NVIDIA cuDNN

Release Notes | NVIDIA Docs

Table of Contents

Chapter 1. cuDNN Release 8.x.x

1.1. cuDNN Release 8.4.1

These are the NVIDIA cuDNN 8.4.1 Release Notes. These Release Notes include fixes from the previous cuDNN releases as well as the following additional changes.

These Release Notes are applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previously released cuDNN documentation, refer to the *NVIDIA cuDNN Archived Documentation*.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Improved runtime subgraph compilation support

  ▶ Added support for `CUDNN_BACKEND_OPERATION_RESAMPLE_FWD` for the `CUDNN_ATTR_RESAMPLE_MODE` set to `CUDNN_RESAMPLE_AVGPOOL` and `CUDNN_RESAMPLE_MAXPOOL` through the runtime fusion engine. It can achieve up to 3x speed up compared to the legacy `cudnnPoolingForward()` API. Pointwise fusions to the output of this operation are also supported. Documentation about the patterns supported can be found in the *Supported Graph Patterns* section in the *cuDNN Developer Guide*.

  ▶ Newly added micro tile sizes for pointwise fusions that provide significantly improved performance on smaller problem sizes.

▶ The *cuDNN Developer Guide* now includes an expanded section on supported patterns of the Graph API. It takes a systematic approach to explain which graph patterns are supported, along with various graphical examples, and details on some of the restrictions.

Fixed Issues

▶ A buffer was shared between threads and caused segmentation faults. There was previously no way to have a per-thread buffer to avoid these segmentation faults. The buffer has been moved to the cuDNN handle. Ensure you have a cuDNN handle for each

thread because the buffer in the cuDNN handle is only for the use of one thread and cannot be shared between two threads.

▶ Fixed operation graph logging under `cudnnBackendExecuteGraphVisualize()` section upon calling `cudnnBackendExecute()` on generic fusion patterns. Added logging for `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR` and `CUDNN_BACKEND_MATMUL_DESCRIPTOR`. Fixed logging for pointwise mode to show the enum value name.

▶ Users specifying backend engines 58, 1063, 2062, and 4039 using `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` with 1x1 convolutions and tensors with more than two GB elements (2G) would see `CUDNN_STATUS_EXECUTION_FAILED` in cuDNN 8.3.x. This issue has been fixed in this release.

▶ cuDNN returned `CUDNN_STATUS_EXECUTION_FAILED` from `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, or `cudnnConvolutionBackwardData()` when computing convolutions with large spatial dimensions and batch sizes. This issue has been fixed. Such problems instead return `CUDNN_STATUS_NOT_SUPPORTED` where applicable.

Known Issues

▶ A compiler bug in NVRTC in CUDA version 11.7 and earlier, was causing incorrect outputs when computing logical operations on boolean input tensors in the runtime fusion engine. A workaround has been integrated in this release to avoid the most common issues. However, it is highly recommended to update to at least CUDA version 11.7u1 for a fix. Specifically, known failure cases are when pointwise operations of mode `CUDNN_POINTWISE_LOGICAL_NOT`, `CUDNN_POINTWISE_LOGICAL_AND` or `CUDNN_POINTWISE_LOGICAL_OR` operates on boolean tensors.

▶ cuDNN is not enforcing the `CUDNN_ATTR_EXECUTION_PLAN_HANDLE` attribute for the `CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR`. This issue will be fixed in a future release.

▶ If cuDNN 8.4.1 or earlier statically links with `libcudart.so` from the CUDA Toolkit 11.7 or later, when the LFL feature is activated, the results from `cudnnFind*Algo` will not be accurate.

▶ For packed NCHW tensors using the FP16 datatype, cuDNN attempts to run an optimized kernel if the values of N, C, H, and W are even. In cuDNN versions before 8.4, it is possible that incorrect values are generated if odd values for the strides of N or C are used. This issue will be resolved in a future release.

▶ Users of cuDNN's `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING` may see `CUDNN_STATUS_BAD_PARAM` returned for a problem that should otherwise be supported by that choice of algo.

▶ `cudnnDropoutForward()` and `cudnnDropoutBackward()` will return incorrect results when input or output tensors have overlapping strides.

▶ The documentation for [cudnnReorderFilterAndBias()](#) requires corrections for clarity.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

▶ FFT and Winograd based algorithms for convolution do not support graph capture.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

▶ CUDA streams internal to cuDNN are not guaranteed to have the same priority as the user stream that is set by `cudnnSetStream()`. We recently discovered some issues that break our ability to document exceptions to this clearly.

▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

▶ cudnnPoolingBackward() enables both `x` and `y` data pointers (together with the related tensor descriptor handles) to be `NULL` for avg-pooling. This could save memory footprint and bandwidth.

▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking with the `cnn_infer` and `cnn_train` static sub libraries. This will come at a cost to the binary size of the application. This linkage requirement will be relaxed in a future release.

▶ Users of cuDNN 8.4.0 may observe a slowdown in the Single Shot Multibox Detector (SSD) model. This will be fixed in a future release.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the [cuDNN Support Matrix](#).

Limitations

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX = 1025` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`) does not support tensors in which the product N*C*H*W of the output gradient tensor equals to or exceeds 2^31.

▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX =1001` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`) does not support tensors in which the product N*H*W of the output gradient tensor equals to or exceeds 2^31. This issue has been present in all previous releases of cuDNN and exercising the use case for the engine would show incorrect results.

▶ Versions of cuDNN before the 8.0 release series do not support the NVIDIA Ampere Architecture and will generate incorrect results if used on that architecture. Furthermore, if used, training operations can succeed with a NaN loss for every epoch.

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later. It also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples must be installed in a writable location. If not installed in a writable location, the samples can crash.

▶ RNN and multihead attention API calls may exhibit nondeterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the [Results Reproducibility](#) section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This happens when two buffer sizes (16 KB and 4 MB) are used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the nondeterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer

sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

► The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

► Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

► For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

► For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

► In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

► In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

► In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

► cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

► When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

► Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

  ▶ cudnnBatchNormalizationBackward()
  ▶ cudnnBatchNormalizationBackwardEx()
  ▶ cudnnBatchNormalizationForwardTraining()
  ▶ cudnnBatchNormalizationForwardTrainingEx()
  ▶ cudnnGetBatchNormalizationBackwardExWorkspaceSize()
  ▶ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()
  ▶ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()
  ▶ cudnnGetNormalizationBackwardWorkspaceSize()
  ▶ cudnnGetNormalizationForwardTrainingWorkspaceSize()
  ▶ cudnnGetNormalizationTrainingReserveSpaceSize()
  ▶ cudnnNormalizationBackward()
  ▶ cudnnNormalizationForwardTraining()

▶ `cudnnAddTensor()` performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

▶ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

Deprecated Features

The following features are deprecated in cuDNN 8.3.3:

▶ We are deprecating the reporting of performance results in the *Best Practices For Using cuDNN 3D Convolutions* and will instead update these *Release Notes* if there is anything interesting to report release-over-release. Starting with cuDNN 8.4.0, this section will be removed. For past performance tables, refer to the *cuDNN Archives* > *Best Practices For Using cuDNN 3D Convolutions*.

