CUDNN

DU-06702-001_v07 | May 2018

Developer Guide

TABLE OF CONTENTS

Chapter 1. OVERVIEW

NVIDIA® cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications:

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

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

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

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

Chapter 2. GENERAL DESCRIPTION

Basic concepts are described in this chapter.

2.1. Programming Model

The cuDNN Library exposes a Host API but assumes that for operations using the GPU, the necessary data is directly accessible from the device.

An application using cuDNN must initialize a handle to the library context by calling **`cudnnCreate()`**. This handle is explicitly passed to every subsequent library function that operates on GPU data. Once the application finishes using cuDNN, it can release the resources associated with the library handle using **`cudnnDestroy()`** . This approach allows the user to explicitly control the library's functioning when using multiple host threads, GPUs and CUDA Streams. For example, an application can use **`cudaSetDevice()`** to associate different devices with different host threads and in each of those host threads, use a unique cuDNN handle which directs library calls to the device associated with it. cuDNN library calls made with different handles will thus automatically run on different devices. The device associated with a particular cuDNN context is assumed to remain unchanged between the corresponding **`cudnnCreate()`** and **`cudnnDestroy()`** calls. In order for the cuDNN library to use a different device within the same host thread, the application must set the new device to be used by calling **`cudaSetDevice()`** and then create another cuDNN context, which will be associated with the new device, by calling **`cudnnCreate()`**.

2.2. Notation

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

`y = layerFunction(x, otherParams)`.

And during backpropagation

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

For convolution the notation is

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

where **w** is the matrix of filter weights, **x** is the previous layer's data (during inference), **y** is the next layer's data, **b** is the bias and ***** is the convolution operator. In backpropagation routines the parameters keep their meanings. **dx,dy,dw,db** always refer to the gradient of the final network error function with respect to a given parameter. So **dy** in all backpropagation routines always refers to error gradient backpropagated through the network computation graph so far. Similarly other parameters in more specialized layers, such as, for instance, **dMeans** or **dBnBias** refer to gradients of the loss function wrt those parameters.

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

2.3. Tensor Descriptor

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

- a dimension **dim** from 3 to 8
- a data type (32-bit floating point, 64 bit-floating point, 16 bit floating point...)
- **dim** integers defining the size of each dimension
- **dim** integers defining the stride of each dimension (e.g the number of elements to add to reach the next element from the same dimension)

The first two dimensions define respectively the batch size **n** and the number of features maps **c**. This tensor definition allows for example to have some dimensions overlapping each others within the same tensor by having the stride of one dimension smaller than the product of the dimension and the stride of the next dimension. In cuDNN, unless specified otherwise, all routines will support tensors with overlapping dimensions for forward pass input tensors, however, dimensions of the output tensors cannot overlap. Even though this tensor format supports negative strides (which can be useful for data mirroring), cuDNN routines do not support tensors with negative strides unless specified otherwise.

2.3.1. WXYZ Tensor Descriptor

Tensor descriptor formats are identified using acronyms, with each letter referencing a corresponding dimension. In this document, the usage of this terminology implies :

- all the strides are strictly positive

- the dimensions referenced by the letters are sorted in decreasing order of their respective strides

2.3.2. 4-D Tensor Descriptor

A 4-D Tensor descriptor is used to define the format for batches of 2D images with 4 letters : N,C,H,W for respectively the batch size, the number of feature maps, the height and the width. The letters are sorted in decreasing order of the strides. The commonly used 4-D tensor formats are :

- NCHW
- NHWC
- CHWN

2.3.3. 5-D Tensor Description

A 5-D Tensor descriptor is used to define the format of batch of 3D images with 5 letters : N,C,D,H,W for respectively the batch size, the number of feature maps, the depth, the height and the width. The letters are sorted in descreasing order of the strides. The commonly used 5-D tensor formats are called :

- NCDHW
- NDHWC
- CDHWN

2.3.4. Fully-packed tensors

A tensor is defined as **`XYZ-fully-packed`** if and only if :

- the number of tensor dimensions is equal to the number of letters preceding the **`fully-packed`** suffix.
- the stride of the i-th dimension is equal to the product of the (i+1)-th dimension by the (i+1)-th stride.
- the stride of the last dimension is 1.

2.3.5. Partially-packed tensors

The partially 'XYZ-packed' terminology only applies in a context of a tensor format described with a superset of the letters used to define a partially-packed tensor. A WXYZ tensor is defined as **`XYZ-packed`** if and only if :

- the strides of all dimensions NOT referenced in the -packed suffix are greater or equal to the product of the next dimension by the next stride.
- the stride of each dimension referenced in the -packed suffix in position i is equal to the product of the (i+1)-st dimension by the (i+1)-st stride.
- if last tensor's dimension is present in the -packed suffix, it's stride is 1.

For example a NHWC tensor WC-packed means that the c_stride is equal to 1 and w_stride is equal to c_dim x c_stride. In practice, the -packed suffix is usually with slowest changing dimensions of a tensor but it is also possible to refer to a NCHW tensor that is only N-packed.

2.3.6. Spatially packed tensors

Spatially-packed tensors are defined as partially-packed in spatial dimensions.

For example a spatially-packed 4D tensor would mean that the tensor is either NCHW HW-packed or CNHW HW-packed.

2.3.7. Overlapping tensors

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

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

2.4. Thread Safety

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

2.5. Reproducibility (determinism)

By design, most of cuDNN's routines from a given version generate the same bit-wise results across runs when executed on GPUs with the same architecture and the same number of SMs. However, bit-wise reproducibility is not guaranteed across versions, as the implementation of a given routine may change. With the current release, the following routines do not guarantee reproducibility because they use atomic operations:

- **`cudnnConvolutionBackwardFilter`** when **`CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0`** or **`CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3`** is used
- **`cudnnConvolutionBackwardData`** when **`CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`** is used
- **`cudnnPoolingBackward`** when **`CUDNN_POOLING_MAX`** is used
- **`cudnnSpatialTfSamplerBackward`**

2.6. Scaling parameters `alpha` and `beta`

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

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

2.7. Tensor Core Operations

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

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

The Convolution functions are:

- cudnnConvolutionForward
- cudnnConvolutionBackwardData
- cudnnConvolutionBackwardFilter

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

- cudnnSetConvolutionMathType is called on the appropriate convolution descriptor setting mathType to CUDNN_TENSOR_OP_MATH.
- cudnnConvolutionForward is called using algo = CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM or CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED; cudnnConvolutionBackwardData using algo = CUDNN_CONVOLUTION_BWD_DATA_ALGO_1 or CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED; and cudnnConvolutionBackwardFilter using algo = CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1 or CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED.

For algorithms other than *_ALGO_WINOGRAD_NONFUSED, the following are some of the requirements to run Tensor Core operations:

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

The RNN functions are:

- cudnnRNNForwardInference
- cudnnRNNForwardTraining
- cudnnRNNBackwardData
- cudnnRNNBackwardWeights

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

- cudnnSetRNNMatrixMathType is called on the appropriate RNN descriptor setting mathType to CUDNN_TENSOR_OP_MATH.
- All routines are called using algo = CUDNN_RNN_ALGO_STANDARD.
- Hidden State size, Input size and Batch size are all multiples of 8.

For all cases, the CUDNN_TENSOR_OP_MATH enumerator is an indicator that the use of Tensor Cores is permissible, but not required. cuDNN may prefer not to use Tensor Core Operations (for instance, when the problem size is not suited to Tensor Core acceleration), and instead use an alternative implementation based on regular floating point operations.

2.7.1. Tensor Core Operations Notes

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

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

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

2.8. GPU and driver requirements

cuDNN v7.0 supports NVIDIA GPUs of compute capability 3.0 and higher. For x86_64 platform, cuDNN v7.0 comes with two deliverables : one requires a NVIDIA Driver compatible with CUDA Toolkit 8.0, the other requires a NVIDIA Driver compatible with CUDA Toolkit 9.0.

2.9. Backward compatibility and deprecation policy

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

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

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

2.10. Grouped Convolutions

cuDNN supports Grouped Convolutions by setting GroupCount > 1 using cudnnSetConvolutionGroupCount(). In memory, all input/output tensors store all independent groups. In this way, all tensor descriptors must describe the size of the entire convolution (as opposed to specifying the sizes per group). See following dimensions/strides explaining how to run Grouped Convolutions for NCHW format for 2-D convolutions. Note that other formats and 3-D convolutions are supported (see associated Convolution API for info on support); the tensor stridings for group count of 1 should still work for any group count.

Note that the symbols "*" and "/" are used to indicate multiplication and division.

xDesc or dxDesc	wDesc or dwDesc	convDesc	yDesc or dyDesc
Dimensions: [batch_size, input_channels, x_height, x_width]	Dimensions: [output_channels, input_channels/ group_count, w_height, w_width]	GroupCount: group_count	Dimensions: [batch_size, output_channels, y_height, y_width]

xDesc or dxDesc	wDesc or dwDesc	convDesc	yDesc or dyDesc
Strides: [output_channels*x_height*x_width, x_height*x_width, x_width, 1]	Format: NCHW		Strides: [output_channels*y_height*y_width, y_height*y_width, y_width, 1]

Chapter 3. CUDNN DATATYPES REFERENCE

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

3.1. 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.

3.2. cudnnStatus_t

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

Values

`CUDNN_STATUS_SUCCESS`

The operation completed successfully.

`CUDNN_STATUS_NOT_INITIALIZED`

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

`CUDNN_STATUS_ALLOC_FAILED`

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

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

CUDNN_STATUS_BAD_PARAM

An incorrect value or parameter was passed to the function.

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

CUDNN_STATUS_ARCH_MISMATCH

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

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

CUDNN_STATUS_MAPPING_ERROR

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

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

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

CUDNN_STATUS_EXECUTION_FAILED

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

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

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

CUDNN_STATUS_INTERNAL_ERROR

An internal cuDNN operation failed.

CUDNN_STATUS_NOT_SUPPORTED

The functionality requested is not presently supported by cuDNN.

CUDNN_STATUS_LICENSE_ERROR

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

CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING

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

CUDNN_STATUS_RUNTIME_IN_PROGRESS

Some tasks in the user stream are not completed.

CUDNN_STATUS_RUNTIME_FP_OVERFLOW

Numerical overflow occurred during the GPU kernel execution.

3.3. cudnnTensorDescriptor_t

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

3.4. 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.

3.5. cudnnConvolutionDescriptor_t

cudnnConvolutionDescriptor_t is a pointer to an opaque structure holding the description of a convolution operation. **cudnnCreateConvolutionDescriptor()** is used to create one instance, and **cudnnSetConvolutionNdDescriptor()** or **cudnnSetConvolution2dDescriptor()** must be used to initialize this instance.

3.6. cudnnMathType_t

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

Values

CUDNN_DEFAULT_MATH

Tensor Core Operations are not used.

CUDNN_TENSOR_OP_MATH

The use of Tensor Core Operations is permitted.

3.7. 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

3.8. cudnnDeterminism_t

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

Values

CUDNN_NON_DETERMINISTIC

Results are not guaranteed to be reproducible

CUDNN_DETERMINISTIC

Results are guaranteed to be reproducible

3.9. cudnnActivationDescriptor_t

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

3.10. 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.

3.11. cudnnOpTensorOp_t

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

Values

CUDNN_OP_TENSOR_ADD

The operation to be performed is addition

CUDNN_OP_TENSOR_MUL

The operation to be performed is multiplication

CUDNN_OP_TENSOR_MIN

The operation to be performed is a minimum comparison

CUDNN_OP_TENSOR_MAX

The operation to be performed is a maximum comparison

CUDNN_OP_TENSOR_SQRT

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

CUDNN_OP_TENSOR_NOT

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

3.12. cudnnOpTensorDescriptor_t

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

3.13. cudnnReduceTensorOp_t

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

Values

CUDNN_REDUCE_TENSOR_ADD

The operation to be performed is addition

CUDNN_REDUCE_TENSOR_MUL

The operation to be performed is multiplication

CUDNN_REDUCE_TENSOR_MIN

The operation to be performed is a minimum comparison

CUDNN_REDUCE_TENSOR_MAX

The operation to be performed is a maximum comparison

CUDNN_REDUCE_TENSOR_AMAX

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

CUDNN_REDUCE_TENSOR_AVG

The operation to be performed is averaging

CUDNN_REDUCE_TENSOR_NORM1

The operation to be performed is addition of absolute values

CUDNN_REDUCE_TENSOR_NORM2

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

CUDNN_REDUCE_TENSOR_MUL_NO_ZEROS

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

3.14. cudnnReduceTensorIndices_t

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

Values

CUDNN_REDUCE_TENSOR_NO_INDICES

Do not compute indices

CUDNN_REDUCE_TENSOR_FLATTENED_INDICES

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

3.15. cudnnIndicesType_t

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

Values

CUDNN_32BIT_INDICES

Compute unsigned int indices

CUDNN_64BIT_INDICES

Compute unsigned long long indices

CUDNN_16BIT_INDICES

Compute unsigned short indices

CUDNN_8BIT_INDICES

Compute unsigned char indices

3.16. 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.

3.17. cudnnCTCLossDescriptor_t

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

3.18. cudnnDataType_t

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

Values

CUDNN_DATA_FLOAT

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

CUDNN_DATA_DOUBLE

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

CUDNN_DATA_HALF

The data is 16-bit floating point.

CUDNN_DATA_INT8

The data is 8-bit signed integer.

CUDNN_DATA_INT32

The data is 8-bit signed integer.

CUDNN_DATA_INT8x4

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

3.19. cudnnTensorFormat_t

cudnnTensorFormat_t is an enumerated type used by **cudnnSetTensor4dDescriptor()** to create a tensor with a pre-defined layout.

Values

CUDNN_TENSOR_NCHW

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

CUDNN_TENSOR_NHWC

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

CUDNN_TENSOR_NCHW_VECT_C

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

3.20. cudnnConvolutionMode_t

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

Values

CUDNN_CONVOLUTION

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

CUDNN_CROSS_CORRELATION

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

3.21. cudnnConvolutionFwdPreference_t

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

Values

CUDNN_CONVOLUTION_FWD_NO_WORKSPACE

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

CUDNN_CONVOLUTION_FWD_PREFER_FASTEST

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

CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT

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

3.22. cudnnConvolutionFwdAlgo_t

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

Values

CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_GEMM

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

CUDNN_CONVOLUTION_FWD_ALGO_DIRECT

This algorithm expresses the convolution as a direct convolution (e.g without implicitly or explicitly doing a matrix multiplication).

CUDNN_CONVOLUTION_FWD_ALGO_FFT

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

CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING

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

CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD

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

CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED

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

3.23. cudnnConvolutionFwdAlgoPerf_t

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

Data Members

cudnnConvolutionFwdAlgo_t algo

The algorithm run to obtain the associated performance metrics.

cudnnStatus_t status

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

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

float time

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

size_t memory

The workspace size (in bytes).

cudnnDeterminism_t determinism

The determinism of the algorithm.

cudnnMathType_t mathType

The math type provided to the algorithm.

int reserved[3]

Reserved space for future properties.

3.24. cudnnConvolutionBwdFilterPreference_t

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

Values

CUDNN_CONVOLUTION_BWD_FILTER_NO_WORKSPACE

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

CUDNN_CONVOLUTION_BWD_FILTER_PREFER_FASTEST

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

CUDNN_CONVOLUTION_BWD_FILTER_SPECIFY_WORKSPACE_LIMIT

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

3.25. cudnnConvolutionBwdFilterAlgo_t

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

Values

CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0

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

CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1

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

CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT

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

CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3

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

CUDNN_CONVOLUTION_BWD_FILTER_WINOGRAD_NONFUSED

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

CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING

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

3.26. cudnnConvolutionBwdFilterAlgoPerf_t

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

Data Members

cudnnConvolutionBwdFilterAlgo_t algo

The algorithm run to obtain the associated performance metrics.

cudnnStatus_t status

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

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

float time

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

size_t memory

The workspace size (in bytes).

cudnnDeterminism_t determinism

The determinism of the algorithm.

cudnnMathType_t mathType

The math type provided to the algorithm.

int reserved[3]

Reserved space for future properties.

3.27. cudnnConvolutionBwdDataPreference_t

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

Values

CUDNN_CONVOLUTION_BWD_DATA_NO_WORKSPACE

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

CUDNN_CONVOLUTION_BWD_DATA_PREFER_FASTEST

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

CUDNN_CONVOLUTION_BWD_DATA_SPECIFY_WORKSPACE_LIMIT

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

3.28. cudnnConvolutionBwdDataAlgo_t

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

Values

CUDNN_CONVOLUTION_BWD_DATA_ALGO_0

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

CUDNN_CONVOLUTION_BWD_DATA_ALGO_1

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

CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT

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

CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING

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

CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD

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

CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED

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

3.29. cudnnConvolutionBwdDataAlgoPerf_t

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

Data Members

cudnnConvolutionBwdDataAlgo_t algo

The algorithm run to obtain the associated performance metrics.

cudnnStatus_t status

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

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

float time

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

size_t memory

The workspace size (in bytes).

cudnnDeterminism_t determinism

The determinism of the algorithm.

cudnnMathType_t mathType

The math type provided to the algorithm.

int reserved[3]

Reserved space for future properties.

3.30. 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`**

3.31. cudnnSoftmaxMode_t

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

Values

`CUDNN_SOFTMAX_MODE_INSTANCE`

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

`CUDNN_SOFTMAX_MODE_CHANNEL`

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

3.32. cudnnPoolingMode_t

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

Values

`CUDNN_POOLING_MAX`

The maximum value inside the pooling window is used.

`CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING`

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

CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING

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

CUDNN_POOLING_MAX_DETERMINISTIC

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

3.33. cudnnActivationMode_t

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

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

3.34. cudnnLRNMode_t

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

Values

CUDNN_LRN_CROSS_CHANNEL_DIM1

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

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

3.36. 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 non-convolutional network layers. In this mode bnBias and bnScale tensor dimensions 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 bnBias, bnScale tensor dimensions are 1xCx1x1.

CUDNN_BATCHNORM_SPATIAL_PERSISTENT

This mode is similar to CUDNN_BATCHNORM_SPATIAL but it can be faster for some tasks. An optimized path may be selected for CUDNN_DATA_FLOAT and CUDNN_DATA_HALF data types, compute capability 6.0 or higher, and the following two batch normalization API calls: cudnnBatchNormalizationForwardTraining, and cudnnBatchNormalizationBackward. In the latter case savedMean and savedInvVariance arguments should not be NULL. The CUDNN_BATCHNORM_SPATIAL_PERSISTENT mode may use scaled atomic integer reduction that is deterministic but imposes some restrictions on the input data range. When a numerical overflow occurs, a NaN (not-a-number) floating point value is written to the output buffer. The user can invoke cudnnQueryRuntimeError to check if a numerical overflow occurred in this mode.

3.37. cudnnRNNDescriptor_t

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

3.38. cudnnPersistentRNNPlan_t

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

3.39. 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 matrices **W, R** and biases b_W, b_R from the following equation:

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

Where **ReLU(x) = max(x, 0)**.

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 matrices **W, R** and biases b_W, b_R from the following equation:

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

Where **tanh** is the hyperbolic tangent function.

CUDNN_LSTM

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

In the forward pass the output 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 matrices **W, R** and biases b_W, b_R from the following equations:

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

```
c'_t = tanh(W_c x_t + R_c h_t-1 + b_Wc + b_Rc)
c_t = f_t∘c_t-1 + i_t∘c'_t
h_t = o_t∘tanh(c_t)
```

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

CUDNN_GRU

A three-gate network consisting of Gated Recurrent Units.

