cuda::dnn::BackendAttribute_t` with prefix `CUDNN_ATTR_OPERATION_MATMUL_`:

**CUDNN\_ATTR\_OPERATION\_MATMUL\_ADESC**

The matrix A descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_BDESC**

The matrix B descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_CDESC**

The matrix C descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_IRREGULARLY\_STRIDED\_BATCH\_COUNT**

Number of `matmul` operations to perform in the batch on matrix. Default = 1.

- ▶ CUDNN\_TYPE\_INT64; one element.
- ▶ Default value is 1.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_GEMM\_M\_OVERRIDE\_DESC**

The tensor `gemm_m_override` descriptor. Allows you to override the M dimension of a batch matrix multiplication through this tensor. It is only supported in pattern 6 in the runtime fusion engine documented in [Runtime Fusion Engine](#).

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_GEMM\_N\_OVERRIDE\_DESC**

The tensor `gemm_n_override` descriptor. Allows you to override the N dimension of a batch matrix multiplication through this tensor. It is only supported in pattern 6 in the runtime fusion engine documented in [Runtime Fusion Engine](#).

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_MATMUL\_GEMM\_K\_OVERRIDE\_DESC**

The tensor `gemm_k_override` descriptor. Allows you to override the K dimension of a batch matrix multiplication through this tensor. It is only supported in pattern 6 in the runtime fusion engine documented in [Runtime Fusion Engine](#).

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.

- ▶ Optional attribute.

#### **CUDNN\_ATTR\_OPERATION\_MATMUL\_DESC**

The `matmul` operation descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_MATMUL_DESCRIPTOR`.
- ▶ Required attribute.

### Finalization

In the finalization of the `matmul` operation, the tensor dimensions of the matrices A, B and C will be checked to ensure that they satisfy the requirements of matrix multiplication:

`cudaBackendFinalize()` with a `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` can have the following return values:

#### **CUDNN\_STATUS\_NOT\_SUPPORTED**

An unsupported attribute value was encountered. Some possible cause:

- ▶ If not all of the matrices A, B and C are at least rank-2 tensors.

#### **CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes:

- ▶ The `CUDNN_ATTR_OPERATION_MATMUL_IRREGULARLY_STRIDED_BATCH_COUNT` specified is a negative value.
- ▶ The `CUDNN_ATTR_OPERATION_MATMUL_IRREGULARLY_STRIDED_BATCH_COUNT` and one or more of the batch sizes of the matrices A, B and C are not equal to one. That is to say there is a conflict where both irregularly and regularly strided batched matrix multiplication are specified, which is not a valid use case.
- ▶ The dimensions of the matrices A, B and C do not satisfy the requirements of matrix multiplication.

#### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.17. **CUDNN\_BACKEND\_OPERATION\_NORM\_BACKWARD\_DESCRIPTOR**

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_NORM_BACKWARD_DESCRIPTOR, &desc)`, the cuDNN backend normalization backward operation specifies a node for a backward normalization that takes as input the gradient tensor `dY` and outputs the gradient tensor `dX` and weight gradients `dScale` and `dBias`. The normalization mode is set using the `CUDNN_ATTR_OPERATION_NORM_BWD_MODE` attribute.

### Limitations

- ▶ Does not support `CUDNN_GROUP_NORM` mode.



Table 56. Supported Configurations for

`CUDNN_BACKEND_OPERATION_NORM_BACKWARD_DESCRIPTOR`

<code>CUDNN_ATTR_OPERATION_NORM_BWD_MODE</code>			
<code>CUDNN_LAYER_NORM</code>	<code>CUDNN_INSTANCE_NORM</code>	<code>CUDNN_BATCH_NORM</code>	<code>CUDNN_GROUP_NORM</code>
Yes	Yes	Yes	No



Note: In addition to single GPU, `CUDNN_BATCH_NORM` also supports single node multi-GPU batch norm, while other normalization modes only support running on a single GPU. For more information, refer to the [DReluForkDBn](#) pattern.

## Attributes

### `CUDNN_ATTR_OPERATION_NORM_BWD_MODE`

Chooses the normalization mode for the norm backward operation.

- ▶ `CUDNN_TYPE_NORM_MODE`; one element.
- ▶ Required attribute.

### `CUDNN_ATTR_OPERATION_NORM_BWD_XDESC`

Input tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

### `CUDNN_ATTR_OPERATION_NORM_BWD_MEAN_DESC`

Saved mean input tensor descriptor for reusing the mean computed during the forward computation of the training phase.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Optional attribute.

### `CUDNN_ATTR_OPERATION_NORM_BWD_INV_VARIANCE_DESC`

Saved inverse variance input tensor descriptor for reusing the mean computed during the forward computation of the training phase.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Optional attribute.

### `CUDNN_ATTR_OPERATION_NORM_BWD_DYDESC`

Gradient tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.

- ▶ Optional attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_DYDESC**

Gradient tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_SCALE\_DESC**

Normalization scale descriptor. Note that the bias descriptor is not necessary for the backward pass.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_EPSILON\_DESC**

Scalar input tensor descriptor for the epsilon value. The epsilon values are needed only if the saved mean and variances are not passed as inputs to the operation.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_DSCALE\_DESC**

Scale gradient tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_DBIAS\_DESC**

Bias gradient tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_DXDESC**

Input gradient tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_BWD\_PEER\_STAT\_DESCS**

Vector of tensor descriptors for the communication buffers used in multi-GPU normalization. Typically, one buffer is provided for every GPU in the node. This is an optional attribute only used for multi-GPU tensor stats reduction.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one or more elements of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**Finalization**

In the finalization stage, the attributes are checked to ensure there are no conflicts.

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes are:

- ▶ The tensor dimensions of the gradient tensors  $dy$ ,  $dx$ , and input tensor  $x$ , do not match.
- ▶ The channel count  $C$  for the `mean`, `scale`, and `inv_variance` tensors do not match.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

**9.3.18. CUDNN\_BACKEND\_OPERATION\_NORM\_FORWARD\_DESCRIPTOR**

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_NORM_FORWARD_DESCRIPTOR, &desc)`, the cuDNN backend normalization forward operation specifies a node for a forward normalization that takes as input a tensor  $X$  and produces a normalized output  $Y$  with the normalization mode set by the `CUDNN_ATTR_OPERATION_NORM_FWD_MODE` attribute. The operation supports optional running stats computation and allows for storing the computed means and variances for reuse in the backwards calculation depending on the setting of the `CUDNN_ATTR_OPERATION_NORM_FWD_PHASE` attribute.

