TABLE OF CONTENTS

Chapter 1. Overview ................................................................. 1
Chapter 2. Programming Model .................................................. 2
Chapter 3. Convolution Formulas .............................................. 4
Chapter 4. Notation ................................................................. 7
Chapter 5. Tensor Descriptor .................................................... 9
  5.1. WXYZ Tensor Descriptor ................................................. 9
  5.2. 4-D Tensor Descriptor .................................................... 9
  5.3. 5-D Tensor Description .................................................. 10
  5.4. Fully-packed Tensors .................................................... 10
  5.5. Partially-packed Tensors ............................................... 10
  5.6. Spatially Packed Tensors .............................................. 11
  5.7. Overlapping Tensors .................................................... 11
Chapter 6. Data Layout Formats .............................................. 12
  6.1. Example ....................................................................... 12
  6.2. NCHW Memory Layout ............................................... 13
  6.3. NHWC Memory Layout ............................................... 14
  6.4. NC/32HW32 Memory Layout ....................................... 15
Chapter 7. Thread Safety .......................................................... 18
Chapter 8. Reproducibility (determinism) .................................. 19
Chapter 9. Scaling Parameters ............................................... 20
Chapter 10. Tensor Core Operations ......................................... 22
  10.1. Basics ........................................................................ 22
  10.2. Convolution Functions ............................................... 23
    10.2.1. Prerequisites ....................................................... 23
    10.2.2. Supported Algorithms ........................................ 23
    10.2.3. Data And Filter Formats ..................................... 23
  10.3. RNN Functions ........................................................ 24
    10.3.1. Prerequisites ....................................................... 24
    10.3.2. Supported Algorithms ........................................ 24
    10.3.3. Data And Filter Formats ..................................... 24
  10.4. Tensor Transformations .............................................. 25
    10.4.1. FP16 Data ........................................................... 25
    10.4.2. FP32-to-FP16 Conversion .................................... 25
    10.4.3. Padding ............................................................. 26
    10.4.4. Folding ............................................................. 27
    10.4.5. Conversion Between NCHW And NHWC ............. 27
  10.5. Guidelines For A Deep Learning Compiler ..................... 27
Chapter 11. GPU And Driver Requirements ............................. 28
Chapter 12. Backward Compatibility And Deprecation Policy ........ 29
Chapter 13. Grouped Convolutions ......................................... 30
Chapter 14. API Logging..........................................................................................32
Chapter 15. Features Of RNN Functions.................................................................34
Chapter 16. Mixed Precision Numerical Accuracy.................................................37
Chapter 17. Acknowledgments.............................................................................38
  17.1. University of Tennessee.............................................................................38
  17.2. University of California, Berkeley...............................................................38
  17.3. Facebook AI Research, New York.............................................................39
NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications:

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

cuDNN’s convolution routines aim for a performance that is competitive with the fastest GEMM (matrix multiply)-based implementations of such routines, while using significantly less memory.

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

cuDNN offers a context-based API that allows for easy multithreading and (optional) interoperability with CUDA streams.
Chapter 2.
PROGRAMMING MODEL

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

An application using cuDNN must initialize a handle to the library context by calling cudnnCreate(). This handle is explicitly passed to every subsequent library function that operates on GPU data. Once the application finishes using cuDNN, it can release the resources associated with the library handle using cudnnDestroy(). This approach allows the user to explicitly control the library’s functioning when using multiple host threads, GPUs and CUDA Streams.

For example, an application can use cudaSetDevice to associate different devices with different host threads, and in each of those host threads, use a unique cuDNN handle that directs the library calls to the device associated with it. Thus the cuDNN library calls made with different handles will automatically run on different devices.

The device associated with a particular cuDNN context is assumed to remain unchanged between the corresponding cudnnCreate() and cudnnDestroy() calls. In order for the cuDNN library to use a different device within the same host thread, the application must set the new device to be used by calling cudaSetDevice() and then create another cuDNN context, which will be associated with the new device, by calling cudnnCreate().

cuDNN API Compatibility

Beginning in cuDNN 7, the binary compatibility of patch and minor releases is maintained as follows:

- Any patch release x.y.z is forward or backward-compatible with applications built against another cuDNN patch release x.y.w (meaning, of the same major and minor version number, but having w!=z).
- cuDNN minor releases beginning with cuDNN 7 are binary backward-compatible with applications built against the same or earlier patch release (meaning, an app built against cuDNN 7.x is binary compatible with cuDNN library 7.y, where y>=x).
Applications compiled with a cuDNN version 7.y are not guaranteed to work with 7.x release when y > x.
Chapter 3.
CONVOLUTION FORMULAS