In the forward pass the output 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 biases b_W, b_R from the following equations:

```
i_t = σ(W_i x_t + R_i h_t-1 + b_Wi + b_Ru)
r_t = σ(W_r x_t + R_r h_t-1 + b_Wr + b_Rr)
h'_t = tanh(W_h x_t + r_t∘(R_h h_t-1 + b_Rh) + b_Wh)
h_t = (1 - i_t)∘h'_t + i_t∘h_t-1
```

Where **σ** is the sigmoid operator: **σ(x) = 1 / (1 + e^-x)**, ∘ represents a point-wise multiplication and **tanh** is the hyperbolic tangent function. i_t, r_t, h'_t represent the input, reset, new gates respectively.

3.40. cudnnDirectionMode_t

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

Values

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

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

3.41. cudnnRNNInputMode_t

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

Values

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

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

3.42. cudnnRNNAlgo_t

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

Values

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

CUDNN_RNN_ALGO_PERSIST_STATIC

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

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

CUDNN_RNN_ALGO_PERSIST_DYNAMIC

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

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

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

3.43. cudnnCTCLossAlgo_t

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

Values

CUDNN_CTC_LOSS_ALGO_DETERMINISTIC

Results are guaranteed to be reproducible

CUDNN_CTC_LOSS_ALGO_NON_DETERMINISTIC

Results are not guaranteed to be reproducible

3.44. cudnnDropoutDescriptor_t

cudnnDropoutDescriptor_t is a pointer to an opaque structure holding the description of a dropout operation. **cudnnCreateDropoutDescriptor()** is used to create one instance, **cudnnSetDropoutDescriptor()** is used to initialize this instance, **cudnnDestroyDropoutDescriptor()** is used to destroy this instance, **cudnnGetDropoutDescriptor()** is used to query fields of a previously initialized instance, **cudnnRestoreDropoutDescriptor()** is used to restore an instance to a previously saved off state.

3.45. cudnnSpatialTransformerDescriptor_t

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

3.46. 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.

3.47. 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.

Chapter 4. CUDNN API REFERENCE

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

4.1. cudnnGetVersion

```
size_t cudnnGetVersion()
```

This function returns the version number of the cuDNN Library. It returns the **CUDNN_VERSION** define present in the cudnn.h header file. Starting with release R2, the routine can be used to identify dynamically the current cuDNN Library used by the application. The define **CUDNN_VERSION** can be used to have the same application linked against different cuDNN versions using conditional compilation statements.

4.2. cudnnGetCudartVersion

```
size_t cudnnGetCudartVersion()
```

The same version of a given cuDNN library can be compiled against different CUDA Toolkit versions. This routine returns the CUDA Toolkit version that the currently used cuDNN library has been compiled against.

4.3. cudnnGetProperty

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

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

Parameters

type

Input. Enumerated type that instructs the function to report the numerical value of the cuDNN major version, minor version, or the patch level.

value

Output. Host pointer where the version information should be written.

Returns

CUDNN_STATUS_INVALID_VALUE

Invalid value of the **type** argument.

CUDNN_STATUS_SUCCESS

Version information was stored successfully at the provided address.

4.4. cudnnGetErrorString

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

This function converts the cuDNN status code to a NUL terminated (ASCIIZ) static string. For example, when the input argument is CUDNN_STATUS_SUCCESS, the returned string is "CUDNN_STATUS_SUCCESS". When an invalid status value is passed to the function, the returned string is "CUDNN_UNKNOWN_STATUS".

Parameters

status

Input. cuDNN enumerated status code.

Returns

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

4.5. cudnnQueryRuntimeError

```
cudnnStatus_t cudnnQueryRuntimeError(
    cudnnHandle_t            handle,
    cudnnStatus_t           *rstatus,
    cudnnErrQueryMode_t      mode,
    cudnnRuntimeTag_t       *tag)
```

cuDNN library functions perform extensive input argument checking before launching GPU kernels. The last step is to verify that the GPU kernel actually started. When a kernel fails to start, CUDNN_STATUS_EXECUTION_FAILED is returned by the corresponding API call. Typically, after a GPU kernel starts, no runtime checks are performed by the kernel itself -- numerical results are simply written to output buffers.

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

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

CUDNN_ERRQUERY_RAWCODE reads the error storage location regardless of the kernel completion status. The kernel might not even started and the error storage (allocated per cuDNN handle) might be used by an earlier call.

CUDNN_ERRQUERY_NONBLOCKING checks if all tasks in the user stream completed. The cudnnQueryRuntimeError function will return immediately and report CUDNN_STATUS_RUNTIME_IN_PROGRESS in 'rstatus' if some tasks in the user stream are pending. Otherwise, the function will copy the remote kernel error code to 'rstatus'.

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

CUDNN_ERRQUERY_NONBLOCKING and CUDNN_ERRQUERY_BLOCKING modes should not be used when the user stream is changed in the cuDNN handle, i.e., cudnnSetStream is invoked between functions that report runtime kernel errors and the cudnnQueryRuntimeError function.

The remote error status reported in rstatus can be set to: CUDNN_STATUS_SUCCESS, CUDNN_STATUS_RUNTIME_IN_PROGRESS, or CUDNN_STATUS_RUNTIME_FP_OVERFLOW. The remote kernel error is automatically cleared by cudnnQueryRuntimeError.

> The cudnnQueryRuntimeError function should be used in conjunction with cudnnBatchNormalizationForwardTraining and cudnnBatchNormalizationBackward when the cudnnBatchNormMode_t argument is CUDNN_BATCHNORM_SPATIAL_PERSISTENT.

Parameters

handle

Input. Handle to a previously created cuDNN context.

rstatus

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

mode

Input. Remote error query mode.

tag

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

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

Returns

CUDNN_STATUS_SUCCESS

No errors detected (rstatus holds a valid value).

CUDNN_STATUS_BAD_PARAM

Invalid input argument.

CUDNN_STATUS_INTERNAL_ERROR

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

CUDNN_STATUS_MAPPING_ERROR

Device cannot access zero-copy memory to report kernel errors.

4.6. cudnnCreate

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

This function initializes the cuDNN library and creates a handle to an opaque structure holding the cuDNN library context. It allocates hardware resources on the host and device and must be called prior to making any other cuDNN library calls. The cuDNN library handle is tied to the current CUDA device (context). To use the library on multiple devices, one cuDNN handle needs to be created for each device. For a given device, multiple cuDNN handles with different configurations (e.g., different current CUDA streams) may be created. Because **cudnnCreate** allocates some internal resources, the release of those resources by calling **cudnnDestroy** will implicitly call **cudaDeviceSynchronize**; therefore, the recommended best practice is to call **cudnnCreate/cudnnDestroy** outside of performance-critical code paths. For multithreaded applications that use the same device from different threads, the recommended programming model is to create one (or a few, as is convenient) cuDNN handle(s) per thread and use that cuDNN handle for the entire life of the thread.

Parameters

handle

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

Returns

CUDNN_STATUS_BAD_PARAM

Invalid (NULL) input pointer supplied.

CUDNN_STATUS_NOT_INITIALIZED

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

CUDNN_STATUS_ARCH_MISMATCH

NVIDIA GPU architecture is too old.

CUDNN_STATUS_ALLOC_FAILED

Host memory allocation failed.

CUDNN_STATUS_INTERNAL_ERROR

CUDA resource allocation failed.

CUDNN_STATUS_LICENSE_ERROR

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

CUDNN_STATUS_SUCCESS

cuDNN handle was created successfully.

4.7. cudnnDestroy

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

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

Parameters

handle

Input. Pointer to the cuDNN handle to be destroyed.

Returns

CUDNN_STATUS_SUCCESS

The cuDNN context destruction was successful.

CUDNN_STATUS_BAD_PARAM

Invalid (NULL) pointer supplied.

4.8. cudnnSetStream

```
cudnnStatus_t cudnnSetStream(
    cudnnHandle_t   handle,
    cudaStream_t    streamId)
```

This function sets the user's CUDA stream in the cuDNN handle. The new stream will be used to launch cuDNN GPU kernels or to synchronize to this stream when cuDNN kernels are launched in the internal streams. If the cuDNN library stream is not set, all kernels use the default (NULL) stream. Setting the user stream in the cuDNN handle guarantees the issue-order execution of cuDNN calls and other GPU kernels launched in the same stream.

Parameters

handle

Input. Pointer to the cuDNN handle.

streamID

Input. New CUDA stream to be written to the cuDNN handle.

Returns

CUDNN_STATUS_BAD_PARAM

Invalid (NULL) handle.

CUDNN_STATUS_MAPPING_ERROR

Mismatch between the user stream and the cuDNN handle context.

CUDNN_STATUS_SUCCESS

The new stream was set successfully.

4.9. cudnnGetStream

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

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

Parameters

handle

Input. Pointer to the cuDNN handle.

streamID

Output. Pointer where the current CUDA stream from the cuDNN handle should be stored.

Returns

CUDNN_STATUS_BAD_PARAM

Invalid (NULL) handle.

CUDNN_STATUS_SUCCESS

The stream identifier was retrieved successfully.

4.10. cudnnCreateTensorDescriptor

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

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

Parameters

tensorDesc

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

Returns

CUDNN_STATUS_BAD_PARAM

Invalid input argument.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

CUDNN_STATUS_SUCCESS

The object was created successfully.

4.11. cudnnSetTensor4dDescriptor

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

This function initializes a previously created generic Tensor descriptor object into a 4D tensor. The strides of the four dimensions are inferred from the format parameter and set in such a way that the data is contiguous in memory with no padding between dimensions.

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

Parameters

tensorDesc

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

format

Input. Type of format.

datatype

Input. Data type.

n

Input. Number of images.

c

Input. Number of feature maps per image.

h

Input. Height of each feature map.

w

Input. Width of each feature map.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_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 maximim limit of 2 Giga-elements.

4.12. cudnnSetTensor4dDescriptorEx

```
cudnnStatus_t cudnnSetTensor4dDescriptorEx(
    cudnnTensorDescriptor_t     tensorDesc,
    cudnnDataType_t             dataType,
    int                         n,
    int                         c,
    int                         h,
    int                         w,
    int                         nStride,
    int                         cStride,
    int                         hStride,
    int                         wStride)
```

This function initializes a previously created generic Tensor descriptor object into a 4D tensor, similarly to **cudnnSetTensor4dDescriptor** but with the strides explicitly passed as parameters. This can be used to lay out the 4D tensor in any order or simply to define gaps between dimensions.

At present, some cuDNN routines have limited support for strides; Those routines will return CUDNN_STATUS_NOT_SUPPORTED if a Tensor4D object with an unsupported stride is used. **cudnnTransformTensor** can be used to convert the data to a supported layout.

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

Parameters

tensorDesc

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

datatype

Input. Data type.

n

Input. Number of images.

c

Input. Number of feature maps per image.

h

Input. Height of each feature map.

w

Input. Width of each feature map.

nStride

Input. Stride between two consecutive images.

cStride

Input. Stride between two consecutive feature maps.

hStride

Input. Stride between two consecutive rows.

wStride

Input. Stride between two consecutive columns.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_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 maximim limit of 2 Giga-elements.

4.13. cudnnGetTensor4dDescriptor

```
cudnnStatus_t cudnnGetTensor4dDescriptor(
    cudnnTensorDescriptor_t  tensorDesc,
    cudnnDataType_t         *dataType,
    int                     *n,
    int                     *c,
    int                     *h,
    int                     *w,
    int                     *nStride,
    int                     *cStride,
    int                     *hStride,
    int                     *wStride)
```

This function queries the parameters of the previouly initialized Tensor4D descriptor object.

Parameters

tensorDesc

Input. Handle to a previously insitialized tensor descriptor.

datatype

Output. Data type.

n

Output. Number of images.

c

Output. Number of feature maps per image.

h

Output. Height of each feature map.

w

Output. Width of each feature map.

nStride

Output. Stride between two consecutive images.

cStride

Output. Stride between two consecutive feature maps.

hStride

Output. Stride between two consecutive rows.

wStride

Output. Stride between two consecutive columns.

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

Returns

CUDNN_STATUS_SUCCESS

The operation succeeded.

4.14. cudnnSetTensorNdDescriptor

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

This function initializes a previously created generic Tensor descriptor object.

> The total size of a tensor including the potential padding between dimensions is limited to 2 Giga-elements of type **`datatype`**. Tensors are restricted to having at least 4 dimensions, and at most CUDNN_DIM_MAX dimensions (defined in cudnn.h). When working with lower dimensional data, it is recommended that the user create a 4D tensor, and set the size along unused dimensions to 1.

Parameters

tensorDesc

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

datatype

Input. Data type.

nbDims

Input. Dimension of the tensor.

dimA

Input. Array of dimension **`nbDims`** that contain the size of the tensor for every dimension. Size along unused dimensions should be set to 1.

strideA

Input. Array of dimension **`nbDims`** that contain the stride of the tensor for every dimension.

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the elements of the array **`dimA`** was negative or zero, or **`dataType`** has an invalid enumerant value.

`CUDNN_STATUS_NOT_SUPPORTED`

The parameter **`nbDims`** is outside the range [4, CUDNN_DIM_MAX], or the total size of the tensor descriptor exceeds the maximim limit of 2 Giga-elements.

4.15. cudnnGetTensorNdDescriptor

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

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

Parameters

tensorDesc

Input. Handle to a previously initialized tensor descriptor.

nbDimsRequested

Input. Number of dimensions to extract from a given tensor descriptor. It is also the minimum size of the arrays **dimA** and **strideA**. If this number is greater than the resulting nbDims[0], only nbDims[0] dimensions will be returned.

datatype

Output. Data type.

nbDims

Output. Actual number of dimensions of the tensor will be returned in nbDims[0].

dimA

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

strideA

Input. Array of dimension of at least **nbDimsRequested** that will be filled with the strides from the provided tensor descriptor.

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

Returns

CUDNN_STATUS_SUCCESS

The results were returned successfully.

CUDNN_STATUS_BAD_PARAM

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

4.16. cudnnGetTensorSizeInBytes

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

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

Parameters

tensorDesc

Input. Handle to a previously initialized tensor descriptor.

size

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

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

Returns

CUDNN_STATUS_SUCCESS

The results were returned successfully.

4.17. cudnnDestroyTensorDescriptor

```
cudnnStatus_t cudnnDestroyTensorDescriptor(cudnnTensorDescriptor_t tensorDesc)
```

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

Parameters

tensorDesc

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

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.18. cudnnTransformTensor

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows: dstValue = alpha[0]*srcValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to a previously initialized tensor descriptor.

x

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

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

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

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.19. cudnnAddTensor

```
cudnnStatus_t cudnnAddTensor(
    cudnnHandle_t                     handle,
    const void                       *alpha,
    const cudnnTensorDescriptor_t     aDesc,
    const void                       *A,
    const void                       *beta,
    const cudnnTensorDescriptor_t     cDesc,
    void                             *C)
```

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

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. Please refer to this section for additional details.

aDesc

Input. Handle to a previously initialized tensor descriptor.

A

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

cDesc

Input. Handle to a previously initialized tensor descriptor.

C

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function executed successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

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

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.20. cudnnOpTensor

```
cudnnStatus_t cudnnOpTensor(
    cudnnHandle_t                     handle,
    const cudnnOpTensorDescriptor_t   opTensorDesc,
    const void                       *alpha1,
    const cudnnTensorDescriptor_t     aDesc,
    const void                       *A,
    const void                       *alpha2,
    const cudnnTensorDescriptor_t     bDesc,
    const void                       *B,
    const void                       *beta,
    const cudnnTensorDescriptor_t     cDesc,
    void                             *C)
```

This function implements the equation C = op (alpha1[0] * A, alpha2[0] * B) + beta[0] * C, given tensors A, B, and C and scaling factors alpha1, alpha2, and beta. The op to use is indicated by the descriptor **`opTensorDesc`**. Currently-supported ops are listed by the **`cudnnOpTensorOp_t`** enum.

Each dimension of the input tensor **`A`** must match the corresponding dimension of the destination tensor **`C`**, and each dimension of the input tensor **`B`** must match the corresponding dimension of the destination tensor **`C`** or must be equal to 1. In the latter case, the same value from the input tensor **`B`** for those dimensions will be used to blend into the **`C`** tensor.

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

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

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

opTensorDesc

Input. Handle to a previously initialized op tensor descriptor.

alpha1, alpha2, beta

Input. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as indicated by the above op equation. Please refer to this section for additional details.

aDesc, bDesc, cDesc

Input. Handle to a previously initialized tensor descriptor.

A, B

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

C

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

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

Returns

CUDNN_STATUS_SUCCESS

The function executed successfully.

CUDNN_STATUS_NOT_SUPPORTED

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

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

CUDNN_STATUS_BAD_PARAM

The data type of the destination tensor **C** is unrecognized or the conditions in the above paragraphs are unmet.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.21. cudnnReduceTensor

```
cudnnStatus_t cudnnReduceTensor(
    cudnnHandle_t                          handle,
    const cudnnReduceTensorDescriptor_t    reduceTensorDesc,
    void                                  *indices,
    size_t                                 indicesSizeInBytes,
    void                                  *workspace,
    size_t                                 workspaceSizeInBytes,
    const void                            *alpha,
    const cudnnTensorDescriptor_t          aDesc,
    const void                            *A,
    const void                            *beta,
    const cudnnTensorDescriptor_t          cDesc,
    void                                  *C)
```

This function reduces tensor A by implementing the equation C = alpha * reduce op (A) + beta * C, given tensors A and C and scaling factors alpha and beta. The reduction op to use is indicated by the descriptor **reduceTensorDesc**. Currently-supported ops are listed by the **cudnnReduceTensorOp_t** enum.

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

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

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

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

The half and int8 data types may be mixed with the float data types. In these cases, the computation enum of **reduceTensorDesc** is required to be of type float.

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

reduceTensorDesc

Input. Handle to a previously initialized reduce tensor descriptor.

indices

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

indicesSizeInBytes

Input. Size of the above previously allocated space.

workspace

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

workspaceSizeInBytes

Input. Size of the above previously allocated space.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as indicated by the above op equation. Please refer to this section for additional details.

aDesc, cDesc

Input. Handle to a previously initialized tensor descriptor.

A

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

C

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function executed successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

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

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

`CUDNN_STATUS_BAD_PARAM`

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

`CUDNN_INVALID_VALUE`

The allocations for the indices or workspace are insufficient.

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.22. cudnnSetTensor

```
cudnnStatus_t cudnnSetTensor(
    cudnnHandle_t                   handle,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y,
    const void                     *valuePtr)
```

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

valuePtr

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

one of the provided pointers is nil

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.23. cudnnScaleTensor

```
cudnnStatus_t cudnnScaleTensor(
    cudnnHandle_t                   handle,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y,
    const void                     *alpha)
```

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

alpha

Input. Pointer in Host memory to a single value that all elements of the tensor will be scaled with. Please refer to this section for additional details.

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

one of the provided pointers is nil

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.24. cudnnCreateFilterDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.25. cudnnSetFilter4dDescriptor

```
cudnnStatus_t cudnnSetFilter4dDescriptor(
    cudnnFilterDescriptor_t    filterDesc,
    cudnnDataType_t            dataType,
    cudnnTensorFormat_t        format,
    int                        k,
```

```
    int                         c,
    int                         h,
    int                         w)
```

This function initializes a previously created filter descriptor object into a 4D filter. Filters layout must be contiguous in memory.

Tensor format CUDNN_TENSOR_NHWC has limited support in **`cudnnConvolutionForward`**, **`cudnnConvolutionBackwardData`** and **`cudnnConvolutionBackwardFilter`**; please refer to each function's documentation for more information.

Parameters

filterDesc

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

datatype

Input. Data type.

format

Input. Type of format.

k

Input. Number of output feature maps.

c

Input. Number of input feature maps.

h

Input. Height of each filter.

w

Input. Width of each filter.

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

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

4.26. cudnnGetFilter4dDescriptor

```
cudnnStatus_t cudnnGetFilter4dDescriptor(
    cudnnFilterDescriptor_t     filterDesc,
    cudnnDataType_t            *dataType,
    cudnnTensorFormat_t        *format,
    int                        *k,
    int                        *c,
    int                        *h,
```

```
    int                         *w)
```

This function queries the parameters of the previouly initialized filter descriptor object.