**Limitations**

- ▶ Does not support `CUDNN_GROUP_NORM` mode.
- ▶ Batch norm only supports forward training and not forward inference.

Table 57. Supported Configurations for `CUDNN_BACKEND_OPERATION_NORM_FORWARD_DESCRIPTOR`

	<b>CUDNN_ATTR_OPERATION_NORM_FWD_MODE</b>			
<b>CUDNN_ATTR_OPERATION_NORM_FWD_PHASE</b>	<b>CUDNN_LAYER_NORM</b>	<b>CUDNN_INSTANCE_NORM</b>	<b>CUDNN_BATCH_NORM</b>	<b>CUDNN_GROUP_NORM</b>
<code>CUDNN_NORM_FWD_TRAINING</code>	Yes	Yes	Yes	No

	CUDNN_ATTR_OPERATION_NORM_FWD_MODE			
CUDNN_ATTR_OPE	CUDNN_LAYER_NOI	CUDNN_INSTANCE_	CUDNN_BATCH_NOI	CUDNN_GROUP_NORM
CUDNN_NORM_FWD_INFERENCE	Yes	Yes	No	No



Note: In addition to single-GPU, batch normalization supports running on single node multi-GPUs, while other normalization modes only support running on a single GPU. For more information, refer to the [BNAddRelu](#) pattern.

## Attributes

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_MODE

Chooses the normalization mode for the norm forward operation.

- ▶ CUDNN\_TYPE\_NORM\_MODE; one element.
- ▶ Required attribute.

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_PHASE

Selects the training or inference phase for the norm forward operation.

- ▶ CUDNN\_TYPE\_NORM\_FWD\_PHASE; one element.
- ▶ Required attribute.

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_XDESC

Input tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_MEAN\_DESC

Estimated mean input tensor descriptor for the inference phase and the computed mean output tensor descriptor for the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_INV\_VARIANCE\_DESC

Estimated inverse variance input tensor descriptor for the inference phase and the computed inverse variance output tensor descriptor for the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

### CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_SCALE\_DESC

Normalization scale input tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_BIAS\_DESC**

Normalization bias input tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_EPSILON\_DESC**

Scalar input tensor descriptor for the epsilon value used in normalization calculation.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_EXP\_AVG\_FACTOR\_DESC**

Scalar input tensor descriptor for the exponential average factor value used in running stats computation.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_INPUT\_RUNNING\_MEAN\_DESC**

Input running mean tensor descriptor for the running stats computation in the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_INPUT\_RUNNING\_VAR\_DESC**

Input running variance tensor descriptor for the running stats computation in the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_OUTPUT\_RUNNING\_MEAN\_DESC**

Output running mean tensor descriptor for the running stats computation in the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.

- ▶ Optional attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_OUTPUT\_RUNNING\_VAR\_DESC**

Output running variance tensor descriptor for the running stats computation in the training phase.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_YDESC**

Tensor descriptor for the output of the normalization operation.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_NORM\_FWD\_PEER\_STAT\_DESCS**

Vector of tensor descriptors for the communication buffers used in multi-GPU normalization. Typically, one buffer is provided for every GPU in the node. This is an optional attribute only used for multi-GPU tensor stats reduction.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one or more elements of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

## Finalization

In the finalization stage, the attributes are checked to ensure there are no conflicts.

#### **CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes are:

- ▶ The output tensor dimensions do not match the input tensor dimensions.
- ▶ The channel count  $C$  for the `mean`, `scale`, `bias`, and `inv_variance` tensors do not match.

#### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.19. CUDNN\_BACKEND\_OPERATION\_POINTWISE\_DESCRIPTOR

Represents a pointwise operation that implements the equation

$Y = \text{op}(\alpha_1 * X)$  or  $Y = \text{op}(\alpha_1 * X, \alpha_2 * B)$  depending on the operation type. The actual type of operation represented by `op()` above depends on the

`CUDNN_ATTR_OPERATION_POINTWISE_PW_DESCRIPTOR` attribute in the descriptor. This operation descriptor supports operations with single-input single-output.

For a list of supported operations, refer to the [`cudaPointwiseMode\_t`](#) section.

For dual-input pointwise operations, broadcasting is assumed when a tensor dimension in one of the tensors is 1 while the other tensors corresponding dimension is not 1.

For three-input single-output pointwise operations, we do not support broadcasting in any tensor.

This opaque struct can be created with `cudaBackendCreateDescriptor()` (`CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR`).

## Attributes

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_PW\_DESCRIPTOR**

Sets the descriptor containing the mathematical settings of the pointwise operation. This attribute is required.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_XDESC**

Sets the descriptor for the input tensor  $X$ . This attribute is required for pointwise mathematical functions or activation forward propagation computations.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_BDESC**

If the operation requires 2 inputs, such as `add` or `multiply`, this attribute sets the second input tensor  $\beta$ . If the operation requires only 1 input, this field is not used and should not be set.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_YDESC**

Sets the descriptor for the output tensor  $Y$ . This attribute is required for pointwise mathematical functions or activation forward propagation computations.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_TDESC**

Sets the descriptor for the tensor  $T$ . This attribute is required for `CUDNN_ATTR_POINTWISE_MODE` set to `CUDNN_POINTWISE_BINARY_SELECT` and acts as the mask based on which the selection is done.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_ALPHA1**

Sets the scalar `alpha1` value in the equation. Can be in float or half. This attribute is optional, if not set, the default value is `1.0`.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_ALPHA2**

If the operation requires two inputs, such as `add` or `multiply`, this attribute sets the scalar `alpha2` value in the equation. Can be in float or half. This attribute is optional, if not set, the default value is `1.0`. If the operation requires only 1 input, this field is not used and should not be set.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_DXDESC**

Sets the descriptor for the output tensor  $dX$ . This attribute is required for pointwise activation back propagation computations.

### **CUDNN\_ATTR\_OPERATION\_POINTWISE\_DYDESC**

Sets the descriptor for the input tensor  $dY$ . This attribute is required for pointwise activation back propagation computations.

## Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

### **CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Some possible causes are:

- ▶ The number of dimensions do not match between the input and output tensors.
- ▶ The input/output tensor dimensions do not agree with the above described automatic broadcasting rules.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.20. CUDNN\_BACKEND\_OPERATION\_REDUCTION\_DESCRIPTOR

The cuDNN backend `reduction` operation descriptor represents an operation node that implements reducing values of an input tensor  $x$  in one or more dimensions to get an output tensor  $y$ . The math operation and compute data type used for reducing tensor values is specified via `CUDNN_ATTR_OPERATION_REDUCTION_DESC`.

This operation descriptor can be created with

```
cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR,
    &desc);
```

The output tensor  $y$  should be the size as that of input tensor  $x$ , except dimension(s) where its size is 1.

## Attributes

Attributes of a cuDNN backend `reduction` descriptor are values of enumeration type `cudaBackendAttributeName_t` with prefix `CUDNN_ATTR_OPERATION_REDUCTION_`:

### **CUDNN\_ATTR\_OPERATION\_REDUCTION\_XDESC**

The matrix  $x$  descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR` one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

### **CUDNN\_ATTR\_OPERATION\_REDUCTION\_YDESC**

The matrix  $y$  descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR` one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

### **CUDNN\_ATTR\_OPERATION\_REDUCTION\_DESC**

The `reduction` operation descriptor.



- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR` one element of descriptor type `CUDNN_BACKEND_REDUCTION_DESCRIPTOR`.
- ▶ Required attribute.

## Finalization

In the finalization of the `reduction` operation, the dimensions of tensors `x` and `y` are checked to ensure that they satisfy the requirements of the reduction operation.

`cudaBackendFinalize()` with a `CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR` can have the following return values:

### `CUDNN_STATUS_BAD_PARAM`

Invalid or inconsistent attribute values are encountered. Some possible causes:

- ▶ The dimensions of the tensors `x` and `y` do not satisfy the requirements of the reduction operation.

### `CUDNN_STATUS_SUCCESS`

The descriptor was finalized successfully.

## 9.3.21. `CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR`

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR, &desc)`; the cuDNN backend resample backward operation descriptor specifies an operation node for backward resampling. It computes the input tensor gradient `dx` from output tensor gradient `dy` with backward resampling done according to `CUDNN_ATTR_RESAMPLE_MODE` with output scaling  $\alpha$  and residual add with  $\beta$  scaling.

## Attributes

### `CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DESC`

Resample operation descriptor (`CUDNN_BACKEND_RESAMPLE_DESCRIPTOR`) instance containing metadata about the operation.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_RESAMPLE_DESCRIPTOR`.
- ▶ Required attribute.

### `CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DXDESC`

Input tensor gradient descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

### `CUDNN_ATTR_OPERATION_RESAMPLE_BWD_DYDESC`

Output tensor gradient descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_BWD\_IDXDESC**

Tensor containing maxpool or nearest neighbor resampling indices to be used in backprop.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_BWD\_ALPHA**

Sets the alpha parameter used in blending.

- ▶ CUDNN\_TYPE\_DOUBLE or CUDNN\_TYPE\_FLOAT; one element.
- ▶ Optional attribute.
- ▶ Default value is 1.0.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_BWD\_BETA**

Sets the beta parameter used in blending.

- ▶ CUDNN\_TYPE\_DOUBLE or CUDNN\_TYPE\_FLOAT; one element.
- ▶ Optional attribute.
- ▶ Default value is 0.0.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_BWD\_XDESC**

Input tensor X descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.
- ▶ Required for NCHW layout.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_BWD\_YDESC**

Input tensor Y descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.
- ▶ Required for NCHW layout

## Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Possible causes are:

- ▶ The output shape calculated based on the padding and strides does not match the given output tensor dimensions.
- ▶ The shape of `YDESC` and `IDXDESC` (if given) do not match.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.22. CUDNN\_BACKEND\_OPERATION\_RESAMPLE\_FWD\_DESCRIPTOR

Created with

`cudaDnnBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR, &desc)`; the cuDNN backend resample forward operation descriptor specifies an operation node for forward resampling. It computes the output tensor  $y$  of image tensor  $x$  resampled according to `CUDNN_ATTR_RESAMPLE_MODE`, with output scaling  $\alpha$  and residual add with  $\beta$  scaling.

The resampling mode acts independently on each spatial dimension. For spatial dimension  $i$ , the output spatial dimension size  $y_i$  can be calculated by combining input image's spatial dimension size  $x_i$ , post padding  $post_i$ , pre padding  $pre_i$ , stride  $s_i$ , window size  $w_i$  as:  $y_i = 1 + (x_i + post_i + pre_i - w_i) / s_i$

### Attributes

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_DESC**

Resample operation descriptor (`CUDNN_BACKEND_RESAMPLE_DESCRIPTOR`) instance containing metadata about the operation.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_RESAMPLE_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_XDESC**

Input tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_YDESC**

Output tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR`; one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR`.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_IDXDESC**

Tensor containing maxpool or nearest neighbor resampling indices to be used in backprop.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute (primarily used for use cases involving training).

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_ALPHA**

Sets the alpha parameter used in blending.

- ▶ CUDNN\_TYPE\_DOUBLE or CUDNN\_TYPE\_FLOAT; one element.
- ▶ Optional attribute.
- ▶ Default value is 1.0.

**CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_BETA**

Sets the beta parameter used in blending.

- ▶ CUDNN\_TYPE\_DOUBLE or CUDNN\_TYPE\_FLOAT; one element.
- ▶ Optional attribute.
- ▶ Default value is 0.0.

## Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered. Possible causes are:

- ▶ The output shape calculated based on the padding and strides does not match the given output tensor dimensions.
- ▶ The shape of the YDESC and IDXDESC (if given) do not match.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.23. CUDNN\_BACKEND\_OPERATION\_RNG\_DESCRIPTOR

Created with

```

cudnnBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_RNG_DESCRIPTOR, &desc);

```

the cuDNN backend `Rng` operation descriptor specifies an operation node for generating a tensor with random numbers based on the probability distribution specified in the `Rng` descriptor.

The random numbers are generated using a [Philox random number generator \(RNG\) as described in Pytorch](#). The Philox object takes a seed value, a subsequence for starting the generation, and an offset for the subsequence. Seed and offset can be set by using the attributes. The subsequence is internally set, to ensure independent random numbers.