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

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

Table 1 Convolution terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>Input (image) Tensor</td>
</tr>
<tr>
<td>$w$</td>
<td>Weight Tensor</td>
</tr>
<tr>
<td>$y$</td>
<td>Output Tensor</td>
</tr>
<tr>
<td>$n$</td>
<td>Current Batch Size</td>
</tr>
<tr>
<td>$c$</td>
<td>Current Input Channel</td>
</tr>
<tr>
<td>$C$</td>
<td>Total Input Channels</td>
</tr>
<tr>
<td>$H$</td>
<td>Input Image Height</td>
</tr>
<tr>
<td>$W$</td>
<td>Input Image Width</td>
</tr>
<tr>
<td>$k$</td>
<td>Current Output Channel</td>
</tr>
<tr>
<td>$K$</td>
<td>Total Output Channels</td>
</tr>
<tr>
<td>$p$</td>
<td>Current Output Height Position</td>
</tr>
<tr>
<td>$q$</td>
<td>Current Output Width Position</td>
</tr>
<tr>
<td>$G$</td>
<td>Group Count</td>
</tr>
<tr>
<td>$pad$</td>
<td>Padding Value</td>
</tr>
<tr>
<td>$u$</td>
<td>Vertical Subsample Stride (along Height)</td>
</tr>
<tr>
<td>$v$</td>
<td>Horizontal Subsample Stride (along Width)</td>
</tr>
<tr>
<td>$dH_h$</td>
<td>Vertical Dilation (along Height)</td>
</tr>
<tr>
<td>$dH_w$</td>
<td>Horizontal Dilation (along Width)</td>
</tr>
</tbody>
</table>
Convolution Formulas

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>Current Filter Height</td>
</tr>
<tr>
<td>$R$</td>
<td>Total Filter Height</td>
</tr>
<tr>
<td>$s$</td>
<td>Current Filter Width</td>
</tr>
<tr>
<td>$S$</td>
<td>Total Filter Width</td>
</tr>
<tr>
<td>$C_g$</td>
<td>$\frac{C}{G}$</td>
</tr>
<tr>
<td>$K_g$</td>
<td>$\frac{K}{G}$</td>
</tr>
</tbody>
</table>

Normal Convolution (using cross-correlation mode)

$$y_{n_k, p, q} = \sum_{c} \sum_{r} \sum_{s} x_{n, c, p+r, q+s} \times w_{k, c, r, s}$$

Convolution with Padding

$$x_{<0, <0} = 0$$
$$x_{>W, >H} = 0$$

$$y_{n_k, p, q} = \sum_{c} \sum_{r} \sum_{s} x_{n, c, p+pad, q+pad} \times w_{k, c, r, s}$$

Convolution with Subsample-Striding

$$y_{n_k, p, q} = \sum_{c} \sum_{r} \sum_{s} x_{n, c, (p*v) + r, (q*v) + s} \times w_{k, c, r, s}$$

Convolution with Dilation

$$y_{n_k, p, q} = \sum_{c} \sum_{r} \sum_{s} x_{n, c, p+(r*dilh), q+(s*dilw)} \times w_{k, c, r, s}$$

Convolution using Convolution Mode

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

Convolution using Grouped Convolution

$$C_g = \frac{C}{G}$$
$$K_g = \frac{K}{G}$$
\[ y_{n,k,p,q} = \sum_c \sum_r \sum_s x_{n_c} g_r \text{floor}(k/K) + c, p+r, q+s \times w_{k,c,r,s} \]
Chapter 4.
NOTATION

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

**During inference**

\[ y = \text{layerFunction}(x, \text{otherParams}) \]

**During backpropagation**

\[ (dx, d\text{otherParams}) = \text{layerFunctionGradient}(x, y, dy, \text{otherParams}) \]

**During convolution**

For convolution, the notation is:

\[ y = x*w+b \]

where:

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

In backpropagation routines the parameters keep their meanings.

The parameters \( dx, dy, dw, db \) always refer to the gradient of the final network error function with respect to a given parameter. So \( dy \) in all backpropagation routines always refers to error gradient backpropagated through the network computation graph so far.
Similarly, other parameters in more specialized layers, such as, for instance, \texttt{dMeans} or \texttt{dBnBias} refer to gradients of the loss function with regard to those parameters.