▶ Updated and migrated the content from the *Best Practices For Using cuDNN 3D Convolutions* to the cuDNN Developer Guide. The Best Practices document has been deprecated.

1.2.   cuDNN Release 8.4.0

These are the NVIDIA cuDNN 8.4.0 Release Notes. These Release Notes include fixes from the previous cuDNN releases as well as the following additional changes.

These Release Notes are applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previously released cuDNN documentation, refer to the *NVIDIA cuDNN Archived Documentation*.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ **API additions**

  ▶ Added API and support for the GEN_INDEX capability. CUDNN_POINTWISE_GEN_INDEX returns the position of an element in an input tensor along a given axis. This operation is similar to NumPy's mesh grid operation as it returns a tensor with the index of all elements calculated according to the specified axis in the original tensor dimensions.

  ▶ Added API and support for the BINARY_SELECT capability. CUDNN_POINTWISE_BINARY_SELECT is similar to the ternary operation and selects between two input elements based on a predicate element.

  ▶ Experimentally supports serialization of execution plans to or from a string representation to enable the user to avoid recompilation of the fusion kernels. This feature only supports the runtime fusion engine currently. Generalized support for additional engines is planned for future releases.

▶ **Runtime fusion engine improvements**

  ▶ Previous versions of the runtime fusion engine only supported a minimum 128-bit alignment for tensors in all the operations. From this release onwards, the minimum alignment requirement has been relaxed down to 32 bit for input tensors in matrix multiplication and convolution for NVIDIA Ampere Architecture GPUs. For output tensors in any operation and input tensors for pointwise operations, the minimum alignment requirement has been relaxed down to 8 bit.

  ▶ Added support for ARM servers.

▶ **Documentation improvements** functions:

  ▶ We added documentation for the following data types and API.

    ▶ CUDNN_BACKEND_OPERATION_REDUCTION_DESCRIPTOR
    ▶ CUDNN_BACKEND_POINTWISE_DESCRIPTOR
    ▶ CUDNN_BACKEND_REDUCTION_DESCRIPTOR
    ▶ cudnnBackendBehaviorNote_t

- ▶ [cudnnBackendTensorReordering_t](#)
- ▶ [cudnnBnFinalizeStatsMode_t](#)
- ▶ [cudnnPaddingMode_t](#)
- ▶ [cudnnResampleMode_t](#)

Fixed Issues

- ▶ Users of cuDNN's `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` when set to 3000 previously could experience a floating point exception when the filter size (filter width * filter height) is greater than or equal to 32. This issue is fixed in this release.

- ▶ Users of cuDNN's `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` when set to 58, 1063, or 2062 may now use the knob count `CUDNN_KNOB_TYPE_WORKSPACE` to set the allowable workspace of these engines.

- ▶ The documentation of [cudnnNormalizationForwardInference()](#) and [cudnnBatchNormalizationForwardInference()](#) has been improved for clarity.

- ▶ Previous versions of cuDNN may produce wrong results when used to compute a matrix multiplication or fusions containing a matrix multiplication on NVIDIA Ampere Architecture based GPUs. This issue has been fixed in this release.

Known Issues

- ▶ Users of cuDNN's `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING` may see `CUDNN_STATUS_BAD_PARAM` returned for a problem that should otherwise be supported by that choice of algo.

- ▶ `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, and `cudnnConvolutionBackwardData()` may generate illegal memory address errors on the NVIDIA Volta and NVIDIA Turing architectures. This issue existed in previous 8.3 releases as well.

- ▶ `cudnnDropoutForward()` and `cudnnDropoutBackward()` will return incorrect results when input or output tensors have overlapping strides.

- ▶ Users specifying backend engines 58, 1063, 2062, and 4039 using `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` with 1x1 convolutions and tensors with more than two GB elements (2G) will see `CUDNN_STATUS_EXECUTION_FAILED` in cuDNN 8.3.x.

- ▶ cuDNN may return `CUDNN_STATUS_EXECUTION_FAILED` from `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, or `cudnnConvolutionBackwardData()` when computing convolutions with large spatial dimensions and batch sizes. This issue will be addressed in a future release so that such problems will instead return `CUDNN_STATUS_NOT_SUPPORTED` where applicable.

- ▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX = 1025` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`) does not support tensors in which the product N*C*H*W of the output gradient tensor equals to or exceeds 2^31. This issue has been

present in all previous releases of cuDNN and exercising the use case for the engine results in incorrect results.

▶ The documentation for cudnnReorderFilterAndBias() requires corrections for clarity.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

▶ FFT and Winograd based algorithms for convolution do not support graph capture.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

▶ CUDA streams internal to cuDNN are not guaranteed to have the same priority as the user stream that is set by `cudnnSetStream()`. We recently discovered some issues that break our ability to document exceptions to this clearly.

▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

▶ cudnnPoolingBackward() enables both `x` and `y` data pointers (together with the related tensor descriptor handles) to be `NULL` for avg-pooling. This could save memory footprint and bandwidth.

▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking with the `cnn_infer` and `cnn_train` static sub libraries. This will come at a cost to the binary size of the application. This linkage requirement will be relaxed in a future release.

▶ Users of cuDNN 8.4.0 may observe a slowdown in the Single Shot Multibox Detector (SSD) model. This will be fixed in a future release.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix.

Limitations

▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX =1001` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`) does not support tensors in which the product N*H*W of the output gradient tensor equals to or exceeds 2^31. This issue has been present in all previous releases of cuDNN and exercising the use case for the engine would show incorrect results.

▶ Versions of cuDNN before the 8.0 release series do not support the NVIDIA Ampere Architecture and will generate incorrect results if used on that architecture. Furthermore, if used, training operations can succeed with a NaN loss for every epoch.

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later. It also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples must be installed in a writable location. If not installed in a writable location, the samples can crash.