Parameters

filterDesc

Input. Handle to a previously created filter descriptor.

datatype

Output. Data type.

format

Output. Type of format.

k

Output. Number of output feature maps.

c

Output. Number of input feature maps.

h

Output. Height of each filter.

w

Output. Width of each filter.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

4.27. cudnnSetFilterNdDescriptor

```
cudnnStatus_t cudnnSetFilterNdDescriptor(
    cudnnFilterDescriptor_t filterDesc,
    cudnnDataType_t         dataType,
    cudnnTensorFormat_t     format,
    int                     nbDims,
    int                     filterDimA[])
```

This function initializes a previously created filter descriptor object. Filters layout must be contiguous in memory.

Tensor format CUDNN_TENSOR_NHWC has limited support in **cudnnConvolutionForward**, **cudnnConvolutionBackwardData** and **cudnnConvolutionBackwardFilter**; please refer to each function's documentation for more information.

Parameters

filterDesc

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

datatype

Input. Data type.

format

Input. Type of format.

nbDims

Input. Dimension of the filter.

filterDimA

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

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

`CUDNN_STATUS_NOT_SUPPORTED`

the parameter **`nbDims`** exceeds CUDNN_DIM_MAX.

4.28. cudnnGetFilterNdDescriptor

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

This function queries a previously initialized filter descriptor object.

Parameters

wDesc

Input. Handle to a previously initialized filter descriptor.

nbDimsRequested

Input. Dimension of the expected filter descriptor. It is also the minimum size of the arrays **`filterDimA`** in order to be able to hold the results

datatype

Output. Data type.

format

Output. Type of format.

nbDims

Output. Actual dimension of the filter.

filterDimA

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

The parameter **`nbDimsRequested`** is negative.

4.29. cudnnDestroyFilterDescriptor

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

This function destroys a previously created Tensor4D descriptor object.

Returns

`CUDNN_STATUS_SUCCESS`

The object was destroyed successfully.

4.30. cudnnCreateConvolutionDescriptor

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was created successfully.

`CUDNN_STATUS_ALLOC_FAILED`

The resources could not be allocated.

4.31. cudnnSetConvolutionMathType

```
cudnnStatus_t cudnnSetConvolutionMathType(
    cudnnConvolutionDescriptor_t    convDesc,
    cudnnMathType_t                 mathType)
```

This function allows the user to specify whether or not the use of tensor op is permitted in library routines associated with a given convolution descriptor.

Returns

CUDNN_STATUS_SUCCESS

The math type was was set successfully.

CUDNN_STATUS_BAD_PARAM

Either an invalid convolution descriptor was provided or an invalid math type was specified.

4.32. cudnnGetConvolutionMathType

```
cudnnStatus_t cudnnGetConvolutionMathType(
    cudnnConvolutionDescriptor_t    convDesc,
    cudnnMathType_t                *mathType)
```

This function returns the math type specified in a given convolution descriptor.

Returns

CUDNN_STATUS_SUCCESS

The math type was returned successfully.

CUDNN_STATUS_BAD_PARAM

An invalid convolution descriptor was provided.

4.33. cudnnSetConvolutionGroupCount

```
cudnnStatus_t cudnnSetConvolutionGroupCount(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             groupCount)
```

This function allows the user to specify the number of groups to be used in the associated convolution.

Returns

CUDNN_STATUS_SUCCESS

The group count was set successfully.

CUDNN_STATUS_BAD_PARAM

An invalid convolution descriptor was provided

4.34. cudnnGetConvolutionGroupCount

```
cudnnStatus_t cudnnGetConvolutionGroupCount(
    cudnnConvolutionDescriptor_t    convDesc,
    int                            *groupCount)
```

This function returns the group count specified in the given convolution descriptor.

Returns

CUDNN_STATUS_SUCCESS

The group count was returned successfully.

CUDNN_STATUS_BAD_PARAM

An invalid convolution descriptor was provided.

4.35. cudnnSetConvolution2dDescriptor

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             pad_h,
    int                             pad_w,
    int                             u,
    int                             v,
    int                             dilation_h,
    int                             dilation_w,
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 computeType)
```

This function initializes a previously created convolution descriptor object into a 2D correlation. This function assumes that the tensor and filter descriptors corresponds to the formard convolution path and checks if their settings are valid. That same convolution descriptor can be reused in the backward path provided it corresponds to the same layer.

Parameters

convDesc

Input/Output. Handle to a previously created convolution descriptor.

pad_h

Input. zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

pad_w

Input. zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

u

Input. Vertical filter stride.

v

Input. Horizontal filter stride.

dilation_h

Input. Filter height dilation.

dilation_w

Input. Filter width dilation.

mode

Input. Selects between **CUDNN_CONVOLUTION** and **CUDNN_CROSS_CORRELATION**.

computeType

Input. compute precision.

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The descriptor **`convDesc`** is nil.
- One of the parameters **`pad_h,pad_w`** is strictly negative.
- One of the parameters **`u,v`** is negative or zero.
- One of the parameters **`dilation_h,dilation_w`** is negative or zero.
- The parameter **`mode`** has an invalid enumerant value.

4.36. cudnnGetConvolution2dDescriptor

```
cudnnStatus_t cudnnGetConvolution2dDescriptor(
    const cudnnConvolutionDescriptor_t  convDesc,
    int                                *pad_h,
    int                                *pad_w,
    int                                *u,
    int                                *v,
    int                                *dilation_h,
    int                                *dilation_w,
    cudnnConvolutionMode_t             *mode,
    cudnnDataType_t                    *computeType)
```

This function queries a previously initialized 2D convolution descriptor object.

Parameters

convDesc

Input/Output. Handle to a previously created convolution descriptor.

pad_h

Output. zero-padding height: number of rows of zeros implicitly concatenated onto the top and onto the bottom of input images.

pad_w

Output. zero-padding width: number of columns of zeros implicitly concatenated onto the left and onto the right of input images.

u

Output. Vertical filter stride.

v

Output. Horizontal filter stride.

dilation_h

Output. Filter height dilation.

dilation_w

Output. Filter width dilation.

mode

Output. Convolution mode.

computeType

Output. Compute precision.

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

Returns

CUDNN_STATUS_SUCCESS

The operation was successful.

CUDNN_STATUS_BAD_PARAM

The parameter **convDesc** is nil.

4.37. cudnnGetConvolution2dForwardOutputDim

```
cudnnStatus_t cudnnGetConvolution2dForwardOutputDim(
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       inputTensorDesc,
    const cudnnFilterDescriptor_t       filterDesc,
    int                                *n,
    int                                *c,
    int                                *h,
    int                                *w)
```

This function returns the dimensions of the resulting 4D tensor of a 2D convolution, given the convolution descriptor, the input tensor descriptor and the filter descriptor This function can help to setup the output tensor and allocate the proper amount of memory prior to launch the actual convolution.

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

```
    outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) )/
convolutionStride;
```

The dimensions provided by this routine must be strictly respected when calling **cudnnConvolutionForward()** or **cudnnConvolutionBackwardBias()**. Providing a smaller or larger output tensor is not supported by the convolution routines.

Parameters

convDesc

Input. Handle to a previously created convolution descriptor.

inputTensorDesc

Input. Handle to a previously initialized tensor descriptor.

filterDesc

Input. Handle to a previously initialized filter descriptor.

n

Output. Number of output images.

c

Output. Number of output feature maps per image.

h

Output. Height of each output feature map.

w

Output. Width of each output feature map.

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

Returns

`CUDNN_STATUS_BAD_PARAM`

One or more of the descriptors has not been created correctly or there is a mismatch between the feature maps of **`inputTensorDesc`** and **`filterDesc`**.

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

4.38. cudnnSetConvolutionNdDescriptor

```
cudnnStatus_t cudnnSetConvolutionNdDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             arrayLength,
    int                             padA[],
    int                             filterStrideA[],
    int                             dilationA[],
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 dataType)
```

This function initializes a previously created generic convolution descriptor object into a n-D correlation. That same convolution descriptor can be reused in the backward path provided it corresponds to the same layer. The convolution computation will done in the specified **`dataType`**, which can be potentially different from the input/output tensors.

Parameters

convDesc

Input/Output. Handle to a previously created convolution descriptor.

arrayLength

Input. Dimension of the convolution.

padA

Input. Array of dimension **arrayLength** containing the zero-padding size for each dimension. For every dimension, the padding represents the number of extra zeros implicitly concatenated at the start and at the end of every element of that dimension .

filterStrideA

Input. Array of dimension **arrayLength** containing the filter stride for each dimension. For every dimension, the fitler stride represents the number of elements to slide to reach the next start of the filtering window of the next point.

dilationA

Input. Array of dimension **arrayLength** containing the dilation factor for each dimension.

mode

Input. Selects between **CUDNN_CONVOLUTION** and **CUDNN_CROSS_CORRELATION**.

datatype

Input. Selects the datatype in which the computation will be done.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **convDesc** is nil.
- The **arrayLengthRequest** is negative.
- The enumerant **mode** has an invalid value.
- The enumerant **datatype** has an invalid value.
- One of the elements of **padA** is strictly negative.
- One of the elements of **strideA** is negative or zero.
- One of the elements of **dilationA** is negative or zero.

CUDNN_STATUS_NOT_SUPPORTED

At least one of the following conditions are met:

- The **arrayLengthRequest** is greater than CUDNN_DIM_MAX.

4.39. cudnnGetConvolutionNdDescriptor

```
cudnnStatus_t cudnnGetConvolutionNdDescriptor(
    const cudnnConvolutionDescriptor_t  convDesc,
    int                                 arrayLengthRequested,
    int                                *arrayLength,
    int                                 padA[],
```

```
    int                                    filterStrideA[],
    int                                    dilationA[],
    cudnnConvolutionMode_t                *mode,
    cudnnDataType_t                       *dataType)
```

This function queries a previously initialized convolution descriptor object.

Parameters

convDesc

Input/Output. Handle to a previously created convolution descriptor.

arrayLengthRequested

Input. Dimension of the expected convolution descriptor. It is also the minimum size of the arrays **padA**, **filterStrideA** and **dilationA** in order to be able to hold the results

arrayLength

Output. Actual dimension of the convolution descriptor.

padA

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

filterStrideA

Output. Array of dimension of at least **arrayLengthRequested** that will be filled with the filter stride from the provided convolution descriptor.

dilationA

Output. Array of dimension of at least **arrayLengthRequested** that will be filled with the dilation parameters from the provided convolution descriptor.

mode

Output. Convolution mode of the provided descriptor.

datatype

Output. datatype of the provided descriptor.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **convDesc** is nil.
- The **arrayLengthRequest** is negative.

CUDNN_STATUS_NOT_SUPPORTED

The **arrayLengthRequest** is greater than CUDNN_DIM_MAX

4.40. cudnnGetConvolutionNdForwardOutputDim

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

This function returns the dimensions of the resulting n-D tensor of a **nbDims-2**-D convolution, given the convolution descriptor, the input tensor descriptor and the filter descriptor This function can help to setup the output tensor and allocate the proper amount of memory prior to launch the actual convolution.

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

```
    outputDim = 1 + ( inputDim + 2*pad - (((filterDim-1)*dilation)+1) )/
convolutionStride;
```

The dimensions provided by this routine must be strictly respected when calling **cudnnConvolutionForward()** or **cudnnConvolutionBackwardBias()**. Providing a smaller or larger output tensor is not supported by the convolution routines.

Parameters

convDesc

Input. Handle to a previously created convolution descriptor.

inputTensorDesc

Input. Handle to a previously initialized tensor descriptor.

filterDesc

Input. Handle to a previously initialized filter descriptor.

nbDims

Input. Dimension of the output tensor

tensorOuputDimA

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

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

Returns

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- One of the parameters **convDesc**, **inputTensorDesc,** and **filterDesc,** is nil
- The dimension of the filter descriptor **filterDesc** is different from the dimension of input tensor descriptor **inputTensorDesc**.

- The dimension of the convolution descriptor is different from the dimension of input tensor descriptor **inputTensorDesc** -2 .
- The features map of the filter descriptor **filterDesc** is different from the one of input tensor descriptor **inputTensorDesc**.
- The size of the dilated filter **filterDesc** is larger than the padded sizes of the input tensor.
- The dimension **nbDims** of the output array is negative or greater than the dimension of input tensor descriptor **inputTensorDesc**.

CUDNN_STATUS_SUCCESS

The routine exits successfully.

4.41. cudnnDestroyConvolutionDescriptor

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

This function destroys a previously created convolution descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.42. cudnnFindConvolutionForwardAlgorithm

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithm(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults)
```

This function attempts all cuDNN algorithms for **cudnnConvolutionForward()**, using memory allocated via **cudaMalloc()**, and outputs performance metrics to a user-allocated array of **cudnnConvolutionFwdAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest compute time.

> This function is host blocking.

> It is recommend to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

wDesc

Input. Handle to a previously initialized filter descriptor.

convDesc

Input. Previously initialized convolution descriptor.

yDesc

Input. Handle to the previously initialized output tensor descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- **`handle`** is not allocated properly.
- **`xDesc`**, **`wDesc`** or **`yDesc`** is not allocated properly.
- **`xDesc`**, **`wDesc`** or **`yDesc`** has fewer than 1 dimension.
- Either **`returnedCount`** or **`perfResults`** is nil.
- **`requestedCount`** is less than 1.

`CUDNN_STATUS_ALLOC_FAILED`

This function was unable to allocate memory to store sample input, filters and output.

`CUDNN_STATUS_INTERNAL_ERROR`

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.43. cudnnFindConvolutionForwardAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithmEx(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const cudnnFilterDescriptor_t      wDesc,
    const void                        *w,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    void                              *y,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults,
    void                              *workSpace,
    size_t                             workSpaceSizeInBytes)
```

This function attempts all available cuDNN algorithms for **cudnnConvolutionForward**, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of **cudnnConvolutionFwdAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest compute time.

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 arbitary values.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

workSpace

Input. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availability of algorithms. A nil pointer is considered a workSpace of 0 bytes.

workSpaceSizeInBytes

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- **`handle`** is not allocated properly.
- **`xDesc`**, **`wDesc`** or **`yDesc`** is not allocated properly.
- **`xDesc`**, **`wDesc`** or **`yDesc`** has fewer than 1 dimension.
- **`x`**, **`w`** or **`y`** is nil.
- Either **`returnedCount`** or **`perfResults`** is nil.
- **`requestedCount`** is less than 1.

`CUDNN_STATUS_INTERNAL_ERROR`

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.44. cudnnGetConvolutionForwardAlgorithm

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    cudnnConvolutionFwdPreference_t    preference,
    size_t                             memoryLimitInbytes,
```

```
    cudnnConvolutionFwdAlgo_t          *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionForward** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, please use **cudnnFindConvolutionForwardAlgorithm**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

wDesc

Input. Handle to a previously initialized convolution filter descriptor.

convDesc

Input. Previously initialized convolution descriptor.

yDesc

Input. Handle to the previously initialized output tensor descriptor.

preference

Input. Enumerant to express the preference criteria in terms of memory requirement and speed.

memoryLimitInBytes

Input. It is used when enumerant **preference** is set to **CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT** to specify the maximum amount of GPU memory the user is willing to use as a workspace

algo

Output. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- One of the parameters handle, xDesc, wDesc, convDesc, yDesc is NULL.
- Either yDesc or wDesc have different dimensions from xDesc.
- The data types of tensors xDesc, yDesc or wDesc are not all the same.
- The number of feature maps in xDesc and wDesc differs.
- The tensor xDesc has a dimension smaller than 3.

4.45. cudnnGetConvolutionForwardAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm_v7(
    cudnnHandle_t                      handle,
    const cudnnTensorDescriptor_t      xDesc,
    const cudnnFilterDescriptor_t      wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t      yDesc,
    const int                          requestedAlgoCount,
    int                               *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t     *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionForward** for the given layer specifications. This function will return all algorithms sorted by expected (based on internal heuristic) relative performance with fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, please use **cudnnFindConvolutionForwardAlgorithm**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

wDesc

Input. Handle to a previously initialized convolution filter descriptor.

convDesc

Input. Previously initialized convolution descriptor.

yDesc

Input. Handle to the previously initialized output tensor descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- One of the parameters handle, xDesc, wDesc, convDesc, yDesc, perfResults, returnedAlgoCount is NULL.
- Either yDesc or wDesc have different dimensions from xDesc.
- The data types of tensors xDesc, yDesc or wDesc are not all the same.
- The number of feature maps in xDesc and wDesc differs.
- The tensor xDesc has a dimension smaller than 3.
- requestedAlgoCount is less than or equal to 0.

4.46. cudnnGetConvolutionForwardWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionForwardWorkspaceSize(
    cudnnHandle_t   handle,
    const   cudnnTensorDescriptor_t         xDesc,
    const   cudnnFilterDescriptor_t         wDesc,
    const   cudnnConvolutionDescriptor_t    convDesc,
    const   cudnnTensorDescriptor_t         yDesc,
    cudnnConvolutionFwdAlgo_t               algo,
    size_t                                 *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **`cudnnConvolutionForward`** with the specified algorithm. The workspace allocated will then be passed to the routine **`cudnnConvolutionForward`**. The specified algorithm can be the result of the call to **`cudnnGetConvolutionForwardAlgorithm`** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized x tensor descriptor.

wDesc

Input. Handle to a previously initialized filter descriptor.

convDesc

Input. Previously initialized convolution descriptor.

yDesc

Input. Handle to the previously initialized y tensor descriptor.

algo

Input. Enumerant that specifies the chosen convolution algorithm

sizeInBytes

Output. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **`algo`**

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- One of the parameters handle, xDesc, wDesc, convDesc, yDesc is NULL.
- The tensor yDesc or wDesc are not of the same dimension as xDesc.
- The tensor xDesc, yDesc or wDesc are not of the same data type.
- The numbers of feature maps of the tensor xDesc and wDesc differ.
- The tensor xDesc has a dimension smaller than 3.

CUDNN_STATUS_NOT_SUPPORTED

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

4.47. cudnnConvolutionForward

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

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to a previously initialized 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.

algo

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

workSpace

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

workSpaceSizeInBytes

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

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

This function supports only eight specific combinations of data types for **`xDesc`**, **`wDesc`**, **`convDesc`** and **`yDesc`**. See the following for an exhaustive list of these configurations.

Data Type Configurations	`xDesc` and `wDesc`	`convDesc`	`yDesc`
TRUE_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_HALF	CUDNN_DATA_HALF
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT	CUDNN_DATA_HALF
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
INT8_CONFIG	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_INT8
INT8_EXT_CONFIG	CUDNN_DATA_INT8	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT
INT8x4_CONFIG	CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_INT8x4

Data Type Configurations	xDesc and wDesc	convDesc	yDesc
INT8x4_EXT_CONFIG	CUDNN_DATA_INT8x4	CUDNN_DATA_INT32	CUDNN_DATA_FLOAT

> TRUE_HALF_CONFIG is only supported on architectures with true fp16 support (compute capability 5.3 and 6.0).

> INT8_CONFIG, INT8_EXT_CONFIG, INT8x4_CONFIG and INT8x4_EXT_CONFIG are only supported on architectures with DP4A support (compute capability 6.1 and later).

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

For the datatype configurations TRUE_HALF_CONFIG, PSEUDO_HALF_CONFIG, FLOAT_CONFIG and DOUBLE_CONFIG, when the filter descriptor **wDesc** is in CUDNN_TENSOR_NCHW format the following is the exhaustive list of algo supported for 2-d convolutions.

- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_FWD_ALGO_GEMM**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
- **CUDNN_CONVOLUTION_FWD_ALGO_DIRECT**
  - This algorithm has no current implementation in cuDNN.
- **CUDNN_CONVOLUTION_FWD_ALGO_FFT**
  - **xDesc** Format Support: NCHW HW-packed
  - **yDesc** Format Support: NCHW HW-packed
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG
  - Dilation: 1 for all dimensions

  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **xDesc**'s feature map height + 2 * **convDesc**'s zero-padding height must equal 256 or less
    - **xDesc**'s feature map width + 2 * **convDesc**'s zero-padding width must equal 256 or less
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
    - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
- **CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING**
  - **xDesc** Format Support: NCHW HW-packed
  - **yDesc** Format Support: NCHW HW-packed
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG (DOUBLE_CONFIG is also supported when the task can be handled by 1D FFT, ie, one of the filter dimension, width or height is 1)
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - when neither of **wDesc**'s filter dimension is 1, the filter width and height must not be larger than 32
    - when either of **wDesc**'s filter dimension is 1, the largest filter dimension should not exceed 256
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
    - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
- **CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter height must be 3
    - **wDesc**'s filter width must be 3
- **CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All except DOUBLE_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:

    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter (height, width) must be (3,3) or (5,5)
    - If **wDesc**'s filter (height, width) is (5,5), data type config TRUE_HALF_CONFIG is not supported

For the datatype configurations TRUE_HALF_CONFIG, PSEUDO_HALF_CONFIG, FLOAT_CONFIG and DOUBLE_CONFIG, when the filter descriptor **wDesc** is in CUDNN_TENSOR_NHWC format the only algo supported is CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM with the following conditions :

- **xDesc** and **yDesc** is NHWC HWC-packed
- Data type configuration is PSEUDO_HALF_CONFIG or FLOAT_CONFIG
- The convolution is 2-dimensional
- Dilation is 1 for all dimensions
- **convDesc** Group Count Support: Equal to 1.

For the datatype configurations TRUE_HALF_CONFIG, PSEUDO_HALF_CONFIG, FLOAT_CONFIG and DOUBLE_CONFIG, when the filter descriptor **wDesc** is in CUDNN_TENSOR_NCHW format the following is the exhaustive list of algo supported for 3-d convolutions.

- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM**
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING**
  - **xDesc** Format Support: NCDHW DHW-packed
  - **yDesc** Format Support: NCDHW DHW-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **wDesc**'s filter height must equal 16 or less
    - **wDesc**'s filter width must equal 16 or less
    - **wDesc**'s filter depth must equal 16 or less
    - **convDesc**'s must have all filter strides equal to 1

  - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
  - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
  - **wDesc**'s filter depth must be greater than **convDesc**'s zero-padding width

For the datatype configurations INT8_CONFIG and INT8_EXT_CONFIG, the only algo supported is CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM with the following conditions :

- **xDesc** Format Support: CUDNN_TENSOR_NHWC
- **yDesc** Format Support: CUDNN_TENSOR_NHWC
- Input and output features maps must be multiple of 4
- **wDesc** Format Support: CUDNN_TENSOR_NHWC
- Dilation: 1 for all dimensions
- **convDesc** Group Count Support: Greater than 0.

For the datatype configurations INT8x4_CONFIG and INT8x4_EXT_CONFIG, the only algo supported is CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM with the following conditions :

- **xDesc** Format Support: CUDNN_TENSOR_NCHW_VECT_C
- **yDesc** Format Support: CUDNN_TENSOR_NCHW when dataype is CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW_VECT_C when datatype is CUDNN_DATA_INT8x4
- Input and output features maps must be multiple of 4
- **wDesc** Format Support: CUDNN_TENSOR_NCHW_VECT_C
- Dilation: 1 for all dimensions
- **convDesc** Group Count Support: Greater than 0.

> Tensors can be converted to/from CUDNN_TENSOR_NCHW_VECT_C with **cudnnTransformTensor()**.

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

Returns

CUDNN_STATUS_SUCCESS

The operation was launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- At least one of the following is NULL: handle, **xDesc**, **wDesc**, **convDesc**, **yDesc**, **xData**, **w**, **yData**, **alpha**, **beta**
- **xDesc** and **yDesc** have a non-matching number of dimensions
- **xDesc** and **wDesc** have a non-matching number of dimensions
- **xDesc** has fewer than three number of dimensions
- **xDesc**'s number of dimensions is not equal to **convDesc**'s array length + 2
- **xDesc** and **wDesc** have a non-matching number of input feature maps per image (or group in case of Grouped Convolutions)

- **yDesc** or **wDesc** indicate an output channel count that isn't a multiple of group count (if group count has been set in convDesc).
- **xDesc**, **wDesc** and **yDesc** have a non-matching data type
- For some spatial dimension, **wDesc** has a spatial size that is larger than the input spatial size (including zero-padding size)

CUDNN_STATUS_NOT_SUPPORTED

At least one of the following conditions are met:

- **xDesc** or **yDesc** have negative tensor striding
- **xDesc**, **wDesc** or **yDesc** has a number of dimensions that is not 4 or 5
- **yDescs's** spatial sizes do not match with the expected size as determined by **cudnnGetConvolutionNdForwardOutputDim**
- 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 occured during the texture binding of the filter data.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.48. cudnnConvolutionBiasActivationForward

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

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha1, alpha2

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as described by the above equation. Please refer to this section for additional details.

xDesc

Input. Handle to a previously initialized tensor descriptor.

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.

algo

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

workSpace

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

workSpaceSizeInBytes

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

zDesc

Input. Handle to a previously initialized tensor descriptor.

z

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

biasDesc

Input. Handle to a previously initialized tensor descriptor.

bias

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

activationDesc

Input. Handle to a previously initialized activation descriptor.

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

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

x	y and z	bias	alpha1/alpha2
CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE
CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
CUDNN_DATA_HALF	CUDNN_DATA_HALF	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT
CUDNN_DATA_INT8	CUDNN_DATA_INT8	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
CUDNN_DATA_INT8	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
CUDNN_DATA_INT8x4	CUDNN_DATA_INT8x4	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
CUDNN_DATA_INT8x4	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT

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

Returns

CUDNN_STATUS_SUCCESS

The operation was launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

CUDNN_STATUS_NOT_SUPPORTED

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

- The **mode** of **activationDesc** is not **CUDNN_ACTIVATION_RELU**
- The **reluNanOpt** of **activationDesc** is not **CUDNN_NOT_PROPAGATE_NAN**
- The second stride of **biasDesc** is not equal to one.
- The data type of **biasDesc** does not correspond to the data type of **yDesc** as listed in the above data types table.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.49. cudnnConvolutionBackwardBias

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

dyDesc

Input. Handle to the previously initialized input tensor descriptor.

dy

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

dbDesc

Input. Handle to the previously initialized output tensor descriptor.

db

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The operation was launched successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the parameters **`n,height,width`** of the output tensor is not 1.

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors are different.

4.50. cudnnFindConvolutionBackwardFilterAlgorithm