## Attributes

### **CUDNN\_ATTR\_OPERATION\_RNG\_DESC**

Rng descriptor (CUDNN\_BACKEND\_RNG\_DESCRIPTOR) instance containing metadata about the operation.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_RNG\_DESCRIPTOR.
- ▶ Required attribute.

### **CUDNN\_ATTR\_OPERATION\_RNG\_YDESC**

Output tensor descriptor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

### **CUDNN\_ATTR\_OPERATION\_RNG\_SEED**

Sets the seed for the random number generator which creates the  $y$  tensor. It can be a host INT64 value or a backend descriptor binded to a value on the device. Only supports a tensor with all dimensions set to 1 and all strides set to 1.

- ▶ CUDNN\_TYPE\_INT64; one element or CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.
- ▶ Default value is 0.

### **CUDNN\_ATTR\_OPERATION\_RNG\_OFFSET\_DESC**

Tensor descriptor for the offset used in the RNG Philox object. Only supports a tensor with all dimensions set to 1 and all strides set to 1.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Required attribute.

## Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

### **CUDNN\_STATUS\_BAD\_PARAM**

CUDNN\_ATTR\_OPERATION\_RNG\_OFFSET\_DESC or CUDNN\_ATTR\_OPERATION\_RNG\_SEED do not have all dimensions and strides set to 1.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.24. CUDNN\_BACKEND\_OPERATION\_SIGNAL\_DESCRIPTOR

Created with

`cudaBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_SIGNAL_DESCRIPTOR, &desc);` the cuDNN backend signal operation descriptor specifies an operation node for updating or waiting on a flag variable. Signaling operations can be used to communicate between cuDNN operation graphs, even with operation graphs in another GPU.

This operation, to connect to other nodes in the graph, also has a pass-through input tensor, which is not operated on and is just passed along to the output tensor. This mandatory pass-through input tensor helps in determining the predecessor node after which the signal operation should be executed. The optional output tensor helps in determining the successor node before which the signal execution should have completed. It is also guaranteed that for a non-virtual tensor as the output tensor, all writes for the tensor will have taken place before the signal value is updated by the operation.

### Attributes

#### **CUDNN\_ATTR\_OPERATION\_SIGNAL\_MODE**

The signaling mode to use.

- ▶ `CUDNN_TYPE_SIGNAL_MODE;`
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_SIGNAL\_FLAGDESC**

Flag tensor descriptor.

#### **CUDNN\_ATTR\_OPERATION\_RESAMPLE\_FWD\_YDESC**

Output tensor descriptor.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR;` one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR.`
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_SIGNAL\_VALUE**

The scalar value to compare or update the flag variable with.

- ▶ `CUDNN_TYPE_INT64`
- ▶ Required attribute.

#### **CUDNN\_ATTR\_OPERATION\_SIGNAL\_XDESC**

A pass-through input tensor to enable connecting this signal operation to other nodes in the graph.

- ▶ `CUDNN_TYPE_BACKEND_DESCRIPTOR;` one element of descriptor type `CUDNN_BACKEND_TENSOR_DESCRIPTOR.`
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATION\_SIGNAL\_YDESC**

The output tensor for the pass-through input tensor.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element of descriptor type CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR.
- ▶ Optional attribute.

## Finalization

In the finalization stage, the attributes are cross checked to make sure there are no conflicts. The status below may be returned:

**CUDNN\_STATUS\_BAD\_PARAM**

Invalid or inconsistent attribute values are encountered.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.25. CUDNN\_BACKEND\_OPERATIONGRAPH\_DESCRIPTOR

Created with descriptor type value CUDNN\_BACKEND\_OPERATIONGRAPH\_DESCRIPTOR, cuDNN backend operation graph descriptor describes an operation graph, a small network of one or more operations connected by virtual tensors. Operation graph defines users' computation case or mathematical expression that they wish to compute.

### Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type [`cudaBackendAttributeName\_t`](#) with prefix CUDNN\_ATTR\_OPERATIONGRAPH\_:

**CUDNN\_ATTR\_OPERATIONGRAPH\_HANDLE**

A cuDNN handle.

- ▶ CUDNN\_TYPE\_HANDLE; one element.
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATIONGRAPH\_OPS**

Operation nodes to form the operation graph.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one or more elements of descriptor type CUDNN\_BACKEND\_OPERATION\_\*\_DESCRIPTOR().
- ▶ Required attribute.

**CUDNN\_ATTR\_OPERATIONGRAPH\_ENGINE\_GLOBAL\_COUNT**

The number of engines to support the operation graph.

- ▶ CUDNN\_TYPE\_INT64; one element.
- ▶ Read-only attribute.

**CUDNN\_ATTR\_OPERATIONGRAPH\_ENGINE\_SUPPORTED\_COUNT**

The number of engines that support the operation graph.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Read-only attribute; placeholder only; currently not supported.

## Finalization

### **CUDNN\_STATUS\_BAD\_PARAM**

An invalid attribute value was encountered. For example:

- ▶ One of the backend descriptors in `CUDNN_ATTR_OPERATIONGRAPH_OPS` is not finalized.
- ▶ The value `CUDNN_ATTR_OPERATIONGRAPH_HANDLE` is not a valid cuDNN handle.

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

An unsupported attribute value was encountered. For example:

- ▶ The combination of operations of attribute `CUDNN_ATTR_OPERATIONGRAPH_OPS` is not supported.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.26. CUDNN\_BACKEND\_POINTWISE\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_POINTWISE_DESCRIPTOR, &desc)`; the cuDNN backend pointwise descriptor specifies the parameters for a pointwise operator like mode, math precision, nan propagation and so on.

## Attributes

Attributes of a cuDNN backend convolution descriptor are values of enumeration type `cudaBackendAttributeName_t` with prefix `CUDNN_ATTR_POINTWISE_`:

### **CUDNN\_ATTR\_POINTWISE\_MODE**

Mode of the pointwise operation.

- ▶ `CUDNN_TYPE_POINTWISE_MODE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_POINTWISE\_MATH\_PREC**

The math precision of the computation.

- ▶ `CUDNN_TYPE_DATA_TYPE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_POINTWISE\_NAN\_PROPAGATION**

Specifies a method by which to propagate NaNs.

- ▶ `CUDNN_TYPE_NAN_PROPAGATION`; one element.
- ▶ Required only for comparison based pointwise modes, like ReLU.
- ▶ Current support only includes enum value `CUDNN_PROPAGATE_NAN`.