▶ RNN and multihead attention API calls may exhibit nondeterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This happens when two buffer sizes (16 KB and 4 MB) are used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer

sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ [cudnnSpatialTfSamplerBackward()](#) returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, [cudnnOpsInferVersionCheck()](#)) to load the kernels in the sub library before opening graph capture.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

  ▶ cudnnBatchNormalizationBackward()

  ▶ cudnnBatchNormalizationBackwardEx()

  ▶ cudnnBatchNormalizationForwardTraining()

  ▶ cudnnBatchNormalizationForwardTrainingEx()

  ▶ cudnnGetBatchNormalizationBackwardExWorkspaceSize()

  ▶ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()

  ▶ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()

  ▶ cudnnGetNormalizationBackwardWorkspaceSize()

  ▶ cudnnGetNormalizationForwardTrainingWorkspaceSize()

  ▶ cudnnGetNormalizationTrainingReserveSpaceSize()

  ▶ cudnnNormalizationBackward()

  ▶ cudnnNormalizationForwardTraining()

▶ `cudnnAddTensor()` performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

▶ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

Deprecated Features

The following features are deprecated in cuDNN 8.3.3:

▶ We are deprecating the reporting of performance results in the *Best Practices For Using cuDNN 3D Convolutions* and will instead update these *Release Notes* if there is anything interesting to report release-over-release. Starting with cuDNN 8.4.0, this section will be removed. For past performance tables, refer to the *cuDNN Archives* > *Best Practices For Using cuDNN 3D Convolutions*.

▶ Updated and migrated the content from the *Best Practices For Using cuDNN 3D Convolutions* to the cuDNN Developer Guide. The Best Practices document has been deprecated.

1.3.   cuDNN Release 8.3.3

These are the NVIDIA cuDNN 8.3.3 Release Notes. These Release Notes include fixes from the previous cuDNN releases as well as the following additional changes.

These Release Notes are applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previously released cuDNN documentation, refer to the *NVIDIA cuDNN Archived Documentation*.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Various improvements were made in the runtime fusion engine:

   ▶ Added heuristics for convolution + x fusion and matmul + x fusion for NVIDIA Volta and NVIDIA Turing architectures.

   ▶ Updated the heuristics for matmul + x fusion for NVIDIA Ampere Architecture.

   ▶ Small performance improvement for the matmul + x fusion.

   ▶ Compilation time reduction.

▶ Improved the performance for NHWC INT8 max pooling.

▶ Updated the list of supported enums in the following data type references:

   ▶ cudnnBackendAttributeName_t

   ▶ cudnnBackendAttributeType_t

   ▶ cudnnBackendDescriptorType_t

   ▶ cudnnBackendNumericalNote_t

▶ Updated and migrated the content from the *Best Practices For Using cuDNN 3D Convolutions* to the cuDNN Developer Guide. The Best Practices document has been deprecated.

Fixed Issues

The following issues have been fixed in this release:

▶ Fixed an issue when fusing pointwise operation with a scalar (that is a [1, 1, 1, 1] shaped tensor) at the output of a matmul or a convolution. When the output is of integer type, the results may be inaccurate or wrong (due to float to INT8 truncation). After the fix, it will properly round to nearest with clamping.

▶ Fixed an issue inside the batch norm finalize descriptor where an implementation detail was erroneously logged. Such unexpected access could intermittently cause a segment fault.

▶ Convolution batch norm fusion engines invoked through the graph API only worked with `cudnnBackendTensor` descriptors with dimensions specified in "n,c,g,h,w" format. This has been fixed and `cudnnBackendTensors` with dimensions specified any of "n,c,h,w" and "n,c,g,h,w" formats can now be passed.

▶ Fixed a numerical overflow issue in the computation of softplus activation function in the runtime fusion engine that was resulting in `log(exp(x))` being computed as infinity for sufficiently large positive values of `x`.

▶ In previous releases, [cudnnTransformFilter()](#) and[cudnnTransformTensorEx()](#) could produce wrong values at some pixels when doing a folding transform. This has been fixed in the current release.

▶ Documentation has been updated for pooling forward and backward API functions. The documentation now discusses which data types and vectorizations are supported for the tensor descriptor arguments (this information was previously incomplete). For more information, refer to the [cudnnPoolingForward()](#) and [cudnnPoolingBackward()](#).

Known Issues

▶ `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, and `cudnnConvolutionBackwardData()` may generate illegal memory address errors on the NVIDIA Volta and NVIDIA Turing architectures. This issue existed in previous 8.3 releases as well.

▶ `cudnnDropoutForward()` and `cudnnDropoutBackward()` will return incorrect results when input or output tensors have overlapping strides.

▶ Users specifying backend engines 58, 1063, 2062, and 4039 using `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` with 1x1 convolutions and tensors with more than two GB elements (2G) will see `CUDNN_STATUS_EXECUTION_FAILED` in cuDNN 8.3.x.

▶ cuDNN may return `CUDNN_STATUS_EXECUTION_FAILED` from `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, or `cudnnConvolutionBackwardData()` when computing convolutions with large spatial dimensions and batch sizes. This issue will be addressed in a future release so that such problems will instead return `CUDNN_STATUS_NOT_SUPPORTED` where applicable.

▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX = 1025` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`) does not support tensors in which the product N*C*H*W of the output gradient tensor equals to or exceeds 2^{31}. This issue has been present in all previous releases of cuDNN and exercising the use case for the engine results in incorrect results.

▶ The documentation for [cudnnReorderFilterAndBias()](#) requires corrections for clarity.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

- ▶ cudnnPoolingForward() with pooling mode CUDNN_POOLING_AVG might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

- ▶ Convolutions (ConvolutionForward, ConvolutionBackwardData, and ConvolutionBackwardFilter) may experience performance regressions when run with math type CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION on CUDNN_DATA_FLOAT data (*input* and *output*).

- ▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

- ▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain dgrad NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

- ▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

- ▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

- ▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

- ▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

- ▶ FFT and Winograd based algorithms for convolution do not support graph capture.

- ▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

- ▶ CUDA streams internal to cuDNN are not guaranteed to have the same priority as the user stream that is set by cudnnSetStream(). We recently discovered some issues that break our ability to document exceptions to this clearly.

- ▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

- ▶ cudnnPoolingBackward() enables both x and y data pointers (together with the related tensor descriptor handles) to be NULL for avg-pooling. This could save memory footprint and bandwidth.

- ▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking with the cnn_infer and cnn_train static sub libraries. This will come at a cost to the binary size of the application. This linkage requirement will be relaxed in a future release.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

▶ `CUDNN_ATTR_ENGINE_GLOBAL_INDEX =1001` (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`) does not support tensors in which the product N*H*W of the output gradient tensor equals to or exceeds 2^31. This issue has been present in all previous releases of cuDNN and exercising the use case for the engine would show incorrect results.