```
cudnnStatus_t cudnnFindConvolutionBackwardFilterAlgorithm(
cudnnHandle_t                                handle,
const cudnnTensorDescriptor_t                xDesc,
const cudnnTensorDescriptor_t                dyDesc,
const cudnnConvolutionDescriptor_t           convDesc,
const cudnnFilterDescriptor_t                dwDesc,
const int                                    requestedAlgoCount,
int                                         *returnedAlgoCount,
cudnnConvolutionBwdFilterAlgoPerf_t         *perfResults)
```

This function attempts all cuDNN algorithms for **cudnnConvolutionBackwardFilter()**, using GPU memory allocated via **cudaMalloc()**, and outputs performance metrics to a user-allocated array of **cudnnConvolutionBwdFilterAlgoPerf_t**. These metrics are written in sorted fashion where the first element has the lowest compute time.

This function is host blocking.

It is recommend to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dwDesc

Input. Handle to a previously initialized filter descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **xDesc**, **dyDesc** or **dwDesc** is not allocated properly.
- **xDesc**, **dyDesc** or **dwDesc** has fewer than 1 dimension.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

CUDNN_STATUS_ALLOC_FAILED

This function was unable to allocate memory to store sample input, filters and output.

CUDNN_STATUS_INTERNAL_ERROR

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.51. cudnnFindConvolutionBackwardFilterAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionBackwardFilterAlgorithmEx(
    cudnnHandle_t                          handle,
    const cudnnTensorDescriptor_t          xDesc,
    const void                            *x,
    const cudnnTensorDescriptor_t          dyDesc,
    const void                            *dy,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnFilterDescriptor_t          dwDesc,
    void                                  *dw,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionBwdFilterAlgoPerf_t   *perfResults,
    void                                  *workSpace,
    size_t                                 workSpaceSizeInBytes)
```

This function attempts all cuDNN algorithms for **cudnnConvolutionBackwardFilter**, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of **`cudnnConvolutionBwdFilterAlgoPerf_t`**. These metrics are written in sorted fashion where the first element has the lowest compute time.

> 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 arbitary values.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

workSpace

Input. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availabilty of algorithms. A nil pointer is considered a workSpace of 0 bytes.

workSpaceSizeInBytes

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- **`handle`** is not allocated properly.
- **`xDesc`**, **`dyDesc`** or **`dwDesc`** is not allocated properly.
- **`xDesc`**, **`dyDesc`** or **`dwDesc`** has fewer than 1 dimension.
- **`x`**, **`dy`** or **`dw`** is nil.
- Either **`returnedCount`** or **`perfResults`** is nil.
- **`requestedCount`** is less than 1.

`CUDNN_STATUS_INTERNAL_ERROR`

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.52. cudnnGetConvolutionBackwardFilterAlgorithm

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithm(
    cudnnHandle_t                          handle,
    const cudnnTensorDescriptor_t          xDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnFilterDescriptor_t          dwDesc,
    cudnnConvolutionBwdFilterPreference_t  preference,
    size_t                                 memoryLimitInbytes,
    cudnnConvolutionBwdFilterAlgo_t       *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **`cudnnConvolutionBackwardFilter`** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, please use **`cudnnFindConvolutionBackwardFilterAlgorithm`**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dwDesc

Input. Handle to a previously initialized filter descriptor.

preference

Input. Enumerant to express the preference criteria in terms of memory requirement and speed.

memoryLimitInbytes

Input. It is to specify the maximum amount of GPU memory the user is willing to use as a workspace. This is currently a placeholder and is not used.

algo

Output. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **dataType** of the two tensor descriptors or the filter are different.

4.53. cudnnGetConvolutionBackwardFilterAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterAlgorithm_v7(
    cudnnHandle_t                          handle,
    const cudnnTensorDescriptor_t          xDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnFilterDescriptor_t          dwDesc,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t         *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardFilter** for the given layer specifications. This function will return all algorithms sorted by expected (based on internal heuristic) relative performance with fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, please use **cudnnFindConvolutionBackwardFilterAlgorithm**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dwDesc

Input. Handle to a previously initialized filter descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the parameters handle, xDesc, dyDesc, convDesc, dwDesc, perfResults, returnedAlgoCount is NULL.
- The numbers of feature maps of the input tensor and output tensor differ.
- The **`dataType`** of the two tensor descriptors or the filter are different.
- requestedAlgoCount is less than or equal to 0.

4.54. cudnnGetConvolutionBackwardFilterWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionBackwardFilterWorkspaceSize(
    cudnnHandle_t                       handle,
    const cudnnTensorDescriptor_t       xDesc,
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnFilterDescriptor_t       dwDesc,
    cudnnConvolutionFwdAlgo_t           algo,
    size_t                             *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **`cudnnConvolutionBackwardFilter`** with the specified algorithm. The workspace allocated will then be passed to the routine **`cudnnConvolutionBackwardFilter`**. The specified algorithm can be the result of the call to **`cudnnGetConvolutionBackwardFilterAlgorithm`** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

Parameters

handle

Input. Handle to a previously created cuDNN context.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dwDesc

Input. Handle to a previously initialized filter descriptor.

algo

Input. Enumerant that specifies the chosen convolution algorithm.

sizeInBytes

Output. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **`algo`**.

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **`dataType`** of the two tensor descriptors or the filter are different.

`CUDNN_STATUS_NOT_SUPPORTED`

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

4.55. cudnnConvolutionBackwardFilter

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

```
    const void                          *beta,
    const cudnnFilterDescriptor_t        dwDesc,
    void                                *dw)
```

This function computes the convolution gradient with respect to filter coefficients using the specified **`algo`**, returning results in **`gradDesc`**.Scaling factors **`alpha`** and **`beta`** can be used to scale the input tensor and the output tensor respectively.

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to a previously initialized tensor descriptor.

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.

algo

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

workSpace

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

workSpaceSizeInBytes

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

dwDesc

Input. Handle to a previously initialized filter gradient descriptor.

dw

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

This function supports only three specific combinations of data types for **`xDesc`**, **`dyDesc`**, **`convDesc`** and **`dwDesc`**. See the following for an exhaustive list of these configurations.

Data Type Configurations	`xDesc`'s, `dyDesc`'s and `dwDesc`'s Data Type	`convDesc`'s Data Type
TRUE_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_HALF
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE

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

`dwDesc` may only have format CUDNN_TENSOR_NHWC when all of the following are true:

- **`algo`** is CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0 or CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1
- **`xDesc`** and **`dyDesc`** is NHWC HWC-packed
- Data type configuration is PSEUDO_HALF_CONFIG or FLOAT_CONFIG
- The convolution is 2-dimensional

The following is an exhaustive list of algo support for 2-d convolutions.

- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0**
  - Deterministic: No
  - **`xDesc`** Format Support: All except NCHW_VECT_C
  - **`dyDesc`** Format Support: NCHW CHW-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **`convDesc`** Group Count Support: Greater than 0.
  - Not supported if output is of type CUDNN_DATA_HALF and the number of elements in dw is odd.
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1**
  - Deterministic: Yes
  - **`xDesc`** Format Support: All
  - **`dyDesc`** Format Support: NCHW CHW-packed
  - Data Type Config Support: All
  - Dilation: 1 for all dimensions
  - **`convDesc`** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT**
  - Deterministic: Yes
  - **`xDesc`** Format Support: NCHW CHW-packed
  - **`dyDesc`** Format Support: NCHW CHW-packed

  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG
  - **convDesc** Group Count Support: Equal to 1.
  - Dilation: 1 for all dimensions
  - Notes:
    - **xDesc**'s feature map height + 2 * **convDesc**'s zero-padding height must equal 256 or less
    - **xDesc**'s feature map width + 2 * **convDesc**'s zero-padding width must equal 256 or less
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **dwDesc**'s filter height must be greater than **convDesc**'s zero-padding height
    - **dwDesc**'s filter width must be greater than **convDesc**'s zero-padding width
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3**
  - Deterministic: No
  - **xDesc** Format Support: All except NCHW_VECT_C
  - **dyDesc** Format Support: NCHW CHW-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - **convDesc** Group Count Support: Greater than 0.
  - Dilation: 1 for all dimensions
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED**
  - Deterministic: Yes
  - **xDesc** Format Support: All except CUDNN_TENSOR_NCHW_VECT_C
  - **yDesc** Format Support: NCHW CHW-packed
  - Data Type Config Support: All except DOUBLE_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter (height, width) must be (3,3) or (5,5)
    - If **wDesc**'s filter (height, width) is (5,5), data type config TRUE_HALF_CONFIG is not supported
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING**
  - Deterministic: Yes
  - **xDesc** Format Support: NCHW CHW-packed
  - **dyDesc** Format Support: NCHW CHW-packed
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG, DOUBLE_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **xDesc**'s width or height must be equal to 1

  - **dyDesc**'s width or height must be equal to 1 (the same dimension as in **xDesc**). The other dimension must be less than or equal to 256, ie, the largest 1D tile size currently supported
  - **convDesc**'s vertical and horizontal filter stride must equal 1
  - **dwDesc**'s filter height must be greater than **convDesc**'s zero-padding height
  - **dwDesc**'s filter width must be greater than **convDesc**'s zero-padding width

The following is an exhaustive list of algo support for 3-d convolutions.

- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0**
  - Deterministic: No
  - **xDesc** Format Support: All except NCHW_VECT_C
  - **dyDesc** Format Support: NCDHW CDHW-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3**
  - Deterministic: No
  - **xDesc** Format Support: NCDHW-fully-packed
  - **dyDesc** Format Support: NCDHW-fully-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.

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

Returns

CUDNN_STATUS_SUCCESS

The operation was launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

CUDNN_STATUS_NOT_SUPPORTED

At least one of the following conditions are met:

- **xDesc** or **dyDesc** have negative tensor striding

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

`CUDNN_STATUS_MAPPING_ERROR`

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

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.56. cudnnFindConvolutionBackwardDataAlgorithm

```
cudnnStatus_t cudnnFindConvolutionBackwardDataAlgorithm(
    cudnnHandle_t                       handle,
    const cudnnFilterDescriptor_t       wDesc,
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       dxDesc,
    const int                           requestedAlgoCount,
    int                                *returnedAlgoCount,
    cudnnConvolutionBwdFilterAlgoPerf_t *perfResults)
```

This function attempts all cuDNN algorithms for **`cudnnConvolutionBackwardData()`**, using memory allocated via **`cudaMalloc()`** and outputs performance metrics to a user-allocated array of **`cudnnConvolutionBwdDataAlgoPerf_t`**. These metrics are written in sorted fashion where the first element has the lowest compute time.

> This function is host blocking.

> It is recommend to run this function prior to allocating layer data; doing otherwise may needlessly inhibit some algorithm options due to resource usage.

Parameters

handle

Input. Handle to a previously created cuDNN context.

wDesc

Input. Handle to a previously initialized filter descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dxDesc

Input. Handle to the previously initialized output tensor descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **wDesc**, **dyDesc** or **dxDesc** is not allocated properly.
- **wDesc**, **dyDesc** or **dxDesc** has fewer than 1 dimension.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

CUDNN_STATUS_ALLOC_FAILED

This function was unable to allocate memory to store sample input, filters and output.

CUDNN_STATUS_INTERNAL_ERROR

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.57. cudnnFindConvolutionBackwardDataAlgorithmEx

```
cudnnStatus_t cudnnFindConvolutionBackwardDataAlgorithmEx(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const void                            *w,
    const cudnnTensorDescriptor_t          dyDesc,
    const void                            *dy,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    void                                  *dx,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionBwdFilterAlgoPerf_t   *perfResults,
    void                                  *workSpace,
    size_t                                 workSpaceSizeInBytes)
```

This function attempts all cuDNN algorithms for **cudnnConvolutionBackwardData**, using user-allocated GPU memory, and outputs performance metrics to a user-allocated array of **`cudnnConvolutionBwdDataAlgoPerf_t`**. These metrics are written in sorted fashion where the first element has the lowest compute time.

> 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 arbitary values.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

workSpace

Input. Data pointer to GPU memory that is a necessary workspace for some algorithms. The size of this workspace will determine the availabilty of algorithms. A nil pointer is considered a workSpace of 0 bytes.

workSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is not allocated properly.
- **wDesc**, **dyDesc** or **dxDesc** is not allocated properly.
- **wDesc**, **dyDesc** or **dxDesc** has fewer than 1 dimension.
- **w**, **dy** or **dx** is nil.
- Either **returnedCount** or **perfResults** is nil.
- **requestedCount** is less than 1.

CUDNN_STATUS_INTERNAL_ERROR

At least one of the following conditions are met:

- The function was unable to allocate neccesary timing objects.
- The function was unable to deallocate neccesary timing objects.
- The function was unable to deallocate sample input, filters and output.

4.58. cudnnGetConvolutionBackwardDataAlgorithm

```
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithm(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    cudnnConvolutionBwdDataPreference_t    preference,
    size_t                                 memoryLimitInbytes,
    cudnnConvolutionBwdDataAlgo_t         *algo)
```

This function serves as a heuristic for obtaining the best suited algorithm for **cudnnConvolutionBackwardData** for the given layer specifications. Based on the input preference, this function will either return the fastest algorithm or the fastest algorithm within a given memory limit. For an exhaustive search for the fastest algorithm, please use **cudnnFindConvolutionBackwardDataAlgorithm**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

wDesc

Input. Handle to a previously initialized filter descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dxDesc

Input. Handle to the previously initialized output tensor descriptor.

preference

Input. Enumerant to express the preference criteria in terms of memory requirement and speed.

memoryLimitInbytes

Input. It is to specify the maximum amount of GPU memory the user is willing to use as a workspace. This is currently a placeholder and is not used.

algo

Output. Enumerant that specifies which convolution algorithm should be used to compute the results according to the specified preference

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **`dataType`** of the two tensor descriptors or the filter are different.