- ▶ Default value: CUDNN\_NOT\_PROPAGATE\_NAN.

#### **CUDNN\_ATTR\_POINTWISE\_RELU\_LOWER\_CLIP**

Sets the lower clip value for ReLU. If  $(value < lower\_clip)$   $value = lower\_clip + lower\_clip\_slope * (value - lower\_clip)$ ;

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: 0.0f.

#### **CUDNN\_ATTR\_POINTWISE\_RELU\_UPPER\_CLIP**

Sets the upper clip value for ReLU. If  $(value > upper\_clip)$   $value = upper\_clip$ ;

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: Numeric limit max.

#### **CUDNN\_ATTR\_POINTWISE\_RELU\_LOWER\_CLIP\_SLOPE**

Sets the lower clip slope value for ReLU. If  $(value < lower\_clip)$   $value = lower\_clip + lower\_clip\_slope * (value - lower\_clip)$ ;

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: 0.0f.

#### **CUDNN\_ATTR\_POINTWISE\_ELU\_ALPHA**

Sets the alpha value for ELU. If  $(value < 0.0)$   $value = alpha * (e^{value} - 1.0)$ ;

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: 1.0f.

#### **CUDNN\_ATTR\_POINTWISE\_SOFTPLUS\_BETA**

Sets the beta value for softplus.  $value = \log(1 + e^{(beta * value)}) / beta$

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: 1.0f

#### **CUDNN\_ATTR\_POINTWISE\_SWISH\_BETA**

Sets the beta value for swish.  $value = value / (1 + e^{(-beta * value)})$

- ▶ CUDNN\_TYPE\_DOUBLE / CUDNN\_TYPE\_FLOAT; one element.
- ▶ Default value: 1.0f.

#### **CUDNN\_ATTR\_POINTWISE\_AXIS**

Sets the axis value for GEN\_INDEX. The index will be generated for this axis.

- ▶ CUDNN\_TYPE\_INT64; one element.
- ▶ Default value: -1.
- ▶ Needs to lie between  $[0, input\_dim\_size - 1]$ . For example, if your input has dimensions  $[N, C, H, W]$ , the axis can be set to anything in  $[0, 3]$ .

## Finalization

`cudaBackendFinalize()` with a `CUDNN_BACKEND_POINTWISE_DESCRIPTOR` can have the following return values:

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.27. CUDNN\_BACKEND\_REDUCTION\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_REDUCTION_DESCRIPTOR, &desc)`; the cuDNN backend `reduction` descriptor specifies any metadata, including the math operation and compute data type, needed for the `reduction` operation.

## Attributes

### **CUDNN\_ATTR\_REDUCTION\_OPERATOR**

The math operation used for the `reduction` operation.

- ▶ `CUDNN_TYPE_REDUCTION_OPERATOR_TYPE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_REDUCTION\_COMP\_TYPE**

The compute precision used for the `reduction` operation.

- ▶ `CUDNN_TYPE_DATA_TYPE`; one element.
- ▶ Required attribute.

## Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is `CUDNN_BACKEND_REDUCTION_DESCRIPTOR` are:

### **CUDNN\_STATUS\_NOT\_SUPPORTED**

An unsupported attribute value was encountered. Some possible causes are:

- ▶ `CUDNN_ATTR_REDUCTION_OPERATOR` is not set to either of `CUDNN_REDUCE_TENSOR_ADD`, `CUDNN_REDUCE_TENSOR_MUL`, `CUDNN_REDUCE_TENSOR_MIN`, or `CUDNN_REDUCE_TENSOR_MAX`.

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.28. CUDNN\_BACKEND\_RESAMPLE\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_RESAMPLE_DESCRIPTOR, &desc)`; the cuDNN backend `resample` descriptor specifies the parameters for a `resample` operation (upsampling or downsampling) in both forward and backward propagation.

## Attributes

### **CUDNN\_ATTR\_RESAMPLE\_MODE**

Specifies mode of resampling, for example, average pool, nearest-neighbor, etc.

- ▶ CUDNN\_TYPE\_RESAMPLE\_MODE; one element.
- ▶ Default value is CUDNN\_RESAMPLE\_NEAREST.

### **CUDNN\_ATTR\_RESAMPLE\_COMP\_TYPE**

Compute data type for the resampling operator.

- ▶ CUDNN\_TYPE\_DATA\_TYPE; one element.
- ▶ Default value is CUDNN\_DATA\_FLOAT.

### **CUDNN\_ATTR\_RESAMPLE\_NAN\_PROPAGATION**

Specifies a method by which to propagate NaNs.

- ▶ CUDNN\_TYPE\_NAN\_PROPAGATION; one element.
- ▶ Default value is CUDNN\_NOT\_PROPAGATE\_NAN.

### **CUDNN\_ATTR\_RESAMPLE\_SPATIAL\_DIMS**

Specifies the number of spatial dimensions to perform the resampling over.

- ▶ CUDNN\_TYPE\_INT64; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_RESAMPLE\_PADDING\_MODE**

Specifies which values to use for padding.

- ▶ CUDNN\_TYPE\_PADDING\_MODE; one element.
- ▶ Default value is CUDNN\_ZERO\_PAD.

### **CUDNN\_ATTR\_RESAMPLE\_STRIDES**

Stride in each dimension for the kernel/filter.

- ▶ CUDNN\_TYPE\_INT64 or CUDNN\_TYPE\_FRACTION; at most CUDNN\_MAX\_DIMS - 2.
- ▶ Required attribute.

### **CUDNN\_ATTR\_RESAMPLE\_PRE\_PADDINGS**

Padding added to the beginning of the input tensor in each dimension.

- ▶ CUDNN\_TYPE\_INT64 or CUDNN\_TYPE\_FRACTION; at most CUDNN\_MAX\_DIMS - 2.
- ▶ Required attribute.

### **CUDNN\_ATTR\_RESAMPLE\_POST\_PADDINGS**

Padding added to the end of the input tensor in each dimension.

- ▶ CUDNN\_TYPE\_INT64 or CUDNN\_TYPE\_FRACTION; at most CUDNN\_MAX\_DIMS - 2.
- ▶ Required attribute.

**CUDNN\_ATTR\_RESAMPLE\_WINDOW\_DIMS**

Spatial dimensions of filter.