▶ Versions of cuDNN before the 8.0 release series do not support the NVIDIA Ampere Architecture and will generate incorrect results if used on that architecture.

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later. It also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples must be installed in a writable location. If not installed in a writable location, the samples can crash.

▶ RNN and multihead attention API calls may exhibit nondeterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This happens when two buffer sizes (16 KB and 4 MB) are used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

- ▸ cudnnBatchNormalizationBackward()
- ▸ cudnnBatchNormalizationBackwardEx()
- ▸ cudnnBatchNormalizationForwardTraining()
- ▸ cudnnBatchNormalizationForwardTrainingEx()
- ▸ cudnnGetBatchNormalizationBackwardExWorkspaceSize()
- ▸ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()
- ▸ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()
- ▸ cudnnGetNormalizationBackwardWorkspaceSize()
- ▸ cudnnGetNormalizationForwardTrainingWorkspaceSize()
- ▸ cudnnGetNormalizationTrainingReserveSpaceSize()
- ▸ cudnnNormalizationBackward()
- ▸ cudnnNormalizationForwardTraining()

▸ `cudnnAddTensor()` performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

▸ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

Deprecated Features

The following features are deprecated in cuDNN 8.3.3:

▸ We are deprecating the reporting of performance results in the *Best Practices For Using cuDNN 3D Convolutions* and will instead update these *Release Notes* if there is anything interesting to report release-over-release. Starting with cuDNN 8.4.0, this section will be removed. For past performance tables, refer to the *cuDNN Archives* > *Best Practices For Using cuDNN 3D Convolutions*.

▸ Updated and migrated the content from the *Best Practices For Using cuDNN 3D Convolutions* to the cuDNN Developer Guide. The Best Practices document has been deprecated.

1.4.   cuDNN Release 8.3.2

This is the NVIDIA cuDNN 8.3.2 Release Notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These Release Notes are

applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the [cuDNN Archived Documentation](#).

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ In the runtime fusion engine, pointwise fusion for batched matmul was extended to support operations with full tensor in the epilog.

Announcements

▶ Debian and RPM local installers are now provided on the [cuDNN download page](#). For more information, refer to the *cuDNN Installation Guide*.

▶ Individual cuDNN packages can be found in the CUDA repository:https://developer.download.nvidia.com/compute/cuda/repos/

▶ As part of a renewed effort to provide tarball and zip archive deliverables for NVIDIA products, the format has changed from existing `.txz` archives. For more information about `*-archive.tar.xz` and `*-archive.zip` deliverables, refer to the [Tarball and Zip Archive Deliverables](#) and the [README for build-system-archive-import-examples](#).

Fixed Issues

▶ cuDNN multihead attention produces incorrect results in case the `postDropout` feature is enabled. The issue has been fixed in this release.

▶ Running `convBiasAct` in `CUDNN_CROSS_CORRELATION` mode could result in incorrect results if the `GroupedDirect` engine is selected.

▶ Documentation has been updated for pooling forward and backward API functions, to update which data types and vectorizations are supported for the tensor descriptor arguments. For more information, refer to the [cudnnPoolingForward()](#) and [cudnnPoolingBackward()](#).

▶ The documentation in the [Reproducibility](#) chapter in the *cuDNN Developer Guide* has been improved upon for clarity.

▶ The documentation for the `CUDNN_BACKEND_OPERATION_MATMUL_DESCRIPTOR`, `CUDNN_BACKEND_OPERATION_RESAMPLE_FWD_DESCRIPTOR`, and `CUDNN_BACKEND_OPERATION_RESAMPLE_BWD_DESCRIPTOR` in the *[cuDNN API Reference](#)* has been improved upon for clarity.

▶ Use of `CUDNN_TENSOR_NCHW_VECT_C` with `cudnnReorderFilterAndBias()` could generate incorrect results when the reordered filter data was used incorrectly within cuDNN. Direct use of `cudnnConvolutionForward()` or `cudnnConvolutionBiasActivationForward()` without `cudnnReorderFilterAndBias()` was unaffected by this issue.

▶ Compared to cuDNN 8.3.0, there was an overall ~5% regression on `convBiasAct` layers on PG199/PG189. The maximum performance regression was around 3x for a select few cases. This issue has been fixed in this release.

Known Issues

▶ cuDNN may return `CUDNN_STATUS_EXECUTION_FAILED` from `cudnnConvolutionForward()`, `cudnnConvolutionBiasActivationForward()`, or `cudnnConvolutionBackwardData()` when computing convolutions with large spatial dimensions and batch sizes. This issue will be addressed in a future release so that such problems will instead return `CUDNN_STATUS_NOT_SUPPORTED` where applicable.

▶ Versions of cuDNN before the 8.0 release series do not support the NVIDIA Ampere Architecture and will generate incorrect results if used on that architecture.

▶ Data gradient `backendEngine` 25 (which is part of legacy `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`) does not support tensors in which the product N*C*H*W of the output gradient tensor equals to or exceeds 2^31. This issue has been present in all previous releases of cuDNN and exercising the use case for the engine results in incorrect results.

▶ The documentation for cudnnReorderFilterAndBias() needs some corrections for clarity.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

- ▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

- ▶ FFT and Winograd based algorithms for convolution do not support graph capture.

- ▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

- ▶ CUDA streams internal to cuDNN are not guaranteed to have the same priority as the user stream that is set by cudnnSetStream(). We recently discovered some issues that break our ability to document exceptions to this clearly.

- ▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

- ▶ cudnnPoolingBackward() allows both x and y data pointers (together with the related tensor descriptor handles) to be NULL for avg-pooling. This could save memory footprint and bandwidth.

- ▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking with the cnn_infer and cnn_train static sub libraries. This will come at a cost to the binary size of the application. This linkage requirement will be relaxed in a future release.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

- ▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of CUDNN_STATUS_NOT_SUPPORTED or CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING will be returned.

- ▶ Samples can crash unless they are installed in a writable location.

- ▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

  When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel

selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

- ▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

- ▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

- ▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

- ▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

- ▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

  - ▶ cudnnBatchNormalizationBackward()
  - ▶ cudnnBatchNormalizationBackwardEx()
  - ▶ cudnnBatchNormalizationForwardTraining()
  - ▶ cudnnBatchNormalizationForwardTrainingEx()
  - ▶ cudnnGetBatchNormalizationBackwardExWorkspaceSize()
  - ▶ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()
  - ▶ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()
  - ▶ cudnnGetNormalizationBackwardWorkspaceSize()
  - ▶ cudnnGetNormalizationForwardTrainingWorkspaceSize()
  - ▶ cudnnGetNormalizationTrainingReserveSpaceSize()
  - ▶ cudnnNormalizationBackward()
  - ▶ cudnnNormalizationForwardTraining()

- ▶ cudnnAddTensor() performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

- ▶ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

Deprecated Features

The following features are deprecated in cuDNN 8.3.2:

- ▶ We are deprecating the reporting of performance results in the *Best Practices For Using cuDNN 3D Convolutions* and will instead update these *Release Notes* if there is anything interesting to report release-over-release. Starting with cuDNN 8.4.0, this section will be

removed. For past performance tables, refer to the *cuDNN Archives* > *Best Practices For Using cuDNN 3D Convolutions*.

1.5.   cuDNN Release 8.3.1

This is the NVIDIA cuDNN 8.3.1 Release Notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These Release Notes are applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Announcements

▶ Debian and RPM local installers are now provided on the cuDNN download page. For more information, refer to the *cuDNN Installation Guide*.

▶ Individual cuDNN packages can be found in the CUDA repository:https:// developer.download.nvidia.com/compute/cuda/repos/

▶ As part of a renewed effort to provide tarball and zip archive deliverables for NVIDIA products, the format has changed from existing `.txz` archives. For more information about `*-archive.tar.xz` and `*-archive.zip` deliverables, refer to the Tarball and Zip Archive Deliverables and the README for build-system-archive-import-examples.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ In the runtime fusion engine:

   ▶ Pointwise logical and comparison operators are now supported, including `CUDNN_POINTWISE_LOGICAL_AND`, `CUDNN_POINTWISE_LOGICAL_OR`, `CUDNN_POINTWISE_LOGICAL_NOT`, `CUDNN_POINTWISE_CMP_EQ`, `CUDNN_POINTWISE_CMP_NEQ`, `CUDNN_POINTWISE_CMP_GT`, `CUDNN_POINTWISE_CMP_GE`, `CUDNN_POINTWISE_CMP_LT`, and `CUDNN_POINTWISE_CMP_LE`. As part of this feature, support for loading/storing/computing with boolean tensors has also been added.

   ▶ Batch support was added for matmul operation. Also, it is allowed to have batch broadcasting. The same matrix A or B can be broadcasted across the batch for matmul operation.

   ▶ The leading dimension support (reflected in the strides of the tensors) was added for matmul operation. It is allowed to compute matmul operation with unpacked tensors.

Fixed Issues

The following issues have been fixed in this release:

▶ cudnnConvolutionBiasActivationForward() could in some cases silently apply a ReLU operation when `Identity` was requested. This issue has been fixed in this release.

▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING`, `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING`, and `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` could exhibit illegal memory access in cuDNN v8 releases. This issue has been fixed in this release.

▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` was wrongly marked with numerical note `CUDNN_NUMERICAL_NOTE_REDUCED_PRECISION_REDUCTION` when output data type is float or double. This issue has been fixed in this release.

▶ There was an error in the documentation for determinism of cudnnConvolutionBackwardFilter() by algo. This issue has been corrected in this release.

▶ When the user selected `algo0` (`CUDNN_RNN_ALGO_STANDARD`) in `cudnnRNNBackwardData_v8()` or invoked the legacy functions, such as `cudnnRNNBackwardDataEx()`, `cudnnRNNBackwardData()`, and the number of RNN layers was more than eight in a unidirectional model or more than four in a bidirectional model, then some internal streams used to parallelize computations may be default streams (aka stream `0`). The computational performance would most likely be affected in those cases. This issue has been fixed in this release.

▶ Calling cudnnSoftmaxForward() with `CUDNN_SOFTMAX_MODE_CHANNEL` mode and N==1 in NCHW layout would result in incorrect results in cuDNN 8.3.0. This has been fixed in this release.

Known Issues

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

▶ FFT and Winograd based algorithms for convolution do not support graph capture.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

▶ In general, the internal CUDA streams inside cuDNN will have the same priority as the user stream that is set by `cudnnSetStream()` (instead of always having default priority). There are two exceptions:

   1. When the user stream is in capture mode (that is, `cudaStreamCaptureStatusActive==1`), the cuDNN-owned streams will still have default priority, and

   2. RNN functions `cudnnRNNForward()`, `cudnnRNNBackwardData_v8()`, `cudnnRNNBackwardWeights_v8()`, and their legacy counterparts still use default priority CUDA streams or higher priority streams to launch concurrent and cooperative grids.

▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

▶ cudnnPoolingBackward() allows both `x` and `y` data pointers (together with the related tensor descriptor handles) to be `NULL` for avg-pooling. This could save memory footprint and bandwidth.

▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking with the `cnn_infer` and `cnn_train` static sub libraries. This will come at a cost to the binary size of the application. This linkage requirement will be relaxed in a future release.

▶ Compared to cuDNN 8.3.0, there is an overall ~5% regression on convBiasAct layers on PG199/PG189. The maximum performance regression is around 3x for a select few cases.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of CUDNN_STATUS_NOT_SUPPORTED or CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the CUBLAS_WORKSPACE_CONFIG environmental variable, for example, :16:8 or :4096:2.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is :16:8:4096:2, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the :16:8 non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the _ALGO_0 algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions C, D (if 3D convolution), H, and W is not supported on devices older than NVIDIA Volta. To prevent a

potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

   ▶ cudnnBatchNormalizationBackward()
   ▶ cudnnBatchNormalizationBackwardEx()
   ▶ cudnnBatchNormalizationForwardTraining()
   ▶ cudnnBatchNormalizationForwardTrainingEx()
   ▶ cudnnGetBatchNormalizationBackwardExWorkspaceSize()
   ▶ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()
   ▶ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()
   ▶ cudnnGetNormalizationBackwardWorkspaceSize()
   ▶ cudnnGetNormalizationForwardTrainingWorkspaceSize()
   ▶ cudnnGetNormalizationTrainingReserveSpaceSize()
   ▶ cudnnNormalizationBackward()
   ▶ cudnnNormalizationForwardTraining()

▶ `cudnnAddTensor()` performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

▶ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

1.6.　cuDNN Release 8.3.0

This is the NVIDIA cuDNN 8.3.0 release notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack™ users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Announcements

▶ cuDNN version 8.3.0 depends on cuBLAS as a shared library dependency.