\texttt{w} is used in the API for both the width of the \texttt{x} tensor and convolution filter matrix. To resolve this ambiguity we use \texttt{w} and \texttt{filter} notation interchangeably for convolution filter weight matrix. The meaning is clear from the context since the layer width is always referenced near its height.
Chapter 5.
TENSOR DESCRIPTOR

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

- a dimension **nbDims** from 3 to 8
- a data type (32-bit floating point, 64 bit-floating point, 16 bit floating point...)
- **dimA** integer array defining the size of each dimension
- **strideA** integer array defining the stride of each dimension (for example, the number of elements to add to reach the next element from the same dimension)

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

5.1. 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

5.2. 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

5.3. 5-D Tensor Description

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

- NCDHW
- NDHWC
- CDHWN

5.4. 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.

5.5. Partially-packed Tensors

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

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

For example, a NHWC tensor WC-packed means that the c_stride is equal to 1 and w_stride is equal to c_dim x c_stride. In practice, the -packed suffix is usually with slowest changing dimensions of a tensor but it is also possible to refer to a NCHW tensor that is only N-packed.
5.6. 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.

5.7. Overlapping Tensors

A tensor is defined to be overlapping if iterating over a full range of dimensions produces the same address more than once.
In practice an overlapped tensor will have $\text{stride}[i-1] < \text{stride}[i] \times \text{dim}[i]$ for some of the $i$ from $[1, \text{nbDims}]$ interval.
This section describes how cuDNN Tensors are arranged in memory. See cudnnTensorFormat_t for enumerated Tensor format types.

6.1. Example

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

- $N$ is the batch size; 1.
- $C$ is the number of feature maps (i.e., number of channels); 64.
- $H$ is the image height; 5.
- $W$ is the image width; 4.

To keep the example simple, the image pixel elements are expressed as a sequence of integers, 0, 1, 2, 3, and so on. See Figure 1.
6.2. NCHW Memory Layout

The above 4D Tensor is laid out in the memory in the NCHW format as below:

1. Beginning with the first channel (c=0), the elements are arranged contiguously in row-major order.
2. Continue with second and subsequent channels until the elements of all the channels are laid out. See Figure 2.
3. Proceed to the next batch (if N is > 1).
For the NHWC memory layout, the corresponding elements in all the C channels are laid out first, as below:

1. Begin with the first element of channel 0, then proceed to the first element of channel 1, and so on, until the first elements of all the C channels are laid out.
2. Next, select the second element of channel 0, then proceed to the second element of channel 1, and so on, until the second element of all the channels are laid out.
3. Follow the row-major order in channel 0 and complete all the elements. See Figure 3.
4. Proceed to the next batch (if N is > 1).
Figure 3  NHWC Memory Layout

6.4. NC/32HW32 Memory Layout

The NC/32HW32 is similar to NHWC, with a key difference. For the NC/32HW32 memory layout, the 64 channels are grouped into two groups of 32 channels each—first group consisting of channels c0 through c31, and the second group consisting of channels c32 through c63. Then each group is laid out using the NHWC format. See Figure 4.
For the generalized NC/xHWx layout format, the following observations apply:

- Only the channel dimension, C, is grouped into x channels each.
- When x = 1, each group has only one channel. Hence, the elements of one channel (i.e., one group) are arranged contiguously (in the row-major order), before proceeding to the next group (i.e., next channel). This is the same as NCHW format.
- When x = C, then NC/xHWx is identical to NHWC, i.e., the entire channel depth C is considered as a single group. The case x = C can be thought of as vectorizing entire C dimension as one big vector, laying out all the Cs, followed by the remaining dimensions, just like NHWC.
- The tensor format `CUDNN_TENSOR_NCHW_VECT_C` can also be interpreted in the following way: The NCHW INT8x32 format is really N x (C/32) x H x W x 32 (32...
Cs for every W), just as the NCHW INT8x4 format is N x (C/4) x H x W x 4 (4 Cs for every W). Hence the "VECT_C" name - each W is a vector (4 or 32) of Cs.
Chapter 7.
THREAD SAFETY

The library is thread safe and its functions can be called from multiple host threads, as long as threads do not share the same cuDNN handle simultaneously.
Chapter 8.
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`
Many cuDNN routines like cudnnConvolutionForward() accept pointers in host memory to scaling factors alpha and beta. These scaling factors are used to blend the computed values with the prior values in the destination tensor as follows (see Figure 5):

\[ \text{dstValue} = \alpha \times \text{computedValue} + \beta \times \text{priorDstValue} \]

The dstValue is written to after being read.