- ▶ CUDNN\_TYPE\_INT64 or CUDNN\_TYPE\_FRACTION; at most CUDNN\_MAX\_DIMS - 2.
- ▶ Required attribute.

## Finalization

The return values for `cudaBackendFinalize()` when called with a CUDNN\_BACKEND\_RESAMPLE\_DESCRIPTOR is:

**CUDNN\_STATUS\_NOT\_SUPPORTED**

An unsupported attribute value was encountered. Some possible causes are:

- ▶ An `elemCount` argument for setting CUDNN\_ATTR\_RESAMPLE\_WINDOW\_DIMS, CUDNN\_ATTR\_RESAMPLE\_STRIDES, CUDNN\_ATTR\_RESAMPLE\_PRE\_PADDINGS, and CUDNN\_ATTR\_RESAMPLE\_POST\_PADDINGS is not equal to the value set for CUDNN\_ATTR\_RESAMPLE\_SPATIAL\_DIMS.
- ▶ CUDNN\_ATTR\_RESAMPLE\_MODE is set to CUDNN\_RESAMPLE\_BILINEAR and any of the CUDNN\_ATTR\_RESAMPLE\_WINDOW\_DIMS are not set to 2.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.29. CUDNN\_BACKEND\_RNG\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_RNG_DESCRIPTOR, &desc)`; the cuDNN backend `Rng` descriptor specifies any metadata, including the probability distribution that will be used to generate the tensor and the distribution's corresponding parameters.

### Attributes

**CUDNN\_ATTR\_RNG\_DISTRIBUTION**

The probability distribution used for the `rng` operation.

- ▶ CUDNN\_TYPE\_RNG\_DISTRIBUTION; one element.
- ▶ Default value is CUDNN\_RNG\_DISTRIBUTION\_BERNOULLI.

**CUDNN\_ATTR\_RNG\_NORMAL\_DIST\_MEAN**

The mean value for the normal distribution, used if CUDNN\_ATTR\_RNG\_DISTRIBUTION = CUDNN\_RNG\_DISTRIBUTION\_NORMAL.

- ▶ CUDNN\_TYPE\_DOUBLE ; one element.
- ▶ Default value is -1.

**CUDNN\_ATTR\_RNG\_NORMAL\_DIST\_STANDARD\_DEVIATION**

The standard deviation value for the normal distribution, used if CUDNN\_ATTR\_RNG\_DISTRIBUTION = CUDNN\_RNG\_DISTRIBUTION\_NORMAL.

- ▶ `CUDNN_TYPE_DOUBLE` ; one element.
- ▶ Default value is -1.

**CUDNN\_ATTR\_RNG\_UNIFORM\_DIST\_MAXIMUM**

The maximum value for the range used in uniform distribution, used if `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_UNIFORM`.

- ▶ `CUDNN_TYPE_DOUBLE` ; one element.
- ▶ Default value is -1.

**CUDNN\_ATTR\_RNG\_UNIFORM\_DIST\_MINIMUM**

The minimum value for the range used in uniform distribution, used if `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_UNIFORM`.

- ▶ `CUDNN_TYPE_DOUBLE` ; one element.
- ▶ Default value is -1.

**CUDNN\_ATTR\_RNG\_BERNOULLI\_DIST\_PROBABILITY**

The probability of generating 1's in the tensor, used if `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_BERNOULLI`.

- ▶ `CUDNN_TYPE_DOUBLE` ; one element.
- ▶ Default value is -1.

## Finalization

Return values of `cudaBackendFinalize(desc)` where `desc` is `CUDNN_BACKEND_RNG_DESCRIPTOR` are:

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid attribute value was encountered. For example:

- ▶ If `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_NORMAL` and the standard deviation supplied is negative.
- ▶ If `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_UNIFORM` and the maximum value of the range is lower than minimum value.
- ▶ If `CUDNN_ATTR_RNG_DISTRIBUTION = CUDNN_RNG_DISTRIBUTION_BERNOULLI` and the probability supplied is negative.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.3.30. CUDNN\_BACKEND\_TENSOR\_DESCRIPTOR

Created with `cudaBackendCreateDescriptor(CUDNN_BACKEND_TENSOR_DESCRIPTOR, &desc)`; the cuDNN backend tensor allows users to specify the memory storage of a generic tensor. A tensor is identified by a unique identifier and described by its data type, its data byte-alignment requirements, and the extents and strides of its dimensions. Optionally, a tensor element can be vector in one of its dimensions. A tensor can also

be set to be virtual when it is an intermediate variable in a computation graph and not mapped to physical global memory storage.

## Attributes

Attributes of a cuDNN backend tensor descriptors are values of enumeration type

`cudaBackendAttributeName_t` with prefix `CUDNN_ATTR_TENSOR_`:

### **CUDNN\_ATTR\_TENSOR\_UNIQUE\_ID**

An integer that uniquely identifies the tensor.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_TENSOR\_DATA\_TYPE**

Data type of tensor.

- ▶ `CUDNN_TYPE_DATA_TYPE`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_TENSOR\_BYTE\_ALIGNMENT**

Byte alignment of pointers for this tensor.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Required attribute.

### **CUDNN\_ATTR\_TENSOR\_DIMENSIONS**

Tensor dimensions.

- ▶ `CUDNN_TYPE_INT64`; at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_TENSOR\_STRIDES**

Tensor strides.

- ▶ `CUDNN_TYPE_INT64`; at most `CUDNN_MAX_DIMS` elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_TENSOR\_VECTOR\_COUNT**

Size of vectorization.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Default value: 1

### **CUDNN\_ATTR\_TENSOR\_VECTORIZED\_DIMENSION**

Index of the vectorized dimension.

- ▶ `CUDNN_TYPE_INT64`; one element.
- ▶ Required to be set before finalization if `CUDNN_ATTR_TENSOR_VECTOR_COUNT` is set to a value different than its default; otherwise it's ignored.

**CUDNN\_ATTR\_TENSOR\_IS\_VIRTUAL**

Indicates whether the tensor is virtual. A virtual tensor is an intermediate tensor in the operation graph that exists in transient and not read from or written to in global device memory.

- ▶ CUDNN\_TYPE\_BOOL; one element.
- ▶ Default value: `false`

**CUDNN\_ATTR\_TENSOR\_RAGGED\_OFFSET\_DESC**

A ragged tensor, that is, a tensor with nested variable length lists as inner dimensions, will have another tensor called the ragged offset descriptor that contains offsets in memory to the next variable length list.