4.59. cudnnGetConvolutionBackwardDataAlgorithm_v7

```
cudnnStatus_t cudnnGetConvolutionBackwardDataAlgorithm_v7(
    cudnnHandle_t                          handle,
    const cudnnFilterDescriptor_t          wDesc,
    const cudnnTensorDescriptor_t          dyDesc,
    const cudnnConvolutionDescriptor_t     convDesc,
    const cudnnTensorDescriptor_t          dxDesc,
    const int                              requestedAlgoCount,
    int                                   *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t         *perfResults)
```

This function serves as a heuristic for obtaining the best suited algorithm for **`cudnnConvolutionBackwardData`** for the given layer specifications. This function will return all algorithms sorted by expected (based on internal heuristic) relative performance with fastest being index 0 of perfResults. For an exhaustive search for the fastest algorithm, please use **`cudnnFindConvolutionBackwardDataAlgorithm`**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

wDesc

Input. Handle to a previously initialized filter descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dxDesc

Input. Handle to the previously initialized output tensor descriptor.

requestedAlgoCount

Input. The maximum number of elements to be stored in perfResults.

returnedAlgoCount

Output. The number of output elements stored in perfResults.

perfResults

Output. A user-allocated array to store performance metrics sorted ascending by compute time.

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the parameters handle, wDesc, dyDesc, convDesc, dxDesc, perfResults, returnedAlgoCount is NULL.
- The numbers of feature maps of the input tensor and output tensor differ.
- The **`dataType`** of the two tensor descriptors or the filter are different.
- requestedAlgoCount is less than or equal to 0.

4.60. cudnnGetConvolutionBackwardDataWorkspaceSize

```
cudnnStatus_t cudnnGetConvolutionBackwardDataWorkspaceSize(
    cudnnHandle_t                       handle,
    const cudnnFilterDescriptor_t       wDesc,
    const cudnnTensorDescriptor_t       dyDesc,
    const cudnnConvolutionDescriptor_t  convDesc,
    const cudnnTensorDescriptor_t       dxDesc,
    cudnnConvolutionFwdAlgo_t           algo,
    size_t                             *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **`cudnnConvolutionBackwardData`** with the specified algorithm. The workspace allocated will then be passed to the routine **`cudnnConvolutionBackwardData`**. The specified algorithm can be the result of the call to **`cudnnGetConvolutionBackwardDataAlgorithm`** or can be chosen arbitrarily by the user. Note that not every algorithm is available for every configuration of the input tensor and/or every configuration of the convolution descriptor.

Parameters

handle

Input. Handle to a previously created cuDNN context.

wDesc

Input. Handle to a previously initialized filter descriptor.

dyDesc

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

convDesc

Input. Previously initialized convolution descriptor.

dxDesc

Input. Handle to the previously initialized output tensor descriptor.

algo

Input. Enumerant that specifies the chosen convolution algorithm

sizeInBytes

Output. Amount of GPU memory needed as workspace to be able to execute a forward convolution with the specified **`algo`**

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The numbers of feature maps of the input tensor and output tensor differ.
- The **`dataType`** of the two tensor descriptors or the filter are different.

`CUDNN_STATUS_NOT_SUPPORTED`

The combination of the tensor descriptors, filter descriptor and convolution descriptor is not supported for the specified algorithm.

4.61. cudnnConvolutionBackwardData

```
cudnnStatus_t cudnnConvolutionBackwardData(
    cudnnHandle_t                        handle,
    const void                          *alpha,
    const cudnnFilterDescriptor_t        wDesc,
    const void                          *w,
    const cudnnTensorDescriptor_t        dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionBwdDataAlgo_t        algo,
    void                                *workSpace,
    size_t                               workSpaceSizeInBytes,
```

```
    const void                          *beta,
    const cudnnTensorDescriptor_t        dxDesc,
    void                                *dx)
```

This function computes the convolution gradient with respect to the output tensor using the specified **algo**, returning results in **gradDesc**. Scaling factors **alpha** and **beta** can be used to scale the input tensor and the output tensor respectively.

Parameters

handle

Input. Handle to a previously created cuDNN context.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

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 input differential tensor descriptor **dyDesc**.

convDesc

Input. Previously initialized convolution descriptor.

algo

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

workSpace

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

workSpaceSizeInBytes

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

dxDesc

Input. Handle to the previously initialized output tensor descriptor.

dx

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

This function supports only three specific combinations of data types for **`wDesc`**, **`dyDesc`**, **`convDesc`** and **`dxDesc`**. See the following for an exhaustive list of these configurations.

Data Type Configurations	`wDesc`'s, `dyDesc`'s and `dxDesc`'s Data Type	`convDesc`'s Data Type
TRUE_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_HALF
PSEUDO_HALF_CONFIG	CUDNN_DATA_HALF	CUDNN_DATA_FLOAT
FLOAT_CONFIG	CUDNN_DATA_FLOAT	CUDNN_DATA_FLOAT
DOUBLE_CONFIG	CUDNN_DATA_DOUBLE	CUDNN_DATA_DOUBLE

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

`wDesc` may only have format CUDNN_TENSOR_NHWC when all of the following are true:

- **`algo`** is CUDNN_CONVOLUTION_BWD_DATA_ALGO_1
- **`dyDesc`** and **`dxDesc`** is NHWC HWC-packed
- Data type configuration is PSEUDO_HALF_CONFIG or FLOAT_CONFIG
- The convolution is 2-dimensional

The following is an exhaustive list of algo support for 2-d convolutions.

- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_0**
  - Deterministic: No
  - **`dyDesc`** Format Support: NCHW CHW-packed
  - **`dxDesc`** Format Support: All except NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **`convDesc`** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_1**
  - Deterministic: Yes
  - **`dyDesc`** Format Support: NCHW CHW-packed
  - **`dxDesc`** Format Support: All except NCHW_VECT_C
  - Data Type Config Support: All
  - Dilation: 1 for all dimensions
  - **`convDesc`** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT**
  - Deterministic: Yes
  - **`dyDesc`** Format Support: NCHW CHW-packed
  - **`dxDesc`** Format Support: NCHW HW-packed
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG
  - Dilation: 1 for all dimensions
  - **`convDesc`** Group Count Support: Equal to 1.

- Notes:
  - **dxDesc**'s feature map height + 2 * **convDesc**'s zero-padding height must equal 256 or less
  - **dxDesc**'s feature map width + 2 * **convDesc**'s zero-padding width must equal 256 or less
  - **convDesc**'s vertical and horizontal filter stride must equal 1
  - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
  - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING**
  - Deterministic: Yes
  - **dyDesc** Format Support: NCHW CHW-packed
  - **dxDesc** Format Support: NCHW HW-packed
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG (DOUBLE_CONFIG is also supported when the task can be handled by 1D FFT, ie, one of the filter dimension, width or height is 1)
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - when neither of **wDesc**'s filter dimension is 1, the filter width and height must not be larger than 32
    - when either of **wDesc**'s filter dimension is 1, the largest filter dimension should not exceed 256
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
    - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD**
  - Deterministic: Yes
  - **xDesc** Format Support: NCHW CHW-packed
  - **yDesc** Format Support: All except NCHW_VECT_C
  - Data Type Config Support: PSEUDO_HALF_CONFIG, FLOAT_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **convDesc**'s vertical and horizontal filter stride must equal 1
    - **wDesc**'s filter height must be 3
    - **wDesc**'s filter width must be 3
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED**
  - Deterministic: Yes
  - **xDesc** Format Support: NCHW CHW-packed
  - **yDesc** Format Support: All except NCHW_VECT_C
  - Data Type Config Support: All except DOUBLE_CONFIG
  - Dilation: 1 for all dimensions

- **convDesc** Group Count Support: Equal to 1.
- Notes:
  - **convDesc**'s vertical and horizontal filter stride must equal 1
  - **wDesc**'s filter (height, width) must be (3,3) or (5,5)
  - If **wDesc**'s filter (height, width) is (5,5), data type config TRUE_HALF_CONFIG is not supported

The following is an exhaustive list of algo support for 3-d convolutions.

- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_0**
  - Deterministic: No
  - **dyDesc** Format Support: NCDHW CDHW-packed
  - **dxDesc** Format Support: All except NCHW_VECT_C
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: greater than 0 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_1**
  - Deterministic: Yes
  - **dyDesc** Format Support: NCDHW-fully-packed
  - **dxDesc** Format Support: NCDHW-fully-packed
  - Data Type Config Support: All
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Greater than 0.
- **CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING**
  - Deterministic: Yes
  - **dyDesc** Format Support: NCDHW CDHW-packed
  - **dxDesc** Format Support: NCDHW DHW-packed
  - Data Type Config Support: All except TRUE_HALF_CONFIG
  - Dilation: 1 for all dimensions
  - **convDesc** Group Count Support: Equal to 1.
  - Notes:
    - **wDesc**'s filter height must equal 16 or less
    - **wDesc**'s filter width must equal 16 or less
    - **wDesc**'s filter depth must equal 16 or less
    - **convDesc**'s must have all filter strides equal to 1
    - **wDesc**'s filter height must be greater than **convDesc**'s zero-padding height
    - **wDesc**'s filter width must be greater than **convDesc**'s zero-padding width
    - **wDesc**'s filter depth must be greater than **convDesc**'s zero-padding width

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

Returns

CUDNN_STATUS_SUCCESS

The operation was launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- At least one of the following is NULL: **handle**, **dyDesc**, **wDesc**, **convDesc**, **dxDesc**, **dy**, **w**, **dx**, **alpha**, **beta**
- **wDesc** and **dyDesc** have a non-matching number of dimensions
- **wDesc** and **dxDesc** have a non-matching number of dimensions
- **wDesc** has fewer than three number of dimensions
- **wDesc**, **dxDesc** and **dyDesc** have a non-matching data type.
- **wDesc** and **dxDesc** have a non-matching number of input feature maps per image (or group in case of Grouped Convolutions).
- **dyDescs's** spatial sizes do not match with the expected size as determined by **cudnnGetConvolutionNdForwardOutputDim**

CUDNN_STATUS_NOT_SUPPORTED

At least one of the following conditions are met:

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

CUDNN_STATUS_MAPPING_ERROR

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

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.62. cudnnSoftmaxForward

```
cudnnStatus_t cudnnSoftmaxForward(
    cudnnHandle_t                    handle,
    cudnnSoftmaxAlgorithm_t          algorithm,
    cudnnSoftmaxMode_t               mode,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

This routine computes the softmax function.

> All tensor formats are supported for all modes and algorithms with 4 and 5D tensors. Performance is expected to be highest with **NCHW fully-packed** tensors. For more than 5 dimensions tensors must be packed in their spatial dimensions

Parameters

handle

Input. Handle to a previously created cuDNN context.

algorithm

Input. Enumerant to specify the softmax algorithm.

mode

Input. Enumerant to specify the softmax mode.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

x

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

yDesc

Input. Handle to the previously initialized output tensor descriptor.

y

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

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

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

4.63. cudnnSoftmaxBackward

```
cudnnStatus_t cudnnSoftmaxBackward(
    cudnnHandle_t                    handle,
    cudnnSoftmaxAlgorithm_t          algorithm,
```

```
    cudnnSoftmaxMode_t                 mode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      yDesc,
    const void                        *yData,
    const cudnnTensorDescriptor_t      dyDesc,
    const void                        *dy,
    const void                        *beta,
    const cudnnTensorDescriptor_t      dxDesc,
    void                              *dx)
```

This routine computes the gradient of the softmax function.

> In-place operation is allowed for this routine; i.e., **dy** and **dx** pointers may be equal. However, this requires **dyDesc** and **dxDesc** descriptors to be identical (particularly, the strides of the input and output must match for in-place operation to be allowed).

> All tensor formats are supported for all modes and algorithms with 4 and 5D tensors. Performance is expected to be highest with **NCHW fully-packed** tensors. For more than 5 dimensions tensors must be packed in their spatial dimensions

Parameters

handle

Input. Handle to a previously created cuDNN context.

algorithm

Input. Enumerant to specify the softmax algorithm.

mode

Input. Enumerant to specify the softmax mode.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

yDesc

Input. Handle to the previously initialized input tensor descriptor.

y

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

dyDesc

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

dy

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

dxDesc

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

dx

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The dimensions **n,c,h,w** of the **yDesc**, **dyDesc** and **dxDesc** tensors differ.
- The strides **nStride, cStride, hStride, wStride** of the **yDesc** and **dyDesc** tensors differ.
- The **datatype** of the three tensors differs.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.64. cudnnCreatePoolingDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.65. cudnnSetPooling2dDescriptor

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

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

Parameters

poolingDesc

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

mode

Input. Enumerant to specify the pooling mode.

maxpoolingNanOpt

Input. Enumerant to specify the Nan propagation mode.

windowHeight

Input. Height of the pooling window.

windowWidth

Input. Width of the pooling window.

verticalPadding

Input. Size of vertical padding.

horizontalPadding

Input. Size of horizontal padding

verticalStride

Input. Pooling vertical stride.

horizontalStride

Input. Pooling horizontal stride.

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

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

4.66. cudnnGetPooling2dDescriptor

```
cudnnStatus_t cudnnGetPooling2dDescriptor(
    const cudnnPoolingDescriptor_t      poolingDesc,
    cudnnPoolingMode_t                 *mode,
    cudnnNanPropagation_t              *maxpoolingNanOpt,
    int                                *windowHeight,
    int                                *windowWidth,
    int                                *verticalPadding,
    int                                *horizontalPadding,
    int                                *verticalStride,
    int                                *horizontalStride)
```

This function queries a previously created 2D pooling descriptor object.

Parameters

poolingDesc

Input. Handle to a previously created pooling descriptor.

mode

Output. Enumerant to specify the pooling mode.

maxpoolingNanOpt

Output. Enumerant to specify the Nan propagation mode.

windowHeight

Output. Height of the pooling window.

windowWidth

Output. Width of the pooling window.

verticalPadding

Output. Size of vertical padding.

horizontalPadding

Output. Size of horizontal padding.

verticalStride

Output. Pooling vertical stride.

horizontalStride

Output. Pooling horizontal stride.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

4.67. cudnnSetPoolingNdDescriptor

```
cudnnStatus_t cudnnSetPoolingNdDescriptor(
    cudnnPoolingDescriptor_t    poolingDesc,
    cudnnPoolingMode_t          mode,
    cudnnNanPropagation_t       maxpoolingNanOpt,
    int                         nbDims,
    int                         windowDimA[],
    int                         paddingA[],
    int                         strideA[])
```

This function initializes a previously created generic pooling descriptor object.

Parameters

poolingDesc

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

mode

Input. Enumerant to specify the pooling mode.

maxpoolingNanOpt

Input. Enumerant to specify the Nan propagation mode.

nbDims

Input. Dimension of the pooling operation.

windowDimA

Output. Array of dimension **nbDims** containing the window size for each dimension.

paddingA

Output. Array of dimension **nbDims** containing the padding size for each dimension.

strideA

Output. Array of dimension **nbDims** containing the striding size for each dimension.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the elements of the arrays **windowDimA, paddingA or strideA** is negative or **mode** or **maxpoolingNanOpt**has an invalid enumerant value.

4.68. cudnnGetPoolingNdDescriptor

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

This function queries a previously initialized generic pooling descriptor object.

Parameters

poolingDesc

Input. Handle to a previously created pooling descriptor.

nbDimsRequested

Input. Dimension of the expected pooling descriptor. It is also the minimum size of the arrays **windowDimA**, **paddingA** and **strideA** in order to be able to hold the results.

mode

Output. Enumerant to specify the pooling mode.

maxpoolingNanOpt

Input. Enumerant to specify the Nan propagation mode.

nbDims

Output. Actual dimension of the pooling descriptor.

windowDimA

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

paddingA

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

strideA

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was queried successfully.

CUDNN_STATUS_NOT_SUPPORTED

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

4.69. cudnnDestroyPoolingDescriptor

```
cudnnStatus_t cudnnDestroyPoolingDescriptor(
    cudnnPoolingDescriptor_t poolingDesc)
```

This function destroys a previously created pooling descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.70. cudnnGetPooling2dForwardOutputDim

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

This function provides the output dimensions of a tensor after 2d pooling has been applied

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

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

Parameters

poolingDesc

Input. Handle to a previously inititalized pooling descriptor.

inputDesc

Input. Handle to the previously initialized input tensor descriptor.

N

Output. Number of images in the output.

C

Output. Number of channels in the output.

H

Output. Height of images in the output.

W

Output. Width of images in the output.

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

4.71. cudnnGetPoolingNdForwardOutputDim

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

This function provides the output dimensions of a tensor after Nd pooling has been applied

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

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

Parameters

poolingDesc

Input. Handle to a previously inititalized pooling descriptor.

inputDesc

Input. Handle to the previously initialized input tensor descriptor.

nbDims

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

outDimA

Output. Array of nbDims output dimensions.

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

4.72. cudnnPoolingForward

```
cudnnStatus_t cudnnPoolingForward(
    cudnnHandle_t                    handle,
    const cudnnPoolingDescriptor_t   poolingDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t    yDesc,
    void                            *y)
```

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

> All tensor formats are supported, best performance is expected when using **`HW-packed`** tensors. Only 2 and 3 spatial dimensions are allowed.

> The dimensions of the ouput tensor **`yDesc`** can be smaller or bigger than the dimensions advised by the routine **`cudnnGetPooling2dForwardOutputDim`** or **`cudnnGetPoolingNdForwardOutputDim`**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

poolingDesc

Input. Handle to a previously initialized pooling descriptor.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

x

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

yDesc

Input. Handle to the previously initialized output tensor descriptor.

y

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

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

`CUDNN_STATUS_NOT_SUPPORTED`

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

- The **wStride** of input tensor or output tensor is not 1.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.73. cudnnPoolingBackward

```
cudnnStatus_t cudnnPoolingBackward(
    cudnnHandle_t                       handle,
    const cudnnPoolingDescriptor_t      poolingDesc,
    const void                         *alpha,
    const cudnnTensorDescriptor_t       yDesc,
    const void                         *y,
    const cudnnTensorDescriptor_t       dyDesc,
    const void                         *dy,
    const cudnnTensorDescriptor_t       xDesc,
    const void                         *xData,
    const void                         *beta,
    const cudnnTensorDescriptor_t       dxDesc,
    void                               *dx)
```

This function computes the gradient of a pooling operation.

As of cuDNN version 6.0, a deterministic algorithm is implemented for max backwards pooling. This algorithm can be chosen via the pooling mode enum of **poolingDesc**. The deterministic algorithm has been measured to be up to 50% slower than the legacy max backwards pooling algorithm, or up to 20% faster, depending upon the use case.

> All tensor formats are supported, best performance is expected when using **HW-packed** tensors. Only 2 and 3 spatial dimensions are allowed

Parameters

handle

Input. Handle to a previously created cuDNN context.

poolingDesc

Input. Handle to the previously initialized pooling descriptor.

alpha, beta

Input. Pointers to scaling factors (in host memory) used to blend the computation result with prior value in the output layer as follows: dstValue = alpha[0]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

yDesc

Input. Handle to the previously initialized input tensor descriptor.

y

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

dyDesc

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

dy

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

xDesc

Input. Handle to the previously initialized output tensor descriptor.

x

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

dxDesc

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

dx

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The dimensions **n,c,h,w** of the **yDesc** and **dyDesc** tensors differ.
- The strides **nStride, cStride, hStride, wStride** of the **yDesc** and **dyDesc** tensors differ.
- The dimensions **n,c,h,w** of the **dxDesc** and **dxDesc** tensors differ.
- The strides **nStride, cStride, hStride, wStride** of the **xDesc** and **dxDesc** tensors differ.
- The **datatype** of the four tensors differ.