Figure 5  Scaling Parameters for Convolution

When beta is zero, the output is not read and may contain uninitialized data (including NaN).

These parameters are passed using a host memory pointer. The storage data types for alpha and beta are:
Scaling Parameters

- **float** for HALF and FLOAT tensors, and
- **double** for DOUBLE tensors.

For improved performance use **beta = 0.0**. Use a non-zero value for beta only when you need to blend the current output tensor values with the prior values of the output tensor.

**Type Conversion**

When the data input \( x \), the filter input \( w \) and the output \( y \) are all in INT8 data type, the function `cudnnConvolutionBiasActivationForward()` will perform the type conversion as shown in **Figure 6**:

Accumulators are 32-bit integers which wrap on overflow.

**Figure 6** INT8 for `cudnnConvolutionBiasActivationForward`
Chapter 10. TENSOR CORE OPERATIONS

The cuDNN v7 library introduced the acceleration of compute-intensive routines using Tensor Core hardware on supported GPU SM versions. Tensor core operations are supported on the Volta and Turing GPU families.

10.1. Basics

Tensor core operations perform parallel floating point accumulation of multiple floating point product terms. Setting the math mode to CUDNN_TENSOR_OP_MATH via the cudnnMathType_t enumerator indicates that the library will use Tensor Core operations. This enumerator specifies the available options to enable the Tensor Core, and should be applied on a per-routine basis.

The default math mode is CUDNN_DEFAULT_MATH, which indicates that the Tensor Core operations will be avoided by the library. Because the CUDNN_TENSOR_OP_MATH mode uses the Tensor Cores, it is possible that these two modes generate slightly different numerical results due to different sequencing of the floating point operations.

For example, the result of multiplying two matrices using Tensor Core operations is very close to, but not always identical, the result achieved using a sequence of scalar floating point operations. For this reason, the cuDNN library requires an explicit user opt-in before enabling the use of Tensor Core operations.

However, experiments with training common deep learning models show negligible differences between using Tensor Core operations and scalar floating point paths, as measured by both the final network accuracy and the iteration count to convergence. Consequently, the cuDNN library treats both modes of operation as functionally indistinguishable, and allows for the scalar paths to serve as legitimate fallbacks for cases in which the use of Tensor Core operations is unsuitable.

Kernels using Tensor Core operations are available for both convolutions and RNNs. See also Training with Mixed Precision.
10.2. Convolution Functions

10.2.1. Prerequisites

For the supported GPUs, the Tensor Core operations will be triggered for convolution functions only when `cudnnSetConvolutionMathType()` is called on the appropriate convolution descriptor by setting the `mathType` to `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`.

10.2.2. Supported Algorithms

When the prerequisite is met, the below convolution functions can be run as Tensor Core operations:

- `cudnnConvolutionForward()`
- `cudnnConvolutionBackwardData()`
- `cudnnConvolutionBackwardFilter()`

See the table below for supported algorithms:

<table>
<thead>
<tr>
<th>Supported Convolution Function</th>
<th>Supported Algos</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cudnnConvolutionForward</code></td>
<td>- CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM</td>
</tr>
<tr>
<td></td>
<td>- CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED</td>
</tr>
<tr>
<td><code>cudnnConvolutionBackwardData</code></td>
<td>- CUDNN_CONVOLUTION_BWD_DATA_ALGO_1</td>
</tr>
<tr>
<td></td>
<td>- CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD_NONFUSED</td>
</tr>
<tr>
<td><code>cudnnConvolutionBackwardFilter</code></td>
<td>- CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1</td>
</tr>
<tr>
<td></td>
<td>- CUDNN_CONVOLUTION_BWD_FILTER_ALGO_WINOGRAD_NONFUSED</td>
</tr>
</tbody>
</table>

10.2.3. Data And Filter Formats

The cuDNN library may use padding, folding, and NCHW-to-NHWC transformations to call the Tensor Core operations. See Tensor Transformations.