- ▶ CUDNN\_TYPE\_BACKEND\_DESCRIPTOR; one element.
- ▶ Default value: `None`

## Finalization

`cudaBackendFinalize()` with a `CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR` can have the following return values:

**CUDNN\_STATUS\_BAD\_PARAM**

An invalid attribute value was encountered. For example:

- ▶ Any of the tensor dimensions or strides is not positive.
- ▶ The value of the tensor alignment attribute is not divisible by the size of the data type.

**CUDNN\_STATUS\_NOT\_SUPPORTED**

An unsupported attribute value was encountered. For example:

- ▶ The data type attribute is `CUDNN_DATA_INT8x4`, `CUDNN_DATA_UINT8x4`, or `CUDNN_DATA_INT8x32`.
- ▶ The data type attribute is `CUDNN_DATA_INT8` and `CUDNN_ATTR_TENSOR_VECTOR_COUNT` value is not 1, 4, or 32.

**CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

### 9.3.31. CUDNN\_BACKEND\_VARIANT\_PACK\_DESCRIPTOR

Created with

```
cudaBackendCreateDescriptor(CUDNN_BACKEND_VARIANT_PACK_DESCRIPTOR, &desc);
```

the cuDNN backend variant pack plan allows users to set up pointers to device buffers to various non-virtual tensors, identified by unique identifiers, of the operation graph, workspace, and computation intermediates.

## Attributes

### **CUDNN\_ATTR\_VARIANT\_PACK\_UNIQUE\_IDS**

A unique identifier of tensor for each data pointer.

- ▶ CUDNN\_TYPE\_INT64; zero or more elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_VARIANT\_PACK\_DATA\_POINTERS**

Tensor data device pointers.

- ▶ CUDNN\_TYPE\_VOID\_PTR; zero or more elements.
- ▶ Required attribute.

### **CUDNN\_ATTR\_VARIANT\_PACK\_INTERMEDIATES**

Intermediate device pointers.

- ▶ CUDNN\_TYPE\_VOID\_PTR; zero or more elements.
- ▶ Setting attribute unsupported. Placeholder for support to be added in a future version.

### **CUDNN\_ATTR\_VARIANT\_PACK\_WORKSPACE**

Workspace to device pointer.

- ▶ CUDNN\_TYPE\_VOID\_PTR; one element.
- ▶ Required attribute.

## Finalization

The return values for `cudaBackendFinalize()` when called with a cuDNN backend variant pack descriptor is:

### **CUDNN\_STATUS\_SUCCESS**

The descriptor was finalized successfully.

## 9.4. Use Cases

This section describes some typical use cases of the cuDNN backend API; for example, setting up a simple operation graph, setting up an engine config for that operation graph, and finally setting up an execution plan and executing it with data pointers set in a variant pack descriptor.

### 9.4.1. Setting Up An Operation Graph For A Grouped Convolution

This use case creates an operation graph with a single grouped 3D convolution forward operation. It starts by setting up the input and output tensors, binding them to a



convolution forward operation, and finally setting up an operation graph with a single node.

### 1. Create tensor descriptors.

```

cudnnBackendDescriptor_t xDesc;
cudnnBackendCreateDescriptor(CUDNN_BACKEND_TENSOR_DESCRIPTOR, &xDesc);

cudnnDataType_t dtype = CUDNN_DATA_FLOAT;
cudnnBackendSetAttribute(xDesc, CUDNN_ATTR_TENSOR_DATA_TYPE,
                        CUDNN_TYPE_DATA_TYPE, 1, &dtype);

int64_t xDim[] = {n, g, c, d, h, w};
int64_t xStr[] = {g * c * d * h * w, c * d * h * w, d * h * w, h * w, w, 1};
int64_t xUi = 'x';
int64_t alignment = 4;

cudnnBackendSetAttribute(xDesc, CUDNN_ATTR_TENSOR_DIMENSIONS,
                        CUDNN_TYPE_INT64, 6, xDim);

cudnnBackendSetAttribute(xDesc, CUDNN_ATTR_TENSOR_STRIDES,
                        CUDNN_TYPE_INT64, 6, xStr);

cudnnBackendSetAttribute(xDesc, CUDNN_ATTR_TENSOR_UNIQUE_ID,
                        CUDNN_TYPE_INT64, 1, &xUi);

cudnnBackendSetAttribute(xDesc, CUDNN_ATTR_TENSOR_BYTE_ALIGNMENT,
                        CUDNN_TYPE_INT64, 1, &alignment);

cudnnBackendFinalize(xDesc);

```

- Repeat the above step for the convolution filter and output tensor descriptor. The six filter tensor dimensions are [g, k, c, t, r, s] and the six output tensor dimensions are [n, g, k, o, p, q], respectively. Below, when finalizing a convolution operator to which the tensors are bound, dimension consistency is checked, meaning all n, g, c, k values shared among the three tensors are required to be the same. Otherwise, CUDNN\_STATUS\_BAD\_PARAM status is returned.

For backward compatibility with how tensors are specified in [cudnnTensorDescriptor\\_t](#) and used in convolution API, it is also possible to specify a 5D tensor with the following dimension:

- ▶ image: [n, g\*c, d, h, w]
- ▶ filter: [g\*k, c, t, r, s]
- ▶ response: [n, g\*k, o, p, q]

In this format, a similar consistency check is performed when finalizing a convolution operator descriptor to which the tensors are bound.