CUDNN_STATUS_NOT_SUPPORTED

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

- The **wStride** of input tensor or output tensor is not 1.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.74. cudnnActivationForward

```
cudnnStatus_t cudnnActivationForward(
    cudnnHandle_t handle,
    cudnnActivationDescriptor_t     activationDesc,
    const void                     *alpha,
    const cudnnTensorDescriptor_t   srcDesc,
    const void                     *srcData,
```

```
    const void                         *beta,
    const cudnnTensorDescriptor_t       destDesc,
    void                               *destData)
```

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

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

activationDesc

Input. Activation 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]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

x

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

yDesc

Input. Handle to the previously initialized output tensor descriptor.

y

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.75. cudnnActivationBackward

```
cudnnStatus_t cudnnActivationBackward(
    cudnnHandle_t                    handle,
    cudnnActivationDescriptor_t      activationDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t    srcDesc,
    const void                      *srcData,
    const cudnnTensorDescriptor_t    srcDiffDesc,
    const void                      *srcDiffData,
    const cudnnTensorDescriptor_t    destDesc,
    const void                      *destData,
    const void                      *beta,
    const cudnnTensorDescriptor_t    destDiffDesc,
    void                            *destDiffData)
```

This routine computes the gradient of a neuron activation function.

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

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

activationDesc,

Input. Activation 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]*result + beta[0]*priorDstValue. Please refer to this section for additional details.

yDesc

Input. Handle to the previously initialized input tensor descriptor.

y

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

dyDesc

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

dy

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

xDesc

Input. Handle to the previously initialized output tensor descriptor.

x

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

dxDesc

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

dx

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

CUDNN_STATUS_NOT_SUPPORTED

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

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

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

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.76. cudnnCreateActivationDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.77. cudnnSetActivationDescriptor

```
cudnnStatus_t cudnnSetActivationDescriptor(
    cudnnActivationDescriptor_t         activationDesc,
    cudnnActivationMode_t               mode,
    cudnnNanPropagation_t               reluNanOpt,
    double                              coef)
```

This function initializes a previously created generic activation descriptor object.

Parameters

activationDesc

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

mode

Input. Enumerant to specify the activation mode.

reluNanOpt

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

coef

Input. floating point number to specify the clipping threashold when the activation mode is set to **CUDNN_ACTIVATION_CLIPPED_RELU** or to specify the alpha coefficient when the activation mode is set to **CUDNN_ACTIVATION_ELU**.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

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

4.78. cudnnGetActivationDescriptor

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

This function queries a previously initialized generic activation descriptor object.

Parameters

activationDesc

Input. Handle to a previously created activation descriptor.

mode

Output. Enumerant to specify the activation mode.

reluNanOpt

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

coef

Output. Floating point number to specify the clipping threashod when the activation mode is set to **CUDNN_ACTIVATION_CLIPPED_RELU** or to specify the alpha coefficient when the activation mode is set to **CUDNN_ACTIVATION_ELU**.

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

Returns

CUDNN_STATUS_SUCCESS

The object was queried successfully.

4.79. cudnnDestroyActivationDescriptor

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

This function destroys a previously created activation descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.80. cudnnCreateLRNDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.81. cudnnSetLRNDescriptor

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

This function initializes a previously created LRN descriptor object.

> Macros CUDNN_LRN_MIN_N, CUDNN_LRN_MAX_N, CUDNN_LRN_MIN_K, CUDNN_LRN_MIN_BETA defined in cudnn.h specify valid ranges for parameters.

> Values of double parameters will be cast down to the tensor datatype during computation.

Parameters

normDesc

Output. Handle to a previously created LRN descriptor.

lrnN

Input. Normalization window width in elements. LRN layer uses a window [center-lookBehind, center+lookAhead], where lookBehind = floor((lrnN-1)/2), lookAhead = lrnN-lookBehind-1. So for n=10, the window is [k-4...k...k+5] with a total of 10 samples. For DivisiveNormalization layer the window has the same extents as above in all 'spatial' dimensions (dimA[2], dimA[3], dimA[4]). By default lrnN is set to 5 in cudnnCreateLRNDescriptor.

lrnAlpha

Input. Value of the alpha variance scaling parameter in the normalization formula. Inside the library code this value is divided by the window width for LRN and by (window width)^#spatialDimensions for DivisiveNormalization. By default this value is set to 1e-4 in cudnnCreateLRNDescriptor.

lrnBeta

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

lrnK

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

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

4.82. cudnnGetLRNDescriptor

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

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

Parameters

normDesc

Output. Handle to a previously created LRN descriptor.

lrnN, lrnAlpha, lrnBeta, lrnK

Output. Pointers to receive values of parameters stored in the descriptor object. See cudnnSetLRNDescriptor for more details. Any of these pointers can be NULL (no value is returned for the corresponding parameter).

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

Returns

CUDNN_STATUS_SUCCESS

Function completed successfully.

4.83. cudnnDestroyLRNDescriptor

```
cudnnStatus_t cudnnDestroyLRNDescriptor(
    cudnnLRNDescriptor_t lrnDesc)
```

This function destroys a previously created LRN descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.84. cudnnLRNCrossChannelForward

```
cudnnStatus_t cudnnLRNCrossChannelForward(
    cudnnHandle_t                      handle,
    cudnnLRNDescriptor_t               normDesc,
    cudnnLRNMode_t                     lrnMode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const void                        *beta,
    const cudnnTensorDescriptor_t      yDesc,
    void                              *y)
```

This function performs the forward LRN layer computation.

> Supported formats are: positive-strided, NCHW for 4D x and y, and only NCDHW DHW-packed for 5D (for both x and y). Only non-overlapping 4D and 5D tensors are supported.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

normDesc

Input. Handle to a previously intialized 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. Please refer to this section for additional details.

xDesc, yDesc

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

x

Input. Input tensor data pointer in device memory.

y

Output. Output tensor data pointer in device memory.

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

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

`CUDNN_STATUS_NOT_SUPPORTED`

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

- Any of the input tensor datatypes is not the same as any of the output tensor datatype.
- x and y tensor dimensions mismatch.
- Any tensor parameters strides are negative.

4.85. cudnnLRNCrossChannelBackward

```
cudnnStatus_t cudnnLRNCrossChannelBackward(
    cudnnHandle_t                      handle,
    cudnnLRNDescriptor_t               normDesc,
    cudnnLRNMode_t                     lrnMode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      yDesc,
    const void                        *y,
    const cudnnTensorDescriptor_t      dyDesc,
    const void                        *dy,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const void                        *beta,
    const cudnnTensorDescriptor_t      dxDesc,
    void                              *dx)
```

This function performs the backward LRN layer computation.

> Supported formats are: positive-strided, NCHW for 4D x and y, and only NCDHW DHW-packed for 5D (for both x and y). Only non-overlapping 4D and 5D tensors are supported.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

normDesc

Input. Handle to a previously intialized 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. Please refer to this section for additional details.

yDesc, y

Input. Tensor descriptor and pointer in device memory for the layer's y data.

dyDesc, dy

Input. Tensor descriptor and pointer in device memory for the layer's input cumulative loss differential data dy (including error backpropagation).

xDesc, x

Input. Tensor descriptor and pointer in device memory for the layer's x data. Note that these values are not modified during backpropagation.

dxDesc, dx

Output. Tensor descriptor and pointer in device memory for the layer's resulting cumulative loss differential data dx (including error backpropagation).

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the tensor pointers **`x, y`** is NULL.
- Number of input tensor dimensions is 2 or less.

- LRN descriptor parameters are outside of their valid ranges.
- One of tensor parameters is 5D but is not in NCDHW DHW-packed format.

CUDNN_STATUS_NOT_SUPPORTED

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

- Any of the input tensor datatypes is not the same as any of the output tensor datatype.
- Any pairwise tensor dimensions mismatch for x,y,dx,dy.
- Any tensor parameters strides are negative.

4.86. cudnnDivisiveNormalizationForward

```
cudnnStatus_t cudnnDivisiveNormalizationForward(
    cudnnHandle_t                   handle,
    cudnnLRNDescriptor_t            normDesc,
    cudnnDivNormMode_t              mode,
    const void                     *alpha,
    const cudnnTensorDescriptor_t   xDesc,
    const void                     *x,
    const void                     *means,
    void                           *temp,
    void                           *temp2,
    const void                     *beta,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y)
```

This function performs the forward spatial DivisiveNormalization layer computation. It divides every value in a layer by the standard deviation of it's spatial neighbors as described in *"What is the Best Multi-Stage Architecture for Object Recognition"*, Jarrett 2009, Local Contrast Normalization Layer section. Note that Divisive Normalization only implements the x/max(c, sigma_x) portion of the computation, where sigma_x is the variance over the spatial neighborhood of x. The full LCN (Local Contrastive Normalization) computation can be implemented as a two-step process:

x_m = x-mean(x);

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

The "x-mean(x)" which is often referred to as "subtractive normalization" portion of the computation can be implemented using cuDNN average pooling layer followed by a call to addTensor.

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

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

normDesc

Input. Handle to a previously intialized 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. Please refer to this section for additional details.

xDesc, yDesc

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

x

Input. Input tensor data pointer in device memory.

means

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

temp, temp2

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

y

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the tensor pointers **`x, y, temp, temp2`** is NULL.
- Number of input tensor or output tensor dimensions is outside of [4,5] range.

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

CUDNN_STATUS_UNSUPPORTED

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

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

4.87. cudnnDivisiveNormalizationBackward

```
cudnnStatus_t cudnnDivisiveNormalizationBackward(
    cudnnHandle_t                      handle,
    cudnnLRNDescriptor_t               normDesc,
    cudnnDivNormMode_t                 mode,
    const void                        *alpha,
    const cudnnTensorDescriptor_t      xDesc,
    const void                        *x,
    const void                        *means,
    const void                        *dy,
    void                              *temp,
    void                              *temp2,
    const void                        *beta,
    const cudnnTensorDescriptor_t      dxDesc,
    void                              *dx,
    void                              *dMeans)
```

This function performs the backward DivisiveNormalization layer computation.

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

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

normDesc

Input. Handle to a previously intialized 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. Please refer to this section for additional details.

xDesc, x, means

Input. Tensor descriptor and pointers in device memory for the layer's x and means data. Note: the means tensor is expected to be precomputed by the user. It can also contain any valid values (not required to be actual means, and can be for instance a result of a convolution with a Gaussian kernel).

dy

Input. Tensor pointer in device memory for the layer's dy cumulative loss differential data (error backpropagation).

temp, temp2

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

dxDesc

Input. Tensor descriptor for dx and dMeans.

dx, dMeans

Output. Tensor pointers (in device memory) for the layer's resulting cumulative gradients dx and dMeans (dLoss/dx and dLoss/dMeans). Both share the same descriptor.

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the tensor pointers **`x, dx, temp, tmep2, dy`** is NULL.
- Number of any of the input or output tensor dimensions is not within the [4,5] range.
- Either alpha or beta pointer is NULL.
- A mismatch in dimensions between xDesc and dxDesc.
- LRN descriptor parameters are outside of their valid ranges.
- Any of the tensor strides is negative.

`CUDNN_STATUS_UNSUPPORTED`

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

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

4.88. cudnnBatchNormalizationForwardInference

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

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

> Only 4D and 5D tensors are supported.

> The input transformation performed by this function is defined as: y := alpha*y + beta *(bnScale * (x-estimatedMean)/sqrt(epsilon + estimatedVariance)+bnBias)

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

> For training phase use cudnnBatchNormalizationForwardTraining.

> Much higher performance when HW-packed tensors are used for all of x, dy, dx.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

mode

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

alpha, beta

Inputs. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows: dstValue = alpha[0]*resultValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc, yDesc, x, y

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

bnScaleBiasMeanVarDesc, bnScaleData, bnBiasData

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

estimatedMean, estimatedVariance

Inputs. Mean and variance tensors (these have the same descriptor as the bias and scale). It is suggested that resultRunningMean, resultRunningVariance from the cudnnBatchNormalizationForwardTraining call accumulated during the training phase are passed as inputs here.

epsilon

Input. Epsilon value used in the batch normalization formula. Minimum allowed value is CUDNN_BN_MIN_EPSILON defined in cudnn.h.

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the pointers **`alpha, beta, x, y, bnScaleData, bnBiasData, estimatedMean, estimatedInvVariance`** is NULL.
- Number of xDesc or yDesc tensor descriptor dimensions is not within the [4,5] range.
- bnScaleBiasMeanVarDesc dimensions are not 1xC(x1)x1x1 for spatial or 1xC(xD)xHxW for per-activation mode (parenthesis for 5D).
- epsilon value is less than CUDNN_BN_MIN_EPSILON
- Dimensions or data types mismatch for xDesc, yDesc

4.89. cudnnBatchNormalizationForwardTraining

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

```
    const void                    *bnBias,
    double                         exponentialAverageFactor,
    void                          *resultRunningMean,
    void                          *resultRunningVariance,
    double                         epsilon,
    void                          *resultSaveMean,
    void                          *resultSaveInvVariance)
```

This function performs the forward BatchNormalization layer computation for training phase.

> Only 4D and 5D tensors are supported.

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

> For inference phase use cudnnBatchNormalizationForwardInference.

> Much higher performance for HW-packed tensors for both x and y.

Parameters

handle

Handle to a previously created cuDNN library descriptor.

mode

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

alpha, beta

Inputs. Pointers to scaling factors (in host memory) used to blend the layer output value with prior value in the destination tensor as follows: dstValue = alpha[0]*resultValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc, yDesc, x, y

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

bnScaleBiasMeanVarDesc

Shared tensor descriptor desc for all the 6 tensors below in the argument list. 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 original paper bias is referred to as beta and scale as gamma). Note that bnBias parameter can replace the previous layer's bias parameter for improved efficiency.

exponentialAverageFactor

Input. Factor used in the moving average computation runningMean = newMean*factor + runningMean*(1-factor). Use a factor=1/(1+n) at N-th call to the function to get Cumulative Moving Average (CMA) behavior CMA[n] = (x[1]+...

+x[n])/n. Since 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)

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 moving average of variance[x] where variance is computed either over batch or spatial+batch dimensions depending on the mode. If these pointers are not NULL, the tensors should be initialized to some reasonable values or to 0.

epsilon

Epsilon value used in the batch normalization formula. Minimum allowed value is CUDNN_BN_MIN_EPSILON defined in cudnn.h. Same epsilon value should be used in forward and backward functions.

resultSaveMean, resultSaveInvVariance

Outputs. Optional cache to save intermediate results computed during the forward pass - these can then be reused to speed up the backward pass. For this to work correctly, the bottom layer data has to remain unchanged until the backward function is called. Note that both of these parameters can be NULL but only at the same time. It is recommended to use this cache since memory overhead is relatively small because these tensors have a much lower product of dimensions than the data tensors.

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- One of the pointers **`alpha, beta, x, y, bnScaleData, bnBiasData`** is NULL.
- Number of xDesc or yDesc tensor descriptor dimensions is not within the [4,5] range.
- bnScaleBiasMeanVarDesc dimensions are not 1xC(x1)x1x1 for spatial or 1xC(xD)xHxW for per-activation mode (parens for 5D).
- Exactly one of resultSaveMean, resultSaveInvVariance pointers is NULL.
- Exactly one of resultRunningMean, resultRunningInvVariance pointers is NULL.
- epsilon value is less than CUDNN_BN_MIN_EPSILON
- Dimensions or data types mismatch for xDesc, yDesc

4.90. cudnnBatchNormalizationBackward

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

This function performs the backward BatchNormalization layer computation.

Only 4D and 5D tensors are supported.

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

Much higher performance when HW-packed tensors are used for all of x, dy, dx.

Parameters

handle

Handle to a previously created cuDNN library descriptor.

mode

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

alphaDataDiff, betaDataDiff

Inputs. Pointers to scaling factors (in host memory) used to blend the gradient output dx with a prior value in the destination tensor as follows: dstValue = alpha[0]*resultValue + beta[0]*priorDstValue. Please refer to this section for additional details.

alphaParamDiff, betaParamDiff

Inputs. Pointers to scaling factors (in host memory) used to blend the gradient outputs dBnScaleResult and dBnBiasResult with prior values in the destination tensor as follows: dstValue = alpha[0]*resultValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc, x, dyDesc, dy, dxDesc, dx

Tensor descriptors and pointers in device memory for the layer's x data, backpropagated differential dy (inputs) and resulting differential with respect to x, dx (output).

bnScaleBiasDiffDesc

Shared tensor descriptor for all the 5 tensors below in the argument list (bnScale, resultBnScaleDiff, resultBnBiasDiff, savedMean, 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.

bnScale

Input. Pointers in device memory for the batch normalization scale parameter (in original paper bias is referred to as gamma). Note that 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 scale and bias gradients are not backpropagated below this layer (since they are dead-end computation DAG nodes).

epsilon

Epsilon value used in batch normalization formula. Minimum allowed value is CUDNN_BN_MIN_EPSILON defined in cudnn.h. Same epsilon value should be used in forward and backward functions.

savedMean, savedInvVariance

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The computation was performed successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- Any of the pointers **`alpha, beta, x, dy, dx, bnScale, resultBnScaleDiff, resultBnBiasDiff`** is NULL.
- Number of xDesc or yDesc or dxDesc tensor descriptor dimensions is not within the [4,5] range.
- bnScaleBiasMeanVarDesc dimensions are not 1xC(x1)x1x1 for spatial or 1xC(xD)xHxW for per-activation mode (parentheses for 5D).

- Exactly one of savedMean, savedInvVariance pointers is NULL.
- epsilon value is less than CUDNN_BN_MIN_EPSILON
- Dimensions or data types mismatch for any pair of xDesc, dyDesc, dxDesc

4.91. cudnnDeriveBNTensorDescriptor

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

Derives a secondary tensor descriptor for BatchNormalization scale, invVariance, bnBias, bnScale subtensors from the layer's x data descriptor. Use the tensor descriptor produced by this function as the bnScaleBiasMeanVarDesc and bnScaleBiasDiffDesc parameters in Spatial and Per-Activation Batch Normalization forward and backward functions. Resulting dimensions will be **1xC(x1)x1x1** for BATCHNORM_MODE_SPATIAL and **1xC(xD)xHxW** for BATCHNORM_MODE_PER_ACTIVATION (parentheses for 5D). 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.

> derivedBnDesc has to be first created using cudnnCreateTensorDescriptor

> xDesc is the descriptor for the layer's x data and has to be setup with proper dimensions prior to calling this function.

Parameters

derivedBnDesc

Output. Handle to a previously created tensor descriptor.

xDesc

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

mode

Input. Batch normalization layer mode of operation.

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

Returns

CUDNN_STATUS_SUCCESS

The computation was performed successfully.

CUDNN_STATUS_BAD_PARAM

Invalid Batch Normalization mode.

4.92. cudnnCreateRNNDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.93. cudnnDestroyRNNDescriptor

```
cudnnStatus_t cudnnDestroyRNNDescriptor(
    cudnnRNNDescriptor_t rnnDesc)
```

This function destroys a previously created RNN descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.94. cudnnCreatePersistentRNNPlan

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

This function creates a plan to execute persistent RNNs when using the **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** algo. This plan is tailored to the current GPU and problem hyperparemeters. This function call is expected to be expensive in terms of runtime, and should be used infrequently.

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING

A prerequisite runtime library cannot be found.

CUDNN_STATUS_NOT_SUPPORTED

The current hyperparameters are invalid.

4.95. cudnnSetPersistentRNNPlan

```
cudnnStatus_t cudnnSetPersistentRNNPlan(
    cudnnRNNDescriptor_t          rnnDesc,
    cudnnPersistentRNNPlan_t      plan)
```

This function sets the persistent RNN plan to be executed when using **rnnDesc** and **CUDNN_RNN_ALGO_PERSIST_DYNAMIC** algo.

Returns

CUDNN_STATUS_SUCCESS

The plan was set successfully.