For algorithms other than `*_ALGO_WINOGRAD_NONFUSED`, when the following requirements are met, the cuDNN library will trigger the Tensor Core operations:

- Input, filter, and output descriptors (`xDesc, yDesc, wDesc, dxDesc, dyDesc` and `dwDesc` as applicable) are of the `dataType = CUDNN_DATA_HALF` (i.e., FP16). For FP32 `dataType` see FP32-to-FP16 Conversion.
- The number of input and output feature maps (i.e., channel dimension `C`) is a multiple of 8. When the channel dimension is not a multiple of 8, see Padding.
- The filter is of type `CUDNN_TENSOR_NCHW` or `CUDNN_TENSOR_NHWC`.
- If using a filter of type `CUDNN_TENSOR_NHWC`, then the input, filter, and output data pointers (`X, Y, W, dX, dY,` and `dW` as applicable) are aligned to 128-bit boundaries.
10.3. RNN Functions

10.3.1. Prerequisites

Tensor core operations will be triggered for these RNN functions only when `cudnnSetRNNMatrixMathType()` is called on the appropriate RNN descriptor setting `mathType` to `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`.

10.3.2. Supported Algorithms

When the above prerequisite is met, the RNN functions below can be run as Tensor Core operations:

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

See the table below for the supported algorithms:

<table>
<thead>
<tr>
<th>RNN Function</th>
<th>Support Algos</th>
</tr>
</thead>
</table>
| All RNN functions that support Tensor Core operations. | `CUDNN_RNN_ALGO_STANDARD`  
|                                                   | `CUDNN_RNN_ALGO_PERSIST_STATIC`                   |

10.3.3. Data And Filter Formats

When the following requirements are met, then the cuDNN library will trigger the Tensor Core operations:

- For `algo = CUDNN_RNN_ALGO_STANDARD`:
  - The hidden state size, input size and the batch size is a multiple of 8.
  - All user-provided tensors, workspace, and reserve space are aligned to 128 bit boundaries.
  - For FP16 input/output, the `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
  - For FP32 input/output, `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
- For `algo = CUDNN_RNN_ALGO_PERSIST_STATIC`:
  - The hidden state size and the input size is a multiple of 32.
The batch size is a multiple of 8.
- If the batch size exceeds 96 (for forward training or inference) or 32 (for backward data), then the batch sizes constraints may be stricter, and large power-of-two batch sizes may be needed.
- All user-provided tensors, workspace, and reserve space are aligned to 128 bit boundaries.
- For FP16 input/output, `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.
- For FP32 input/output, `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is selected.

See also Features Of RNN Functions.

### 10.4. Tensor Transformations

A few functions in the cuDNN library will perform transformations such as folding, padding, and NCHW-to-NHWC conversion while performing the actual function operation. See below.

#### 10.4.1. FP16 Data

Tensor Cores operate on FP16 input data with FP32 accumulation. The FP16 multiply leads to a full-precision result that is accumulated in FP32 operations with the other products in a given dot product for a matrix with $m \times n \times k$ dimensions. See Figure 7.

**Figure 7 Tensor Operation with FP16 Inputs**

#### 10.4.2. FP32-to-FP16 Conversion

The cuDNN API for allows the user to specify that FP32 input data may be copied and converted to FP16 data internally to use Tensor Core operations.
for potentially improved performance. This can be achieved by selecting `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum for `cudnnMathType_t`. In this mode, the FP32 tensors are internally down-converted to FP16, the Tensor Op math is performed, and finally up-converted to FP32 as outputs. See Figure 8.

For convolutions, the FP32-to-FP16 conversion can be achieved by passing the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum value to the `cudnnSetConvolutionMathType()` call.

```c
// Set the math type to allow cuDNN to use Tensor Cores:
checkCudnnErr(cudnnSetConvolutionMathType(cudnnConvDesc,
    CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION));
```

For RNNs, the FP32-to-FP16 conversion can be achieved by passing the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` enum value to the `cudnnSetRNNMatrixMathType()` call to allow FP32 data to be converted for use in RNNs.

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

### 10.4.3. Padding

For packed NCHW data, when the channel dimension is not a multiple of 8, then the cuDNN library will pad the tensors as needed to enable Tensor Core operations. This padding is automatic for packed NCHW data in both the `CUDNN_TENSOR_OP_MATH` and `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` cases.
The padding occurs with a negligible loss of performance. Hence, the NCHW tensor dimensions such as below are allowed:

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

### 10.4.4. Folding

In the folding operation the cuDNN library implicitly performs the formatting of input tensors and saves the input tensors in an internal workspace. This can lead to an acceleration of the call to Tensor Cores.