▶ The cuDNN version 8.3.0 `libcudnn_static.a` deliverable is replaced with the following:

  ▶ `libcudnn_ops_infer_static.a`

  ▶ `libcudnn_ops_train_static.a`

  ▶ `libcudnn_cnn_infer_static.a`

  ▶ `libcudnn_cnn_train_static.a`

  ▶ `libcudnn_adv_infer_static.a`

  ▶ `libcudnn_adv_train_static.a`

▶ cuDNN version 8.3.0 depends on zlib as a shared library dependency. Refer to the zlib instructions in the *NVIDIA cuDNN Installation Guide* for instructions.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ WSL 2 is released as a preview feature in this cuDNN 8.3.0.

▶ Various improvements were made in our multihead attention API:

  ▶ Added HSH support - FP16 data type with FP32 math precision. Allow to achieve FP16 mixed-precision Tensor Core performance without sacrificing accuracy.

  ▶ Added support to bias gradient computation. Before cuDNN 8.3.0, bias was supported only for inference.

▶ Multihead attention has two dropout layers active in training mode. The first dropout operation is applied directly to the softmax output. The second dropout operation alters the multihead attention output, just before the point where residual connections are added. Before cuDNN 8.3.0, only the first dropout layer was supported.

▶ Significant performance improvement out of the box (no changes are required from users) for both multihead attention forward and backward paths.

▶ Various improvements were made in our runtime fusion engine:

  ▶ The cuDNN runtime fusion engine now supports resample operations of upsample and downsample. Support has been added for 2*2 average pooling with stride 2 and upsample by a factor of 2 using bilinear interpolation. The datatype supported is FP32 and the compute datatype is FP32. The resample operations can also be fused with other operations provided the input to the resample operation is located in global memory.

  ▶ The cuDNN runtime fusion engine is now generalized to accurately obey the intermediate storage datatype users specified in the operation graph. The support datatypes include INT8, BF16, FP16, INT32, FP32. As a general rule, we recommend users to use FP32 as the intermediate storage type that provides balanced numerical precision and performance.

  ▶ The cuDNN runtime fusion engine now supports batched matmul operation with row/column broadcast or row/column reduction operations in the epilog.

  ▶ The cuDNN runtime fusion engine now does numeric clamping while converting from a data type with a larger dynamic range to one with a relatively smaller dynamic range to avoid numeric overflows at all times.

  ▶ The cuDNN runtime fusion engine extends the support for broadcast pointwise operations in the epilogue to now include those between a tensor and a scalar value as well.

  ▶ Extended general fusion heuristic support to convolution forward and backward data and weight gradient operation patterns, with FP16, TF32, INT8 I/O data types, to ensure a good heuristic selection to improve out-of-the-box performance.

▶ Added more detailed error reporting that is accessible from the existing API log or the logging callback function. Error and warning severity levels are added into the error reporting. Environment variables `CUDNN_LOGERR_DBG` and `CUDNN_LOGWARN_DBG` can be used to enable these severity levels respectively. Within these error severity levels, the error or warning message will now include a traceback of the error conditions triggered the error as hints for troubleshooting purposes.

▶ Engine heuristics now supports a new mode called `HEUR_MODE_FALLBACK` which gives a list of engine configurations that run most of the convolution problems without the performance guarantee. Use this mode when all engines suggested by heuristics are not supported.

▶ In prior cuDNN versions, certain engines required reordered filters for int8x32 format, but there was no way to disambiguate whether the filter was

reordered. Engines that require reordered filters now have a new behavior note `CUDNN_BEHAVIOR_NOTE_REQUIRES_FILTER_REORDER` which specifies the tensors must be reordered before being passed to the engine.

▶ RNN functions `cudnnRNNBackwardData_v8()`, `cudnnRNNBackwardDataEx()`, and `cudnnRNNBackwardData()` have been improved to internally invoke the cooperative group API `cudaLaunchCooperativeKernel()` to launch GPU kernels when threads must synchronize across all CUDA® thread blocks of a grid. Starting in CUDA 11.2, the `cudaLaunchCooperativeKernel()` function is able to run multiple cooperative grids concurrently in multiple streams. This feature has been used in `CUDNN_RNN_ALGO_PERSIST_STATIC` and `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` algorithms to improve the computational performance. A method of launching these types of kernels using `cudaLaunchCooperativeKernel()` is more robust in preventing potential deadlocks when in rare scenarios when multiple cooperative grids are launched concurrently.

cuDNN 8.3 compiled with CUDA 10.2 must still rely on a regular method of launching kernels. Deadlocks are mitigated by employing higher priority CUDA streams. Currently, `cuDNN RNN APIs still` use higher priority streams, however, in future cuDNN versions, priorities of auxiliary streams will match the priority of the user stream defined by the `cudnnSetStream()` call. Future cuDNN versions will also use the `cudaLaunchCooperativeKernel()` API to launch cooperative grids in forward RNN functions such as `cudnnRNNForward()`.

Fixed Issues

The following issues have been fixed in this release:

▶ `cudnnAddTensor()` did not support some tensor shapes that were previously specified as supported. This issue has been fixed in this release.

▶ When using the cuDNN CTC Loss API function, the computed gradients array was not zero initialized. This meant it was possible the gradients array returned a mix of valid values and uninitialized values. This issue has been fixed in this release.

▶ Compared to version 8.0.5, legacy convolution APIs increased CPU computational costs. On x86, this was measured to be as high as 10 microseconds. This issue has been fixed in this release.

▶ `cudnnSetStream()` API was generating errors when graph capture was enabled. This issue is fixed in this release.

▶ There was a known 60% performance regression for ResNet-50 on the GTX 1080 when run using FP16 data with large batch sizes (over 128). This regression has been fixed in this release.

▶ In some cases, `cudnnConvolutionBackwardFilter()` generates numerically imprecise results when used with algo set to `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`. This issue is most frequently encountered with three-dimensional spatial tensors. Users of the backend API may explicitly avoid backend engine 2032 or consider the numerical

notes of engines and reject any marked as offering reduced precision reduction (`CUDNN_NUMERICAL_NOTE_REDUCED_PRECISION_REDUCTION`).

▶ For `cudnnConvolutionBiasActivationForward()`, there was previously a restriction on aliasing device memory pointers labeled `x` and `z` in the documentation of that function. This restriction has been relaxed so that `x` may alias `z` by pointing to the same device memory location if desired. Note that the restriction against aliasing the pointers labeled `x` and `y` remains.

▶ cuDNN may be observed to contain a small leak related to the use of dlopen. Currently, this is believed to be a false positive when indicated by valgrind. Should this thinking change, the known issues of this document will reflect that understanding in subsequent releases.

▶ Previously, on NVIDIA Pascal and Maxwell architectures, users of cuDNN's 8.X's backend engine 34 with `CUDNN_CONVOLUTION` mode set for forward convolution could witness-illegal memory access when this engine is specifically selected outside of heuristic query. This issue has been fixed in this release.