CUDNN_STATUS_BAD_PARAM

The algo selected in **rnnDesc** is not **CUDNN_RNN_ALGO_PERSIST_DYNAMIC**.

4.96. cudnnDestroyPersistentRNNPlan

```
cudnnStatus_t cudnnDestroyPersistentRNNPlan(
    cudnnPersistentRNNPlan_t plan)
```

This function destroys a previously created persistent RNN plan object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.97. cudnnSetRNNDescriptor

```
cudnnStatus_t cudnnSetRNNDescriptor(
    cudnnHandle_t                 cudnnHandle,
    cudnnRNNDescriptor_t          rnnDesc,
    int                           hiddenSize,
    int                           numLayers,
    cudnnDropoutDescriptor_t      dropoutDesc,
    cudnnRNNInputMode_t           inputMode,
    cudnnDirectionMode_t          direction,
    cudnnRNNMode_t                mode,
    cudnnRNNAlgo_t                algo,
    cudnnDataType_t               dataType)
```

This function initializes a previously created RNN descriptor object.

> Larger networks (e.g., longer sequences, more layers) are expected to be more efficient than smaller networks.

Parameters

rnnDesc

Input/Output. A previously created RNN descriptor.

hiddenSize

Input. Size of the internal hidden state for each layer.

numLayers

Input. Number of stacked layers.

dropoutDesc

Input. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers; a single layer network will have no dropout applied.

inputMode

Input. Specifies the behavior at the input to the first layer.

direction

Input. Specifies the recurrence pattern. (e.g., bidirectional).

mode

Input. Specifies the type of RNN to compute.

dataType

Input. Math precision.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

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

4.98. cudnnSetRNNDescriptor_v6

```
cudnnStatus_t cudnnSetRNNDescriptor_v6(
    cudnnHandle_t               cudnnHandle,
    cudnnRNNDescriptor_t        rnnDesc,
    int                         hiddenSize,
    int                         numLayers,
    cudnnDropoutDescriptor_t    dropoutDesc,
    cudnnRNNInputMode_t         inputMode,
    cudnnDirectionMode_t        direction,
    cudnnRNNMode_t              mode,
    cudnnRNNAlgo_t              algo,
    cudnnDataType_t             dataType)
```

This function initializes a previously created RNN descriptor object.

> Larger networks (e.g., longer sequences, more layers) are expected to be more efficient than smaller networks.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input/Output. A previously created RNN descriptor.

hiddenSize

Input. Size of the internal hidden state for each layer.

numLayers

Input. Number of stacked layers.

dropoutDesc

Input. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers (e.g., a single layer network will have no dropout applied).

inputMode

Input. Specifies the behavior at the input to the first layer

direction

Input. Specifies the recurrence pattern. (e.g., bidirectional)

mode

Input. Specifies the type of RNN to compute.

algo

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

dataType

Input. Compute precision.

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

Returns

`CUDNN_STATUS_SUCCESS`

The object was set successfully.

`CUDNN_STATUS_BAD_PARAM`

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

4.99. cudnnSetRNNDescriptor_v5

```
cudnnStatus_t cudnnSetRNNDescriptor_v5(
    cudnnRNNDescriptor_t     rnnDesc,
    int                      hiddenSize,
    int                      numLayers,
    cudnnDropoutDescriptor_t dropoutDesc,
    cudnnRNNInputMode_t      inputMode,
    cudnnDirectionMode_t     direction,
    cudnnRNNMode_t           mode,
    cudnnDataType_t          dataType)
```

This function initializes a previously created RNN descriptor object.

Larger networks (e.g., longer sequences, more layers) are expected to be more efficient than smaller networks.

Parameters

rnnDesc

Input/Output. A previously created RNN descriptor.

hiddenSize

Input. Size of the internal hidden state for each layer.

numLayers

Input. Number of stacked layers.

dropoutDesc

Input. Handle to a previously created and initialized dropout descriptor. Dropout will be applied between layers (e.g., a single layer network will have no dropout applied).

inputMode

Input. Specifies the behavior at the input to the first layer

direction

Input. Specifies the recurrence pattern. (e.g., bidirectional)

mode

Input. Specifies the type of RNN to compute.

dataType

Input. Compute precision.

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

Returns

CUDNN_STATUS_SUCCESS

The object was set successfully.

CUDNN_STATUS_BAD_PARAM

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

4.100. cudnnGetRNNWorkspaceSize

```
cudnnStatus_t cudnnGetRNNWorkspaceSize(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       seqLength,
    const cudnnTensorDescriptor_t  *xDesc,
    size_t                         *sizeInBytes)
```

This function is used to query the amount of work space required to execute the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input. A previously initialized RNN descriptor.

seqLength

Input. Number of iterations to unroll over.

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 workspace to be able to execute an RNN with the specified descriptor and input tensors.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors in **xDesc** is invalid.
- The descriptors in **xDesc** have inconsistent second dimensions, strides or data types.

- The descriptors in **`xDesc`** have increasing first dimensions.
- The descriptors in **`xDesc`** is not fully packed.

`CUDNN_STATUS_NOT_SUPPORTED`

The data types in tensors described by xDesc is not supported.

4.101. cudnnGetRNNTrainingReserveSize

```
cudnnStatus_t cudnnGetRNNTrainingReserveSize(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const int                          seqLength,
    const cudnnTensorDescriptor_t     *xDesc,
    size_t                            *sizeInBytes)
```

This function is used to query the amount of reserved space required for training the RNN described by **`rnnDesc`** with inputs dimensions defined by **`xDesc`**. The same reserved space buffer must be passed to **`cudnnRNNForwardTraining`**, **`cudnnRNNBackwardData`** and **`cudnnRNNBackwardWeights`**. Each of these calls overwrites the contents of the reserved space, however it can safely be backed up and restored between calls if reuse of the memory is desired.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input. A previously initialized RNN descriptor.

seqLength

Input. Number of iterations to unroll over.

xDesc

Input. An array of tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **`n`** to element **`n+1`** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

sizeInBytes

Output. Minimum amount of GPU memory needed as reserve space to be able to train an RNN with the specified descriptor and input tensors.

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors in **xDesc** is invalid.
- The descriptors in **xDesc** have inconsistent second dimensions, strides or data types.
- The descriptors in **xDesc** have increasing first dimensions.
- The descriptors in **xDesc** is not fully packed.

CUDNN_STATUS_NOT_SUPPORTED

The the data types in tensors described by xDesc is not supported.

4.102. cudnnGetRNNParamsSize

```
cudnnStatus_t cudnnGetRNNParamsSize(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const cudnnTensorDescriptor_t    xDesc,
    size_t                          *sizeInBytes,
    cudnnDataType_t                  dataType)
```

This function is used to query the amount of parameter space required to execute the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input. A previously initialized RNN descriptor.

xDesc

Input. A fully packed tensor descriptor describing the input to one recurrent iteration.

sizeInBytes

Output. Minimum amount of GPU memory needed as parameter space to be able to execute an RNN with the specified descriptor and input tensors.

dataType

Input. The data type of the parameters.

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- The descriptor **xDesc** is invalid.
- The descriptor **xDesc** is not fully packed.

- The combination of **dataType** and tensor descriptor data type is invalid.

CUDNN_STATUS_NOT_SUPPORTED

The combination of the RNN descriptor and tensor descriptors is not supported.

4.103. cudnnGetRNNLinLayerMatrixParams

```
cudnnStatus_t cudnnGetRNNLinLayerMatrixParams(
cudnnHandle_t                    handle,
const cudnnRNNDescriptor_t       rnnDesc,
const int                        layer,
const cudnnTensorDescriptor_t    xDesc,
const cudnnFilterDescriptor_t    wDesc,
const void                      *w,
const int                        linLayerID,
cudnnFilterDescriptor_t          linLayerMatDesc,
void                           **linLayerMat)
```

This function is used to obtain a pointer and descriptor for the matrix parameters in **layer** within the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input. A previously initialized RNN descriptor.

layer

Input. The layer to query.

xDesc

Input. A fully packed tensor descriptor describing the input to one recurrent iteration.

wDesc

Input. Handle to a previously initialized filter descriptor describing the weights for the RNN.

w

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

linLayerID

Input. The linear layer to obtain information about:

- If **mode** in **rnnDesc** was set to **CUDNN_RNN_RELU** or **CUDNN_RNN_TANH** a value of 0 references the matrix multiplication applied to the input from the previous layer, a value of 1 references the matrix multiplication applied to the recurrent input.
- If **mode** in **rnnDesc** was set to **CUDNN_LSTM** values of 0-3 reference matrix multiplications applied to the input from the previous layer, value of 4-7 reference matrix multiplications applied to the recurrent input.
    - Values 0 and 4 reference the input gate.

    - Values 1 and 5 reference the forget gate.
    - Values 2 and 6 reference the new memory gate.
    - Values 3 and 7 reference the output gate.
- If **mode** in **rnnDesc** was set to **CUDNN_GRU** values of 0-2 reference matrix multiplications applied to the input from the previous layer, value of 3-5 reference matrix multiplications applied to the recurrent input.
    - Values 0 and 3 reference the reset gate.
    - Values 1 and 4 reference the update gate.
    - Values 2 and 5 reference the new memory gate.

Please refer to this section for additional details on modes.

linLayerMatDesc

Output. Handle to a previously created filter descriptor.

linLayerMat

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

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- One of the descriptors **xDesc, wDesc, linLayerMatDesc** is invalid.
- One of **layer, linLayerID** is invalid.

4.104. cudnnGetRNNLinLayerBiasParams

```
cudnnStatus_t cudnnGetRNNLinLayerBiasParams(
    cudnnHandle_t                  handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       layer,
    const cudnnTensorDescriptor_t   xDesc,
    const cudnnFilterDescriptor_t   wDesc,
    const void                     *w,
    const int                       linLayerID,
    cudnnFilterDescriptor_t         linLayerBiasDesc,
    void                           **linLayerBias)
```

This function is used to obtain a pointer and descriptor for the bias parameters in **layer** within the RNN described by **rnnDesc** with inputs dimensions defined by **xDesc**.

Parameters

handle

Input. Handle to a previously created cuDNN library descriptor.

rnnDesc

Input. A previously initialized RNN descriptor.

layer

Input. The layer to query.

xDesc

Input. A fully packed tensor descriptor describing the input to one recurrent iteration.

wDesc

Input. Handle to a previously initialized filter descriptor describing the weights for the RNN.

w

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

linLayerID

Input. The linear layer to obtain information about:

- If **`mode`** in **`rnnDesc`** was set to **`CUDNN_RNN_RELU`** or **`CUDNN_RNN_TANH`** a value of 0 references the bias applied to the input from the previous layer, a value of 1 references the bias applied to the recurrent input.
- If **`mode`** in **`rnnDesc`** was set to **`CUDNN_LSTM`** values of 0, 1, 2 and 3 reference bias applied to the input from the previous layer, value of 4, 5, 6 and 7 reference bias applied to the recurrent input.
  - Values 0 and 4 reference the input gate.
  - Values 1 and 5 reference the forget gate.
  - Values 2 and 6 reference the new memory gate.
  - Values 3 and 7 reference the output gate.
- If **`mode`** in **`rnnDesc`** was set to **`CUDNN_GRU`** values of 0, 1 and 2 reference bias applied to the input from the previous layer, value of 3, 4 and 5 reference bias applied to the recurrent input.
  - Values 0 and 3 reference the reset gate.
  - Values 1 and 4 reference the update gate.
  - Values 2 and 5 reference the new memory gate.

Please refer to this section for additional details on modes.

linLayerBiasDesc

Output. Handle to a previously created filter descriptor.

linLayerBias

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The descriptor **`rnnDesc`** is invalid.
- One of the descriptors **`xDesc, wDesc, linLayerBiasDesc`** is invalid.
- One of **`layer, linLayerID`** is invalid.

4.105. cudnnRNNForwardInference

```
cudnnStatus_t cudnnRNNForwardInference(
    cudnnHandle_t                      handle,
    const cudnnRNNDescriptor_t         rnnDesc,
    const int                          seqLength,
    const cudnnTensorDescriptor_t     *xDesc,
    const void                        *x,
    const cudnnTensorDescriptor_t      hxDesc,
    const void                        *hx,
    const cudnnTensorDescriptor_t      cxDesc,
    const void                        *cx,
    const cudnnFilterDescriptor_t      wDesc,
    const void                        *w,
    const cudnnTensorDescriptor_t      *yDesc,
    void                              *y,
    const cudnnTensorDescriptor_t      hyDesc,
    void                              *hy,
    const cudnnTensorDescriptor_t      cyDesc,
    void                              *cy,
    void                              *workspace,
    size_t                             workSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **`rnnDesc`** with inputs **`x, hx, cx`**, weights **`w`** and outputs **`y, hy, cy`**. **`workspace`** is required for intermediate storage. This function does not store intermediate data required for training; **`cudnnRNNForwardTraining`** should be used for that purpose.

Parameters

handle

Input. Handle to a previously created cuDNN context.

rnnDesc

Input. A previously initialized RNN descriptor.

seqLength

Input. Number of iterations to unroll over.

xDesc

Input. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

x

Input. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**. The data are expected to be packed contiguously with the first element of iteration n+1 following directly from the last element of iteration n.

hxDesc

Input. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

hx

Input. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a NULL pointer is passed, the initial hidden state of the network will be initialized to zero.

cxDesc

Input. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

cx

Input. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a NULL pointer is passed, the initial cell state of the network will be initialized to zero.

wDesc

Input. Handle to a previously initialized filter descriptor describing the weights for the RNN.

w

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

yDesc

Input. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **`direction`** argument passed to the **`cudnnSetRNNDescriptor`** call used to initialize **`rnnDesc`**:

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

The first dimension of the tensor **`n`** must match the first dimension of the tensor **`n`** in **`xDesc`**.

y

Output. Data pointer to GPU memory associated with the output tensor descriptor **`yDesc`**. The data are expected to be packed contiguously with the first element of iteration n+1 following directly from the last element of iteration n.

hyDesc

Input. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **`direction`** argument passed to the **`cudnnSetRNNDescriptor`** call used to initialize **`rnnDesc`**:

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

The second dimension must match the first dimension of the tensors described in **`xDesc`**. The third dimension must match the **`hiddenSize`** argument passed to the **`cudnnSetRNNDescriptor`** call used to initialize **`rnnDesc`**. The tensor must be fully packed.

hy

Output. Data pointer to GPU memory associated with the tensor descriptor **`hyDesc`**. If a NULL pointer is passed, the final hidden state of the network will not be saved.

cyDesc

Input. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **`direction`** argument passed to the **`cudnnSetRNNDescriptor`** call used to initialize **`rnnDesc`**:

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

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

cy

Output. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a NULL pointer is passed, the final cell state of the network will be not be saved.

workspace

Input. Data pointer to GPU memory to be used as a workspace for this call.

workSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **hxDesc, cxDesc, wDesc, hyDesc, cyDesc** or one of the descriptors in **xDesc, yDesc** is invalid.
- The descriptors in one of **xDesc, hxDesc, cxDesc, wDesc, yDesc, hyDesc, cyDesc** have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

CUDNN_STATUS_ALLOC_FAILED

The function was unable to allocate memory.

4.106. cudnnRNNForwardTraining

```
cudnnStatus_t cudnnRNNForwardTraining(
    cudnnHandle_t                   handle,
    const cudnnRNNDescriptor_t      rnnDesc,
    const int                       seqLength,
    const cudnnTensorDescriptor_t  *xDesc,
    const void                     *x,
    const cudnnTensorDescriptor_t   hxDesc,
```

```
    const void                     *hx,
    const cudnnTensorDescriptor_t   cxDesc,
    const void                     *cx,
    const cudnnFilterDescriptor_t   wDesc,
    const void                     *w,
    const cudnnTensorDescriptor_t  *yDesc,
    void                           *y,
    const cudnnTensorDescriptor_t   hyDesc,
    void                           *hy,
    const cudnnTensorDescriptor_t   cyDesc,
    void                           *cy,
    void                           *workspace,
    size_t                          workSpaceSizeInBytes,
    void                           *reserveSpace,
    size_t                          reserveSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **rnnDesc** with inputs **x, hx, cx**, weights **w** and outputs **y, hy, cy**. **workspace** is required for intermediate storage. **reserveSpace** stores data required for training. The same reserveSpace data must be used for future calls to **cudnnRNNBackwardData** and **cudnnRNNBackwardWeights** if these execute on the same input data.

Parameters

handle

Input. Handle to a previously created cuDNN context.

rnnDesc

Input. A previously initialized RNN descriptor.

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.

x

Input. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**.

hxDesc

Input. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the

cudnnSetRNNDescriptor call used to initialize **rnnDesc**. The tensor must be fully packed.

hx

Input. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a NULL pointer is passed, the initial hidden state of the network will be initialized to zero.

cxDesc

Input. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

cx

Input. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a NULL pointer is passed, the initial cell state of the network will be initialized to zero.

wDesc

Input. Handle to a previously initialized filter descriptor describing the weights for the RNN.

w

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

yDesc

Input. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **xDesc**.

y

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

hyDesc

Input. A fully packed tensor descriptor describing the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

hy

Output. Data pointer to GPU memory associated with the tensor descriptor **hyDesc**. If a NULL pointer is passed, the final hidden state of the network will not be saved.

cyDesc

Input. A fully packed tensor descriptor describing the final cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

cy

Output. Data pointer to GPU memory associated with the tensor descriptor **cyDesc**. If a NULL pointer is passed, the final cell state of the network will be not be saved.

workspace

Input. Data pointer to GPU memory to be used as a workspace for this call.

workSpaceSizeInBytes

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

reserveSpace

Input/Output. Data pointer to GPU memory to be used as a reserve space for this call.

reserveSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **hxDesc, cxDesc, wDesc, hyDesc, cyDesc** or one of the descriptors in **xDesc, yDesc** is invalid.
- The descriptors in one of **xDesc, hxDesc, cxDesc, wDesc, yDesc, hyDesc, cyDesc** have incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

CUDNN_STATUS_ALLOC_FAILED

The function was unable to allocate memory.

4.107. cudnnRNNBackwardData

```
cudnnStatus_t cudnnRNNBackwardData(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t   *yDesc,
    const void                      *y,
    const cudnnTensorDescriptor_t   *dyDesc,
    const void                      *dy,
    const cudnnTensorDescriptor_t    dhyDesc,
    const void                      *dhy,
    const cudnnTensorDescriptor_t    dcyDesc,
    const void                      *dcy,
    const cudnnFilterDescriptor_t    wDesc,
    const void                      *w,
    const cudnnTensorDescriptor_t    hxDesc,
    const void                      *hx,
    const cudnnTensorDescriptor_t    cxDesc,
    const void                      *cx,
    const cudnnTensorDescriptor_t   *dxDesc,
    void                            *dx,
    const cudnnTensorDescriptor_t    dhxDesc,
    void                            *dhx,
    const cudnnTensorDescriptor_t    dcxDesc,
    void                            *dcx,
    void                            *workspace,
    size_t                           workSpaceSizeInBytes,
    const void                      *reserveSpace,
    size_t                           reserveSpaceSizeInBytes)
```

This routine executes the recurrent neural network described by **rnnDesc** with output gradients **dy, dhy, dhc**, weights **w** and input gradients **dx, dhx, dcx**. **workspace** is required for intermediate storage. The data in **reserveSpace** must have previously been generated by **cudnnRNNForwardTraining**. The same reserveSpace data must be used for future calls to **cudnnRNNBackwardWeights** if they execute on the same input data.