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

### 10.4.5. Conversion Between NCHW And NHWC

Tensor Cores require that the tensors be in NHWC data layout. Conversion between NCHW and NHWC is performed when the user requests Tensor Op math. However, as stated in Basics, a request to use Tensor Cores is just that, a request, and Tensor Cores may not be used in some cases. The cuDNN library converts between NCHW and NHWC if and only if Tensor Cores are requested and are actually used.

If your input (and output) are NCHW, then expect a layout change.

Non-Tensor Op convolutions will not perform conversions between NCHW and NHWC.

In very rare and difficult-to-qualify cases that are a complex function of padding and filter sizes, it is possible that Tensor Ops are not enabled. In such cases, users should pre-pad.

### 10.5. Guidelines For A Deep Learning Compiler

For a deep learning compiler, the following are the key guidelines:

- Make sure that the convolution operation is eligible for Tensor Cores by avoiding any combinations of large padding and large filters.
- Transform the inputs and filters to NHWC, pre-pad channel and batch size to be a multiple of 8.
- Make sure that all user-provided tensors, workspace and reserve space are aligned to 128 bit boundaries.
Chapter 11.
GPU AND DRIVER REQUIREMENTS

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix.
Chapter 12.
BACKWARD COMPATIBILITY AND DEPRECATION POLICY

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

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

As a rule of thumb, when a routine appears in two forms, one with a suffix and one with no suffix, the non-suffixed entry is to be treated as deprecated. In this case, it is strongly advised that users migrate to the new suffixed API entry to guarantee backwards compatibility in the following cuDNN release. When a routine appears with multiple suffixes, the unsuffixed API entry is mapped to the higher numbered suffix. In that case, it is strongly advised to use the non-suffixed API entry to guarantee backward compatibility with the following cuDNN release.
cuDNN supports grouped convolutions by setting `groupCount` > 1 for the convolution descriptor `convDesc`, using `cudnnSetConvolutionGroupCount()`.

By default the convolution descriptor `convDesc` is set to `groupCount` of 1.

**Basic Idea**

Conceptually, in grouped convolutions the input channels and the filter channels are split into `groupCount` number of independent groups, with each group having a reduced number of channels. Convolution operation is then performed separately on these input and filter groups.

For example, consider the following: if the number of input channels is 4, and the number of filter channels of 12. For a normal, ungrouped convolution, the number of computation operations performed are 12*4.

If the `groupCount` is set to 2, then there are now two input channel groups of two input channels each, and two filter channel groups of six filter channels each.

As a result, each grouped convolution will now perform 2*6 computation operations, and two such grouped convolutions are performed. Hence the computation savings are 2x: (12*4)/(2*(2*6))

**cuDNN Grouped Convolution**

- When using `groupCount` for grouped convolutions, you must still define all tensor descriptors so that they describe the size of the entire convolution, instead of specifying the sizes per group.
- Grouped convolutions are supported for all formats that are currently supported by the functions `cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()` and `cudnnConvolutionBackwardFilter()`. 
- The tensor stridings that are set for `groupCount` of 1 are also valid for any group count.
- By default the convolution descriptor `convDesc` is set to `groupCount` of 1.

See Convolution Formulas for the math behind the cuDNN Grouped Convolution.

**Example**

Below is an example showing the dimensions and strides for grouped convolutions for NCHW format, for 2D convolution.

The symbols "*" and "/" are used to indicate multiplication and division.

**xDesc or dxDesc:**
- Dimensions: `[batch_size, input_channel, x_height, x_width]`
- Strides: `[input_channels*x_height*x_width, x_height*x_width, x_width, 1]`

**wDesc or dwDesc:**
- Dimensions: `[output_channels, input_channels/groupCount, w_height, w_width]`
- Format: NCHW

**convDesc:**
- Group Count: `groupCount`

**yDesc or dyDesc:**
- Dimensions: `[batch_size, output_channels, y_height, y_width]`
- Strides: `[output_channels*y_height*y_width, y_height*y_width, y_width, 1]`
cuDNN API logging is a tool that records all input parameters passed into every cuDNN API function call. This functionality is disabled by default, and can be enabled through methods described in this section.

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

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

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

There are two methods to enable API logging.