▶ Previously, on K80 GPUs when `cudnnConvolutionForward()` is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width Q is 1 and both height and width padding are zero; this particular case will now be rejected by cuDNN as not supported in this and all other successive releases for this GPU architecture.

▶ cuDNN does not package `libfreeimg` as a static library for users of cuDNN's MNIST sample code. The included `readme.txt` file contains instructions on where to locate this dependency and how to compile and link this sample.

▶ The parameters section for cudnnBatchNormalizationForwardInference() has been updated to reflect a correct `*y` description.

▶ Compared to cuDNN 7.6.5, there was a known performance regression on various convolutional models using INT8 data types on NVIDIA Volta GPUs. This issue has been fixed in this release.

▶ A memory leak as well as a possible delayed memory deallocation in the cuDNN runtime fusion engine have been fixed.

▶ In previous releases of cuDNN 8, user applications might crash in rare instances due to large stack allocation requirements; this issue is fixed in this release by preferring heap allocation in cases where large stack allocations were previously occurring.

▶ Some dgrad batchnorm fusion engines were previously not supported on Windows. We now support this starting in cuDNN 8.3.0.

▶ The documentation for cudnnReorderFilterAndBias() needed some corrections for clarity. The topic has been updated in this release.

Known Issues

▶ When the user selects `algo0` (`CUDNN_RNN_ALGO_STANDARD`) in `cudnnRNNBackwardData_v8()` or invokes the legacy functions, such as

`cudnnRNNBackwardDataEx()`, `cudnnRNNBackwardData()`, and the number of RNN layers is more than eight in a unidirectional model or more than four in a bidirectional model, then some internal streams used to parallelize computations may be default streams (aka stream 0). RNN `algo0` dgrad APIs will not crash and the numerical results will be correct but the computational performance will likely be affected in those cases.

▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

▶ FFT and Winograd based algorithms for convolution do not support graph capture.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

▶ The internal CUDA streams inside cuDNN 8.3.0 will have the same priority as the user stream that is set by `cudnnSetStream()` (instead of always having default priority). There are two limitations:

1. When the user stream is in capture mode (that is, `cudaStreamCaptureStatusActive==1`), the cuDNN-owned streams will still have default priority, and

2. RNN functions `cudnnRNNForward()`, `cudnnRNNBackwardData_v8()`, `cudnnRNNBackwardWeights_v8()`, and their legacy counterparts still use default priority CUDA streams or higher priority streams to launch concurrent and cooperative grids.

▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

▶ cudnnPoolingBackward() allows both `x` and `y` data pointers (together with the related tensor descriptor handles) to be `NULL` for avg-pooling. This could save memory footprint and bandwidth.

▶ When applications using cuDNN with an older 11.x CUDA toolkit in compatibility mode are tested with compute-sanitizer, `cuGetProcAddress` failures with error code 500 will arise due to missing functions. This error can be ignored, or suppressed with the `--report-api-errors no` option, as this is due to CUDA backward compatibility checking if a function is usable with the CUDA toolkit combination. The functions are introduced in a later version of CUDA but are not available on the current platform. The absence of these functions is harmless and will not give rise to any functional issues.

▶ Calling cudnnSoftmaxForward() with `CUDNN_SOFTMAX_MODE_CHANNEL` mode and N==1 in NCHW layout may result in incorrect results. This will be fixed in the next release.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the

same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions C, D (if 3D convolution), H, and W is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, [cudnnTransformTensor()](#) can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.

3. Convert all output tensors from a supported type to your desired output type.

> 📝 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORTED` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

▶ The spatial persistent batch normalization API is only available for NVIDIA Pascal and later architectures. Pre-Pascal architectures return `CUDNN_STATUS_ARCH_MISMATCH` instead. The affected APIs include:

  ▶ cudnnBatchNormalizationBackward()
  ▶ cudnnBatchNormalizationBackwardEx()
  ▶ cudnnBatchNormalizationForwardTraining()
  ▶ cudnnBatchNormalizationForwardTrainingEx()
  ▶ cudnnGetBatchNormalizationBackwardExWorkspaceSize()
  ▶ cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()
  ▶ cudnnGetBatchNormalizationTrainingExReserveSpaceSize()
  ▶ cudnnGetNormalizationBackwardWorkspaceSize()
  ▶ cudnnGetNormalizationForwardTrainingWorkspaceSize()
  ▶ cudnnGetNormalizationTrainingReserveSpaceSize()

- ▶ cudnnNormalizationBackward()
- ▶ cudnnNormalizationForwardTraining()
- ▶ cudnnAddTensor() performance may regress from 8.2 to 8.3 for pre-Pascal architectures.

1.7.   cuDNN Release 8.2.4

This is the cuDNN 8.2.4 release notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Known Issues

- ▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

- ▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

- ▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

- ▶ cudnnAddTensor() does not support all tensor shapes even though the cuDNN documentation says otherwise.

- ▶ cudnnPoolingForward() with pooling mode CUDNN_POOLING_AVG might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

- ▶ Convolutions (ConvolutionForward, ConvolutionBackwardData, and ConvolutionBackwardFilter) may experience performance regressions when run with math type CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION on CUDNN_DATA_FLOAT data (*input* and *output*).

- ▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

- Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

- Compared to cuDNN 7.6.5, there are known performance regressions on various convolutional models using INT8 data types on NVIDIA Volta GPUs.

- The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

- It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

- There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

- The documentation for cudnnReorderFilterAndBias() needs some corrections for clarity.

- Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

- NVIDIA Turing users of cuDNN can observe intermittent illegal memory access errors for some convolution workloads.

- FFT and Winograd based algorithms for convolution do not support graph capture.

- In a multi-GPU setting, with complex scheduling, cuDNN can segfault. It is not clear that this is a cuDNN issue, but the issue is under active investigation so that the known limitations section of this document can be updated in a future release.

- Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

- There is a known 60% performance regression for ResNet-50 on the GTX 1080 when run using FP16 data with large batch sizes (over 128).

- Users of the static library requiring the best possible convolution performance should use whole-archive linking. This will come at a cost to the binary size that will require resolution in a future release, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

- cuDNN may contain a small memory leak related to the usage of `dlopen()` within the library; this is not confirmed but currently under investigation. The possible leak does not affect Windows users or users of the static library.

- On NVIDIA Pascal and Maxwell architectures, users of cuDNN's 8.0 backend engine 34 for forward convolution can witness-illegal memory access when this engine is specifically selected outside of heuristic query. Heuristics users of these architectures will not witness this issue, as happened in previous versions. The possibility of the illegal memory access affects all previous versions of cuDNN 8.0 and will be fixed in a future release.

