NVIDIA CUDNN V2

March 2015

Release Notes

CUDNN V2 RELEASE NOTES

WHAT'S NEW IN CUDNN V2

- Performance of many routines – especially convolutions – has been improved considerably.
- Forward convolution is now implemented via several different algorithms, and the interface allows the application to choose one of these algorithms specifically or to specify a strategy (e.g., prefer fastest, use no additional working space) by which the library should select the best algorithm automatically. The four algorithms currently given are as follows:
    - **IMPLICIT_GEMM** corresponds to the sole algorithm that was provided in cuDNN Release 1; it is the only algorithm that supports all input sizes while using no additional working space.
    - **IMPLICIT_PRECOMP_GEMM** is a modification of this approach that uses a small amount of working space (specifically, **`C*R*S*sizeof(int)`** bytes, where **`C`** is the number of input feature maps and **`R`** and **`S`** are the filter height and width, respectively, for the case of 2D convolutions) to achieve significantly higher performance in most cases. This algorithm achieves its highest performance when zero-padding is not used.
    - **GEMM** is an "im2col"-based approach, explicitly expanding the input data in memory and then using an otherwise-pure matrix multiplication that obeys cuDNN's input and output stridings, avoiding a separate transposition step on the input or output. Note that this algorithm requires *significant* working space, though it may be faster than either of the two "implicit GEMM" approaches in some cases. As such, the **PREFER_FASTEST** forward convolution algorithm preference may sometimes recommend this

  approach. When memory should be used more sparingly, **SPECIFY_WORKSPACE_LIMIT** can be used instead of **PREFER_FASTEST** to ensure that the algorithm recommended will not require more than a given amount of working space.
  - **DIRECT** is a placeholder for a future implementation of direct convolution.
- The interface of cuDNN has been generalized so that data sets with other than two spatial dimensions (e.g., 1D or 3D data) can be supported in future releases.
  - Note: while the interface now allows arbitrary N-dimensional tensors, most routines in this release remain limited to two spatial dimensions. This may be relaxed in future releases based on community feedback.
  - As a BETA preview in this release, the convolution forward, convolution weight and data gradient, and cudnnSetTensor/cudnnScaleTensor routines now support 3D datasets through the “Nd” interface. Note, however, that these 3D routines have not yet been tuned for optimal performance. This will be improved in future releases.
- Many routines now allow alpha and beta scaling parameters of the following form:

$$C := \alpha * OP(...) + \beta * C$$

  - This replaces the cudnnAccumulateResult_t enumerated type in cuDNN Release 1, which allowed only two combinations, namely (alpha, beta) = (1.0, 0.0) and (alpha, beta) = (1.0, 1.0). Note that the alpha and beta arguments to these routines should be specified as pointers to host system memory.
- The pooling routines now allow zero-padding of the borders in a manner similar to what was already supported for convolutions.
- OS X is now supported.
- Support for arbitrary strides is improved. While performance is still generally best tuned for cases where the data is in NCHW order and tightly packed (i.e., the stride of especially innermost dimension is 1), arbitrary tensor orderings and/or stridings are supported and performance of these has been improved in some routines. Further improvements may be made in future releases based on community feedback.
  - cudnnSetTensor4dDescriptor now supports **CUDNN_TENSOR_NHWC**.
  - The performance of the cudnnSoftmax*() routines when using **CUDNN_SOFTMAX_MODE_CHANNEL** with tensor layouts other than ***C has been improved considerably.
  - Note: While tensors allow arbitrary data layout, filters are still assumed to be stored in KCRS order and tightly packed.

Note: Several of the aforementioned improvements required changes to the interface of cuDNN in version 2 as compared to cuDNN Release 1. Hence applications previously using cuDNN R1 are likely to require minor modifications for API compatibility with cuDNN v2 – this is not a drop-in version upgrade.

ISSUES RESOLVED

- When NULL pointers were passed to the convolution routines, **CUDNN_STATUS_MAPPING_ERROR** (which is normally indicative of an internal error) could be returned in some circumstances. These routines now check for NULL arguments and will return **CUDNN_STATUS_BAD_PARAM** if one is found.
- The static library build for Windows has been removed due to linking issues across versions of Visual Studio.
- Some routines could fail unexpectedly if certain dimensions (typically N or C) were very large. All routines now either handle these large dimensions or return **CUDNN_STATUS_NOT_SUPPORTED**.
- Activation functions allow in-place operation (i.e., the input and output pointers and the tensors structures describing them are identical). This is now documented, and a check that input and output strides are equal if the input and output pointers are equal has been added.
- The behavior of **CUDNN_POOLING _MAX** with cudnnPoolingBackward() has been improved: due to the inherent non-differentiability of the max() function, the approximation made by cuDNN Release 1 was to propagate a given srcDiff value to *all* destDiff locations for which destData == srcData, though in the case of ties, could result in an amplification of the overall gradient. In cuDNN v2, only the first encountered instance of the maximum in a given pooling window will receive the propagated gradient.
- The behavior of **CUDNN_POOLING_AVERAGE** together with zero-padding was ambiguous in previous versions: in cuDNN Release 1, any zero-padding (to the extent allowed by the interface) was *ignored* in the averages produced along the boundaries; in cuDNN v2 RC1, zero-padding was *included* in these averages. As either behavior could be argued correct, and as there has been demand for each option, both modes are now provided: **CUDNN_POOLING_AVERAGE_COUNT_(INCLUDE|EXCLUDE)_PADDING**.

- Potential out-of-bounds memory accesses in some routines have been fixed. For example, in cudnnConvolutionBackwardData(), if the number of data elements in diffData was greater than or equal to (2^{27} – 512), then in previous versions an out-of-bounds memory access could sometimes result.
- cudnnGetPooling*ForwardOutputDim did not properly account for padding when calculating output size. This has been fixed.
- In the convolution routines, if the filter size exceeds the input size in any spatial dimension, then the error **CUDNN_STATUS_NOT_SUPPORTED** is now returned.

KNOWN ISSUES

- If some dimension of a tensor is 1, checks for unsupported (e.g., zero) strides could be relaxed.
- For non-trivial routines, the documentation should explicitly describe the function implemented. In a future release, pseudocode and/or mathematical descriptions of these functions will be provided for improved clarity.
- Several cudnn*Backward() routines assume that the corresponding cudnn*Forward() routine will have been called previously without documenting this assumption.
- When using the **CUDNN_CONVOLUTION_FWD_ALGO_GEMM** algorithm for forward convolution with Maxwell-generation GPUs (i.e., those of compute capability 5.x), best performance may be achieved with NVIDIA Driver version 346.xx or later.
- cuDNN v2 is built against CUDA 6.5. Other components of an application (libraries, custom kernels, or plug-ins, for example) can be built against older or newer versions of the CUDA Toolkit without issue as long as the installed version of the NVIDIA Driver is sufficient for all components. For example, an application built with CUDA 7.0 can still use cuDNN v2. As a special exception to this, on OS X, the CUDA compiler switched from libstdc++ in CUDA 6.5 to libc++ in CUDA 7.0; hence, on OS X, cuDNN v2 is incompatible with applications built against CUDA 7.0.
- cuDNN v2 will support the forthcoming Tegra X1 processor via *PTX JIT* compilation. This will incur a roughly 30-second startup delay upon first use of cuDNN on a Tegra X1 system. Note that the JIT-compiled binaries will be cached, so this should be a one-time expense.
- In cuDNN v2, the Im2Col function is exposed as a public function; however, it is intended for internal use only. As such, it will likely be removed from the public interface in the next version.