**Method 1: Using Environment Variables**

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

- Set the environment variable `CUDNN_LOGINFO_DBG` to “1”, and
- Set the environment variable `CUDNN_LOGDEST_DBG` to one of the following:
  - `stdout`, `stderr`, or a user-desired file path, for example, `/home/userName1/log.txt`.
  - Include the conversion specifiers in the file name. For example:
    - To include date and time in the file name, use the date and time conversion specifiers: `log_%Y_%m_%d_%H_%M_%S.txt`. The conversion specifiers will be automatically replaced with the date and time when the program is initiated, resulting in `log_2017_11_21_09_41_00.txt`.  

To include the process id in the file name, use the \%i conversion specifier: \texttt{log\_%Y\_%m\_%d\_%H\_%M\_%S\_%i.txt} for the result: \texttt{log\_2017\_11\_21\_09\_41\_00\_21264.txt} when the process id is \texttt{21264}. When you have several processes running, using the process id conversion specifier will prevent these processes writing to the same file at the same time.

The supported conversion specifiers are similar to the \texttt{strftime} function.

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

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

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

Table 2  API Logging Using Environment Variables

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>CUDNN_LOGINFO_DBG=0</th>
<th>CUDNN_LOGINFO_DBG=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDNN_LOGDEST_DBG not set</td>
<td>No logging output</td>
<td>No logging output</td>
</tr>
<tr>
<td></td>
<td>No performance loss</td>
<td>No performance loss</td>
</tr>
<tr>
<td>CUDNN_LOGDEST_DBG = NULL</td>
<td>No logging output</td>
<td>No logging output</td>
</tr>
<tr>
<td></td>
<td>No performance loss</td>
<td>No performance loss</td>
</tr>
<tr>
<td>CUDNN_LOGDEST_DBG=stdout or stderr</td>
<td>No logging output</td>
<td>Logging to stdout or stderr</td>
</tr>
<tr>
<td></td>
<td>No performance loss</td>
<td>Some performance loss</td>
</tr>
<tr>
<td>CUDNN_LOGDEST_DBG=filename.txt</td>
<td>No logging output</td>
<td>Logging to filename.txt</td>
</tr>
<tr>
<td></td>
<td>No performance loss</td>
<td>Some performance loss</td>
</tr>
</tbody>
</table>

Method 2

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

www.nvidia.com

cuDNN Developer's Guide
Chapter 15.
FEATURES OF RNN FUNCTIONS

The RNN functions are:

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

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

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

<table>
<thead>
<tr>
<th>Functions</th>
<th>Input output layout supported</th>
<th>Supports variable sequence length in batch</th>
<th>Commonly supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudnnRNNForwardInference()</td>
<td>Only, Sequence major, packed (non-padded)</td>
<td>Only with _ALGO_STANDARD</td>
<td>Mode (cell type) supported: CUDNN_RNN_RELU, CUDNN_RNN_TANH, CUDNN_LSTM, CUDNN_GRU</td>
</tr>
<tr>
<td>cudnnRNNForwardTraining</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudnnRNNBackwardData</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudnnRNNBackwardWeights</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudnnRNNForwardInferenceEx</td>
<td></td>
<td></td>
<td>Algo supported1 (see the table below for an elaboration on these algorithms):</td>
</tr>
<tr>
<td>cudnnRNNForwardTrainingEx</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 Do not mix different algos for different steps of training. It’s also not recommended to mix non-extended and extended API for different steps of training.
### Features Of RNN Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Input output layout supported</th>
<th>Supports variable sequence length in batch</th>
<th>Commonly supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudnnRNNBackwardDataEx</td>
<td>Batch major unpacked(^2)</td>
<td>For unpacked layout(^2), no input sorting required.</td>
<td>_ALGO_STANDARD, _ALGO_PERSIST_STATIC, _ALGO_PERSIST_DYNAMIC</td>
</tr>
<tr>
<td>cudnnRNNBackwardWeightsEx</td>
<td>Sequence major packed(^2)</td>
<td>For packed layout, require input sequences descending sorted according to length</td>
<td>Math mode supported: CUDNN_DEFAULT_MATH, CUDNN_TENSOR_OP_MATH (will automatically fall back if run on pre-Volta or if algo doesn’t support Tensor Cores) _ALLOW_CONVERSION (may do down conversion to utilize Tensor Cores) Direction mode supported: CUDNN_UNIDIRECTIONAL, CUDNN_BIDIRECTIONAL RNN input mode: CUDNN_LINEAR_INPUT, CUDNN_SKIP_INPUT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Features</th>
<th>_ALGO_STANDARD</th>
<th>_ALGO_PERSIST_STATIC</th>
<th>_ALGO_PERSIST_DYNAMIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half input</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single accumulation</td>
<td>Half intermediate storage</td>
<td>Single accumulation</td>
<td></td>
</tr>
<tr>
<td>Half output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single input</td>
<td>Supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single accumulation</td>
<td>If running on Volta, with CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION(^1), will down-convert and use half intermediate storage. Otherwise: Single intermediate storage</td>
<td>Single accumulation</td>
<td></td>
</tr>
<tr>
<td>Single output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double input</td>
<td>Supported</td>
<td></td>
<td>Not Supported</td>
</tr>
<tr>
<td>Double accumulation</td>
<td>Double intermediate storage</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Double output</td>
<td>Double accumulation</td>
<td></td>
<td>Double intermediate storage</td>
</tr>
</tbody>
</table>