Parameters

handle

Input. Handle to a previously created cuDNN context.

rnnDesc

Input. A previously initialized RNN descriptor.

seqLength

Input. Number of iterations to unroll over.

yDesc

Input. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

y

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

dyDesc

Input. An array of fully packed tensor descriptors describing the gradient at the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The first dimension of the tensor **n** must match the second dimension of the tensor **n** in **dxDesc**.

dy

Input. Data pointer to GPU memory associated with the tensor descriptors in the array **dyDesc**.

dhyDesc

Input. A fully packed tensor descriptor describing the gradients at the final hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

dhy

Input. Data pointer to GPU memory associated with the tensor descriptor **dhyDesc**. If a NULL pointer is passed, the gradients at the final hidden state of the network will be initialized to zero.

dcyDesc

Input. A fully packed tensor descriptor describing the gradients at the final cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

dcy

Input. Data pointer to GPU memory associated with the tensor descriptor **dcyDesc**. If a NULL pointer is passed, the gradients at the final cell state of the network will be initialized to zero.

wDesc

Input. Handle to a previously initialized filter descriptor describing the weights for the RNN.

w

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

hxDesc

Input. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the second dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

hx

Input. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a NULL pointer is passed, the initial hidden state of the network will be initialized to zero.

cxDesc

Input. A fully packed tensor descriptor describing the initial cell state for LSTM networks. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the second dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

cx

Input. Data pointer to GPU memory associated with the tensor descriptor **cxDesc**. If a NULL pointer is passed, the initial cell state of the network will be initialized to zero.

dxDesc

Input. An array of fully packed tensor descriptors describing the gradient at the input of each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

dx

Output. Data pointer to GPU memory associated with the tensor descriptors in the array **dxDesc**.

dhxDesc

Input. A fully packed tensor descriptor describing the gradient at the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

dhx

Output. Data pointer to GPU memory associated with the tensor descriptor **dhxDesc**. If a NULL pointer is passed, the gradient at the hidden input of the network will not be set.

dcxDesc

Input. A fully packed tensor descriptor describing the gradient at the initial cell state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

dcx

Output. Data pointer to GPU memory associated with the tensor descriptor **dcxDesc**. If a NULL pointer is passed, the gradient at the cell input of the network will not be set.

workspace

Input. Data pointer to GPU memory to be used as a workspace for this call.

workSpaceSizeInBytes

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

reserveSpace

Input/Output. Data pointer to GPU memory to be used as a reserve space for this call.

reserveSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The function launched successfully.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The descriptor **rnnDesc** is invalid.
- At least one of the descriptors **dhxDesc, wDesc, hxDesc, cxDesc, dcxDesc, dhyDesc, dcyDesc** or one of the descriptors in **yDesc, dxdesc, dydesc** is invalid.
- The descriptors in one of **yDesc, dxDesc, dyDesc, dhxDesc, wDesc, hxDesc, cxDesc, dcxDesc, dhyDesc, dcyDesc** has incorrect strides or dimensions.
- **workSpaceSizeInBytes** is too small.
- **reserveSpaceSizeInBytes** is too small.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

CUDNN_STATUS_ALLOC_FAILED

The function was unable to allocate memory.

4.108. cudnnRNNBackwardWeights

```
cudnnStatus_t cudnnRNNBackwardWeights(
    cudnnHandle_t                    handle,
    const cudnnRNNDescriptor_t       rnnDesc,
    const int                        seqLength,
    const cudnnTensorDescriptor_t   *xDesc,
    const void                      *x,
    const cudnnTensorDescriptor_t    hxDesc,
    const void                      *hx,
    const cudnnTensorDescriptor_t   *yDesc,
    const void                      *y,
    const void                      *workspace,
    size_t                           workSpaceSizeInBytes,
    const cudnnFilterDescriptor_t    dwDesc,
    void                            *dw,
    const void                      *reserveSpace,
    size_t                           reserveSpaceSizeInBytes)
```

This routine accumulates weight gradients **dw** from the recurrent neural network described by **rnnDesc** with inputs **x, hx**, and outputs **y**. The mode of operation in this case is additive, the weight gradients calculated will be added to those already existing in **dw**. **workspace** is required for intermediate storage. The data in **reserveSpace** must have previously been generated by **cudnnRNNBackwardData**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

rnnDesc

Input. A previously initialized RNN descriptor.

seqLength

Input. Number of iterations to unroll over.

xDesc

Input. An array of fully packed tensor descriptors describing the input to each recurrent iteration (one descriptor per iteration). The first dimension (batch size) of the tensors may decrease from element **n** to element **n+1** but may not increase. Each tensor descriptor must have the same second dimension (vector length).

x

Input. Data pointer to GPU memory associated with the tensor descriptors in the array **xDesc**.

hxDesc

Input. A fully packed tensor descriptor describing the initial hidden state of the RNN. The first dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The second dimension must match the first dimension of the tensors described in **xDesc**. The third dimension must match the **hiddenSize** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**. The tensor must be fully packed.

hx

Input. Data pointer to GPU memory associated with the tensor descriptor **hxDesc**. If a NULL pointer is passed, the initial hidden state of the network will be initialized to zero.

yDesc

Input. An array of fully packed tensor descriptors describing the output from each recurrent iteration (one descriptor per iteration). The second dimension of the tensor depends on the **direction** argument passed to the **cudnnSetRNNDescriptor** call used to initialize **rnnDesc**:

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

The first dimension of the tensor **n** must match the first dimension of the tensor **n** in **dyDesc**.

y

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

workspace

Input. Data pointer to GPU memory to be used as a workspace for this call.

workSpaceSizeInBytes

Input. Specifies the size in bytes of the provided **`workspace`**.

dwDesc

Input. Handle to a previously initialized filter descriptor describing the gradients of the weights for the RNN.

dw

Input/Output. Data pointer to GPU memory associated with the filter descriptor **`dwDesc`**.

reserveSpace

Input. Data pointer to GPU memory to be used as a reserve space for this call.

reserveSpaceSizeInBytes

Input. Specifies the size in bytes of the provided **`reserveSpace`**

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

Returns

`CUDNN_STATUS_SUCCESS`

The function launched successfully.

`CUDNN_STATUS_NOT_SUPPORTED`

The function does not support the provided configuration.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- The descriptor **`rnnDesc`** is invalid.
- At least one of the descriptors **`hxDesc, dwDesc`** or one of the descriptors in **`xDesc, yDesc`** is invalid.
- The descriptors in one of **`xDesc, hxDesc, yDesc, dwDesc`** has incorrect strides or dimensions.
- **`workSpaceSizeInBytes`** is too small.
- **`reserveSpaceSizeInBytes`** is too small.

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU.

`CUDNN_STATUS_ALLOC_FAILED`

The function was unable to allocate memory.

4.109. cudnnGetCTCLossWorkspaceSize

```
cudnnStatus_t cudnnGetCTCLossWorkspaceSize(
    cudnnHandle_t                        handle,
    const   cudnnTensorDescriptor_t      probsDesc,
    const   cudnnTensorDescriptor_t      gradientsDesc,
    const   int                         *labels,
    const   int                         *labelLengths,
    const   int                         *inputLengths,
    cudnnCTCLossAlgo_t                   algo,
    const   cudnnCTCLossDescriptor_t     ctcLossDesc,
    size_t                              *sizeInBytes)
```

This function returns the amount of GPU memory workspace the user needs to allocate to be able to call **cudnnCTCLoss** with the specified algorithm. The workspace allocated will then be passed to the routine **cudnnCTCLoss**.

Parameters

handle

Input. Handle to a previously created cuDNN context.

probsDesc

Input. Handle to the previously initialized probabilities tensor descriptor.

gradientsDesc

Input. Handle to a previously initialized gradients tensor descriptor.

labels

Input. Pointer to a previously initialized labels list.

labelLengths

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

inputLengths

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

algo

Input. Enumerant that specifies the chosen CTC loss algorithm

ctcLossDesc

Input. Handle to the previously initialized CTC loss descriptor.

sizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

CUDNN_STATUS_NOT_SUPPORTED

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

4.110. cudnnCTCLoss

```
cudnnStatus_t cudnnCTCLoss(
    cudnnHandle_t                        handle,
    const   cudnnTensorDescriptor_t      probsDesc,
    const   void                        *probs,
    const   int                         *labels,
    const   int                         *labelLengths,
    const   int                         *inputLengths,
    void                                *costs,
    const   cudnnTensorDescriptor_t      gradientsDesc,
    const   void                        *gradients,
    cudnnCTCLossAlgo_t                   algo,
    const   cudnnCTCLossDescriptor_t     ctcLossDesc,
    void                                *workspace,
    size_t                              *workSpaceSizeInBytes)
```

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

Parameters

handle

Input. Handle to a previously created cuDNN context.

probsDesc

Input. Handle to the previously initialized probabilities tensor descriptor.

probs

Input. Pointer to a previously initialized probabilities tensor.

labels

Input. Pointer to a previously initialized labels list.

labelLengths

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

inputLengths

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

costs

Output. Pointer to the computed costs of CTC.

gradientsDesc

Input. Handle to a previously initialized gradients tensor descriptor.

gradients

Output. Pointer to the computed gradients of CTC.

algo

Input. Enumerant that specifies the chosen CTC loss algorithm.

ctcLossDesc

Input. Handle to the previously initialized CTC loss descriptor.

workspace

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

sizeInBytes

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The query was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

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

`CUDNN_STATUS_NOT_SUPPORTED`

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

`CUDNN_STATUS_EXECUTION_FAILED`

The function failed to launch on the GPU

4.111. cudnnCreateDropoutDescriptor

```
cudnnStatus_t cudnnCreateDropoutDescriptor(
    cudnnRNNDescriptor_t    *rnnDesc)
```

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.112. cudnnDestroyDropoutDescriptor

```
cudnnStatus_t cudnnDestroyDropoutDescriptor(
    cudnnDropoutDescriptor_t rnnDesc)
```

This function destroys a previously created dropout descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.113. cudnnDropoutGetStatesSize

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

This function is used to query the amount of space required to store the states of the random number generators used by **cudnnDropoutForward** function.

Parameters

handle

Input. Handle to a previously created cuDNN context.

sizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

4.114. cudnnDropoutGetReserveSpaceSize

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

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

Parameters

xDesc

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

sizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The query was successful.

4.115. cudnnSetDropoutDescriptor

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

This function initializes a previously created dropout descriptor object. If **states** argument is equal to NULL, random number generator states won't be initialized, and only **dropout** value will be set. No other function should be writing to the memory pointed at by **states** argument while this function is running. The user is expected not to change memory pointed at by **states** for the duration of the computation.

Parameters

dropoutDesc

Input/Output. Previously created dropout descriptor object.

handle

Input. Handle to a previously created cuDNN context.

dropout

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

states

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

stateSizeInBytes

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

seed

Input. Seed used to initialize random number generator states.

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_INVALID_VALUE

sizeInBytes is less than the value returned by **cudnnDropoutGetStatesSize**.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU

4.116. cudnnGetDropoutDescriptor

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

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

Parameters

dropoutDesc

Input. Previously initialized dropout descriptor.

handle

Input. Handle to a previously created cuDNN context.

dropout

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

states

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

seed

Output. Seed used to initialize random number generator states.

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_BAD_PARAM

One or more of the arguments was an invalid pointer.

4.117. cudnnRestoreDropoutDescriptor

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

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

Parameters

dropoutDesc

Input/Output. Previously created dropout descriptor.

handle

Input. Handle to a previously created cuDNN context.

dropout

Input. Probability with which the value from an input tensor is set to 0 when performing dropout.

states

Input. Pointer to GPU memory that holds random number generator states initialized by a prior call to **cudnnSetDropoutDescriptor**.

stateSizeInBytes

Input. Size in bytes of buffer holding random number generator states.

seed

Input. Seed used in prior call to **cudnnSetDropoutDescriptor** that initialized 'states' buffer. Using a different seed from this has no effect. A change of seed, and subsequent update to random number generator states can be achieved by calling **cudnnSetDropoutDescriptor**.

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_INVALID_VALUE

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

4.118. cudnnDropoutForward

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

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

Better performance is obtained for fully packed tensors

Should not be called during inference

Parameters

handle

Input. Handle to a previously created cuDNN context.

dropoutDesc

Input. Previously created dropout descriptor object.

xDesc

Input. Handle to a previously initialized tensor descriptor.

x

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

yDesc

Input. Handle to a previously initialized tensor descriptor.

y

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

reserveSpace

Output. Pointer to user-allocated GPU memory used by this function. It is expected that contents of **reserveSpace** doe not change between **cudnnDropoutForward** and **cudnnDropoutBackward** calls.

reserveSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The number of elements of input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differs.
- The strides of the input tensor and output tensors differ and in-place operation is used (i.e., **x** and **y** pointers are equal).
- The provided **reserveSpaceSizeInBytes** is less then the value returned by **cudnnDropoutGetReserveSpaceSize**.
- **cudnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-NULL **states** argument.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.119. cudnnDropoutBackward

```
cudnnStatus_t cudnnDropoutBackward(
    cudnnHandle_t                    handle,
    const cudnnDropoutDescriptor_t  dropoutDesc,
    const cudnnTensorDescriptor_t   dydesc,
    const void                     *dy,
    const cudnnTensorDescriptor_t   dxdesc,
    void                           *dx,
    void                           *reserveSpace,
    size_t                          reserveSpaceSizeInBytes)
```

This function performs backward dropout operation over **dy** returning results in **dx**. If during forward dropout operation value from **x** was propagated to **y** then during backward operation value from **dy** will be propagated to **dx**, otherwise, **dx** value will be set to **0**.

Better performance is obtained for fully packed tensors

Parameters

handle

Input. Handle to a previously created cuDNN context.

dropoutDesc

Input. Previously created dropout descriptor object.

dyDesc

Input. Handle to a previously initialized tensor descriptor.

dy

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

dxDesc

Input. Handle to a previously initialized tensor descriptor.

dx

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

reserveSpace

Input. Pointer to user-allocated GPU memory used by this function. It is expected that **reserveSpace** was populated during a call to **cudnnDropoutForward** and has not been changed.

reserveSpaceSizeInBytes

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_NOT_SUPPORTED

The function does not support the provided configuration.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- The number of elements of input tensor and output tensors differ.
- The **datatype** of the input tensor and output tensors differs.
- The strides of the input tensor and output tensors differ and in-place operation is used (i.e., **x** and **y** pointers are equal).
- The provided **reserveSpaceSizeInBytes** is less then the value returned by **cudnnDropoutGetReserveSpaceSize**
- **cudnnSetDropoutDescriptor** has not been called on **dropoutDesc** with the non-NULL **states** argument

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.120. cudnnCreateSpatialTransformerDescriptor

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

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

Returns

CUDNN_STATUS_SUCCESS

The object was created successfully.

CUDNN_STATUS_ALLOC_FAILED

The resources could not be allocated.

4.121. cudnnDestroySpatialTransformerDescriptor

```
cudnnStatus_t cudnnDestroySpatialTransformerDescriptor(
    cudnnSpatialTransformerDescriptor_t stDesc)
```

This function destroys a previously created spatial transformer descriptor object.

Returns

CUDNN_STATUS_SUCCESS

The object was destroyed successfully.

4.122. cudnnSetSpatialTransformerNdDescriptor

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

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

Parameters

stDesc

Input/Output. Previously created spatial transformer descriptor object.

samplerType

Input. Enumerant to specify the sampler type.

dataType

Input. Data type.

nbDims

Input. Dimension of the transformed tensor.

dimA

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

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

4.123. cudnnSpatialTfGridGeneratorForward

```
cudnnStatus_t cudnnSpatialTfGridGeneratorForward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t   stDesc,
    const void                                *theta,
    void                                      *grid)
```

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

Only 2d transformation is supported.

Parameters

handle

Input. Handle to a previously created cuDNN context.

stDesc

Input. Previously created spatial transformer descriptor object.

theta

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

grid

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is NULL.
- One of the parameters **grid, theta** is NULL.

CUDNN_STATUS_NOT_SUPPORTED

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

- The dimension of transformed tensor specified in **stDesc** > 4.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.124. cudnnSpatialTfGridGeneratorBackward

```
cudnnStatus_t cudnnSpatialTfGridGeneratorBackward(
    cudnnHandle_t                                handle,
    const cudnnSpatialTransformerDescriptor_t    stDesc,
    const void                                  *dgrid,
    void                                        *dtheta)
```

This function computes the gradient of a grid generation operation.

> Only 2d transformation is supported.

Parameters

handle

Input. Handle to a previously created cuDNN context.

stDesc

Input. Previously created spatial transformer descriptor object.

dgrid

Input. Data pointer to GPU memory contains the input differential data.

dtheta

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is NULL.
- One of the parameters **dgrid, dtheta** is NULL.

CUDNN_STATUS_NOT_SUPPORTED

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

- The dimension of transformed tensor specified in **stDesc** > 4.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.125. cudnnSpatialTfSamplerForward

```
cudnnStatus_t cudnnSpatialTfSamplerForward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t  stDesc,
    const void                                *alpha,
    const cudnnTensorDescriptor_t              xDesc,
    const void                                *x,
    const void                                *grid,
    const void                                *beta,
    cudnnTensorDescriptor_t                    yDesc,
    void                                      *y)
```

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

Only 2d transformation is supported.

Parameters

handle

Input. Handle to a previously created cuDNN context.

stDesc

Input. Previously created spatial transformer descriptor object.

alpha,beta

Input. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows: dstValue = alpha[0]*srcValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

x

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

grid

Input. A grid of coordinates generated by **cudnnSpatialTfGridGeneratorForward**.

yDesc

Input. Handle to the previously initialized output tensor descriptor.

y

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

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

Returns

CUDNN_STATUS_SUCCESS

The call was successful.

CUDNN_STATUS_BAD_PARAM

At least one of the following conditions are met:

- **handle** is NULL.
- One of the parameters **x, y, grid** is NULL.

CUDNN_STATUS_NOT_SUPPORTED

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

- The dimension of transformed tensor > 4.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

4.126. cudnnSpatialTfSamplerBackward

```
cudnnStatus_t cudnnSpatialTfSamplerBackward(
    cudnnHandle_t                              handle,
    const cudnnSpatialTransformerDescriptor_t  stDesc,
    const void                                *alpha,
    const cudnnTensorDescriptor_t              xDesc,
    const void                                *x,
    const void                                *beta,
    const cudnnTensorDescriptor_t              dxDesc,
    void                                      *dx,
    const void                                *alphaDgrid,
    const cudnnTensorDescriptor_t              dyDesc,
    const void                                *dy,
    const void                                *grid,
    const void                                *betaDgrid,
    void                                      *dgrid)
```

This function computes the gradient of a sampling operation.

> Only 2d transformation is supported.

Parameters

handle

Input. Handle to a previously created cuDNN context.

stDesc

Input. Previously created spatial transformer descriptor object.

alpha,beta

Input. Pointers to scaling factors (in host memory) used to blend the source value with prior value in the destination tensor as follows: dstValue = alpha[0]*srcValue + beta[0]*priorDstValue. Please refer to this section for additional details.

xDesc

Input. Handle to the previously initialized input tensor descriptor.

x

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

dxDesc

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

dx

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

alphaDgrid,betaDgrid

Input. Pointers to scaling factors (in host memory) used to blend the gradient outputs dgrid with prior value in the destination pointer as follows: dstValue = alpha[0]*srcValue + beta[0]*priorDstValue. Please refer to this section for additional details.

dyDesc

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

dy

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

grid

Input. A grid of coordinates generated by **`cudnnSpatialTfGridGeneratorForward`**.

dgrid

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

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

Returns

`CUDNN_STATUS_SUCCESS`

The call was successful.

`CUDNN_STATUS_BAD_PARAM`

At least one of the following conditions are met:

- **`handle`** is NULL.
- One of the parameters **`x,dx,y,dy,grid,dgrid`** is NULL.
- The dimension of **`dy`** differs from those specified in **`stDesc`**

CUDNN_STATUS_NOT_SUPPORTED

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

- The dimension of transformed tensor > 4.

CUDNN_STATUS_EXECUTION_FAILED

The function failed to launch on the GPU.

Chapter 5. ACKNOWLEDGMENTS

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

5.1. University of Tennessee

5.2. University of California, Berkeley

5.3. Facebook AI Research, New York

Notice

Trademarks

Copyright

www.nvidia.com