### 3. Create, set, and finalize a convolution operator descriptor.

```

cudnnBackendDescriptor_t cDesc;
int64_t nbDims = 3;
cudnnDataType_t compType = CUDNN_DATA_FLOAT;
cudnnConvolutionMode_t mode = CUDNN_CONVOLUTION;
int64_t pad[] = {0, 0, 0};
int64_t filterStr[] = {1, 1, 1};
int64_t dilation[] = {1, 1, 1};

cudnnBackendCreateDescriptor(CUDNN_BACKEND_CONVOLUTION_DESCRIPTOR, &cDesc);

cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_SPATIAL_DIMS,
                        CUDNN_TYPE_INT64, 1, &nbDims);

```

```

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_COMP_TYPE,
                           CUDNN_TYPE_DATA_TYPE, 1, &compType);

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_CONV_MODE,
                           CUDNN_TYPE_CONVOLUTION_MODE, 1, &mode);

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_PRE_PADDINGS,
                           CUDNN_TYPE_INT64, nbDims, pad);

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_POST_PADDINGS,
                           CUDNN_TYPE_INT64, nbDims, pad);

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_DILATIONS,
                           CUDNN_TYPE_INT64, nbDims, dilation);

    cudnnBackendSetAttribute(cDesc, CUDNN_ATTR_CONVOLUTION_FILTER_STRIDES,
                           CUDNN_TYPE_INT64, nbDims, filterStr);
    cudnnBackendFinalize(cDesc);

```

#### 4. Create, set, and finalize a convolution forward operation descriptor.

```

    cudnnBackendDescriptor_t fprop;
    float alpha = 1.0;
    float beta = 0.5;

    cudnnBackendCreateDescriptor(CUDNN_BACKEND_OPERATION_CONVOLUTION_FORWARD_DESCRIPTOR,
                                &fprop);
    cudnnBackendSetAttribute(fprop, CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_X,
                            CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &xDesc);
    cudnnBackendSetAttribute(fprop, CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_W,
                            CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &wDesc);
    cudnnBackendSetAttribute(fprop, CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_Y,
                            CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &yDesc);
    cudnnBackendSetAttribute(fprop,
                            CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_CONV_DESC,
                            CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &cDesc);

    cudnnBackendSetAttribute(fprop, CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_ALPHA,
                            dtype, 1, alpha);
    cudnnBackendSetAttribute(fprop, CUDNN_ATTR_OPERATION_CONVOLUTION_FORWARD_BETA,
                            dtype, 1, beta);

    cudnnBackendFinalize(fprop);

```

#### 5. Create, set, and finalize an operation graph descriptor.

```

    cudnnBackendDescriptor_t op_graph;
    cudnnBackendCreateDescriptor(CUDNN_BACKEND_OPERATIONGRAPH_DESCRIPTOR, op_graph);
    cudnnBackendSetAttribute(op_graph, CUDNN_ATTR_OPERATIONGRAPH_OPS,
                            CUDNN_TYPE_BACKEND_DESCRIPTOR, len, ops);
    cudnnBackendSetAttribute(op_graph, CUDNN_ATTR_OPERATIONGRAPH_HANDLE,
                            CUDNN_TYPE_HANDLE, 1, &handle);
    cudnnBackendFinalize(op_graph);

```

## 9.4.2. Setting Up An Engine Configuration

This use case describes the steps with which users can set up an engine config from a previously finalized operation graph. This is an example in which users would like to use the engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX 0` for this operation graph and does not set any performance knobs.

#### 1. Create, set, and finalize an engine descriptor.

```

    cudnnBackendDescriptor_t engine;
    cudnnBackendCreateDescriptor(CUDNN_BACKEND_ENGINE_DESCRIPTOR, &engine);
    cudnnBackendSetAttribute(engine, CUDNN_ATTR_ENGINE_OPERATION_GRAPH,

```

```

        CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &opset);
Int64_t gidx = 0;
cudnnBackendSetAttribute(engine, CUDNN_ATTR_ENGINE_GLOBAL_INDEX,
        CUDNN_TYPE_INT64, 1, &gidx);
cudnnBackendFinalize(engine);

```

The user can query a finalized engine descriptor with `cudnnBackendGetAttribute()` API call for its attributes, including the performance knobs that it has. For simplicity, this use case skips this step and assumes the user is setting up an engine config descriptor below without making any changes to performance knobs.

2. Create, set, and finalize an engine config descriptor.

```

cudnnBackendDescriptor_t engcfg;
cudnnBackendSetAttribute(engcfg, CUDNN_ATTR_ENGINECFG_ENGINE,
        CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &engine);
cudnnBackendFinalize(engcfg);

```

### 9.4.3. Setting Up And Executing A Plan

This use case describes the steps with which users set up an execution plan with a previously finalized engine config descriptor, set up the data pointer variant pack, and finally execute the plan.

1. Create, set, and finalize an execution plan descriptor. Obtain workspace size to allocate.

```

cudnnBackendDescriptor_t plan;
cudnnBackendCreateDescriptor(CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR, &plan);
cudnnBackendSetAttribute(plan, CUDNN_ATTR_EXECUTION_PLAN_ENGINE_CONFIG,
        CUDNN_TYPE_BACKEND_DESCRIPTOR, 1, &engcfg);
cudnnBackendFinalize(plan);

int64_t workspaceSize;
cudnnBackendGetAttribute(plan, CUDNN_ATTR_EXECUTION_PLAN_WORKSPACE_SIZE,
        CUDNN_TYPE_INT64, 1, NULL, &workspaceSize)

```

2. Create, set and finalize a variant pack descriptor.

```

void *dev_ptrs[3] = {xData, wData, yData}; // device pointer
int64_t uids[3] = {'x', 'w', 'y'};
Void *workspace;

cudnnBackendDescriptor_t varpack;
cudnnBackendCreateDescriptor(CUDNN_BACKEND_VARIANT_PACK_DESCRIPTOR, &varpack);
cudnnBackendSetAttribute(varpack, CUDNN_ATTR_VARIANT_PACK_DATA_POINTERS,
        CUDNN_TYPE_VOID_PTR, 3, dev_ptrs);
cudnnBackendSetAttribute(varpack, CUDNN_ATTR_VARIANT_PACK_UNIQUE_IDS,
        CUDNN_TYPE_INT64, 3, uids);
cudnnBackendSetAttribute(varpack, CUDNN_ATTR_VARIANT_PACK_WORKSPACE,
        CUDNN_TYPE_VOID_PTR, 1, &workspace);
cudnnBackendFinalize(varPack);

```

3. Execute the plan with a variant pack.

```

cudnnBackendExecute(handle, plan, varpack);

```

## Notice

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

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

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

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

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

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

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

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

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

## Arm

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

## HDMI

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

## BlackBerry/QNX

Copyright © 2020 BlackBerry Limited. All rights reserved.

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

## Google

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

## Trademarks

NVIDIA, the NVIDIA logo, and BlueField, CUDA, DALI, DRIVE, Hopper, JetPack, Jetson AGX Xavier, Jetson Nano, Maxwell, NGC, Nsight, Orin, Pascal, Quadro, Tegra, TensorRT, Triton, Turing and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

## Copyright

© 2014-2024 NVIDIA Corporation & affiliates. All rights reserved.