\(^2\) To use unpacked layout, user need to set CUDNN_RNN_PADDED_IO_ENABLED through cudnnSetRNNPaddingMode().

---

www.nvidia.com

cuDNN Developer’s Guide

DU-06702-001_v7.6.5 | 35
## Features Of RNN Functions

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

---

3 CUDNN_TENSOR_OP_MATH or CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION can be set through cudnnSetRNNMatrixMathType().
Chapter 16.
MIXED PRECISION NUMERICAL ACCURACY

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

For example, when the computation is performed in FP32 and the output is in FP16, the CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0 (ALGO_0) has lower accuracy compared to the CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1 (ALGO_1). This is because ALGO_0 does not use extra workspace, and is forced to accumulate the intermediate results in FP16, i.e., half precision float, and this reduces the accuracy. The ALGO_1, on the other hand, uses additional workspace to accumulate the intermediate values in FP32, i.e., full precision float.
Chapter 17. 
ACKNOWLEDGMENTS

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

17.1. University of Tennessee

Copyright (c) 2010 The University of Tennessee.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
* Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

17.2. University of California, Berkeley

COPYRIGHT

All contributions by the University of California:

Copyright (c) 2014, The Regents of the University of California (Regents)
Acknowledgments

All rights reserved.

All other contributions:
Copyright (c) 2014, the respective contributors
All rights reserved.

Caffe uses a shared copyright model: each contributor holds copyright over their contributions to Caffe. The project versioning records all such contribution and copyright details. If a contributor wants to further mark their specific copyright on a particular contribution, they should indicate their copyright solely in the commit message of the change when it is committed.

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CONTRIBUTION AGREEMENT

By contributing to the BVLC/caffe repository through pull-request, comment, or otherwise, the contributor releases their content to the license and copyright terms herein.

17.3. Facebook AI Research, New York

Copyright (c) 2014, Facebook, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name Facebook nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
<table>
<thead>
<tr>
<th>Acknowledgments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional Grant of Patent Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Software&quot; means fbctnn software distributed by Facebook, Inc.</td>
</tr>
</tbody>
</table>

Facebook hereby grants you a perpetual, worldwide, royalty-free, non-exclusive, irrevocable (subject to the termination provision below) license under any rights in any patent claims owned by Facebook, to make, have made, use, sell, offer to sell, import, and otherwise transfer the Software. For avoidance of doubt, no license is granted under Facebook’s rights in any patent claims that are infringed by (i) modifications to the Software made by you or a third party, or (ii) the Software in combination with any software or other technology provided by you or a third party.

The license granted hereunder will terminate, automatically and without notice, for anyone that makes any claim (including by filing any lawsuit, assertion or other action) alleging (a) direct, indirect, or contributory infringement or inducement to infringe any patent: (i) by Facebook or any of its subsidiaries or affiliates, whether or not such claim is related to the Software, (ii) by any party if such claim arises in whole or in part from any software, product or service of Facebook or any of its subsidiaries or affiliates, whether or not such claim is related to the Software, or (iii) by any party relating to the Software; or (b) that any right in any patent claim of Facebook is invalid or unenforceable.
Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, DALI, DIGITS, DGX, DGX-1, DGX-2, DGX Station, DLProf, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NvCaffe, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, TensorRT Inference Server, Tesla, TF-TRT, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2019 NVIDIA Corporation. All rights reserved.

www.nvidia.com