- When using the cuDNN CTC Loss API function, the computed gradients array is not zero initialized. When sequence lengths are exceeded, some gradient entries are returned uninitialized.

▶ The internal CUDA streams inside cuDNN 8.2.4 will have the same priority (instead of the default priority) as the user stream that is set by `cudnnSetStream()`, while an exception/limitation is that they will have priority as (highest - 1) for the user stream with the highest priority. This is true only when the user stream is NOT in capture mode (`cudaStreamCaptureStatusActive`), otherwise the behavior does not change.

▶ Fusion engine operation mode 16 is not currently supported on Windows; this will be fixed in a future release.

▶ The functional support criteria of cuDNN's convolution kernels is not required to consider padding. Users of cuDNN can witness an unexpected lack of problem support when forward convolution spatial dimensions are less than the filter size and padding is nonzero, however, is sufficient to extend spatial dimensions to or beyond filter dimensions. This is commonly observed with, but not limited to, INT8 convolution kernels.

Limitations

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte

boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 🗨 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the

engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ [cudnnSpatialTfSamplerBackward()](#) returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, [cudnnOpsInferVersionCheck()](#)) to load the kernels in the sub library before opening graph capture.

▶ Starting in cuDNN version 8.1.0, we are no longer shipping the `libfreeimg` static library with the MNIST sample. Users can follow the instructions in the `readme.txt` file to download and compile the library separately and link with the MNIST sample.

▶ For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

▶ Starting in version 8.1, cuDNN uses AVX intrinsics on the x86_64 architecture; users of this architecture without support for AVX intrinsics may see illegal instruction errors.

1.8.　cuDNN Release 8.2.2

This is the cuDNN 8.2.2 release notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the [cuDNN Archived Documentation](#).

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Experimental runtime fusion heuristics are now supported to facilitate an intelligent and efficient heuristic recommendation based on the predicted execution time for the runtime fusion engine. The current coverage is limited to fusion patterns involving a convolution forward operation in FP16 mixed precision configuration on NVIDIA Ampere Architecture GPUs. We will continue to expand the support and improve the prediction accuracy in future releases.

▶ The cuDNN runtime fusion now supports pure pointwise fusion or pointwise plus reduction fusion. It supports FP16/FP32 as I/O type and FP32 as compute type.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

- ▶ For platforms that ship a compiler version older than GCC 6 by default, linking to static cuDNN using the default compiler is not supported.

- ▶ There was a 15% performance regression for inference on the PyTorch WaveGlow model on the NVIDIA Turing architecture. This regression has been fixed.

- ▶ The `convolve_common_engine_int8_NHWC` kernel had an undesired FP32 > INT32 truncation before outputting the FP32 result directly. This issue has been fixed in this release.

- ▶ In previous cuDNN versions, `cudnnRNNBackwardData()`, `cudnnRNNBackwardDataEx()`, or `cudnnRNNBackwardData_v8()` could return `CUDNN_STATUS_INTERNAL_ERROR` when `CUDNN_RNN_ALGO_PERSIST_STATIC` and `CUDNN_LSTM` were selected. This issue occurred mainly on smaller GPUs, such as NVIDIA Turing with 36 or 48 SMs and smaller `hiddenSize` values. This issue has been fixed in this release.

- ▶ NVIDIA Turing GTX 16xx users of cuDNN would observe invalid values in convolution output. This issue has been fixed in this release.

- ▶ Users would experience NCHW transformations causing a floating point exception and the CPU reference code producing incorrect results for tensor format `CUDNN_TENSOR_NCHW_VECT_C`. Corner cases in the convolution sample code have been fixed in this release.

Known Issues

- ▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

- ▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

- ▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

- ▶ cudnnAddTensor() does not support all tensor shapes even though the cuDNN documentation says otherwise.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ Compared to cuDNN 7.6.5, there are known performance regressions on various convolutional models using INT8 data types on NVIDIA Volta GPUs.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

▶ It is possible, starting in cuDNN 7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ The documentation for cudnnReorderFilterAndBias() needs some corrections for clarity.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

▶ NVIDIA Turing users of cuDNN can observe intermittent illegal memory access errors for some convolution workloads.

▶ FFT and Winograd based algorithms for convolution do not support graph capture.

▶ In a multi-GPU setting, with complex scheduling, cuDNN can segfault. It is not clear that this is a cuDNN issue, but the issue is under active investigation so that the known limitations section of this document can be updated in a future release.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

▶ There is a known 60% performance regression for ResNet-50 on the GTX 1080 when run using FP16 data with large batch sizes (over 128).

▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking. This will come at a cost to the binary size that will require resolution in a future release, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ cuDNN may contain a small memory leak related to the usage of `dlopen()` within the library; this is not confirmed but currently under investigation. The possible leak does not affect Windows users or users of the static library.

▶ On NVIDIA Pascal and Maxwell architectures, users of cuDNN's 8.0 backend engine 34 for forward convolution can witness-illegal memory access; this affects all previous versions of cuDNN 8.0 and will be fixed in a future release.

▶ When using the cuDNN CTC Loss API function, the computed gradients array is not zero initialized. When sequence lengths are exceeded, some gradient entries are returned uninitialized.

Limitations

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of CUDNN_STATUS_NOT_SUPPORTED or CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the CUBLAS_WORKSPACE_CONFIG environmental variable, for example, :16:8 or :4096:2.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is :16:8:4096:2, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the :16:8 non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are

cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

   1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
   2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
   3. Convert all output tensors from a supported type to your desired output type.

> 🗨 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

- ▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.
- ▶ Starting in cuDNN version 8.1.0, we are no longer shipping the `libfreeimg` static library with the MNIST sample. Users can follow the instructions in the `readme.txt` file to download and compile the library separately and link with the MNIST sample.
- ▶ For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.
- ▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

Deprecated Features

The following features are deprecated in cuDNN 8.2.2:

- ▶ Support for Ubuntu 16.04 has been deprecated in cuDNN 8.2.2 for CUDA 11.4. For a list of supported OS, refer to the cuDNN Support Matrix.
- ▶ Support for RHEL7 for ppc64le has been deprecated in cuDNN 8.2.2 for CUDA 11.4. For a list of supported OS, refer to the cuDNN Support Matrix.

1.9.   cuDNN Release 8.2.1

This is the cuDNN 8.2.1 release notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

- ▶ The cuDNN runtime fusion engine now supports generating Tensor Core kernels with input tensors of:

  - ▶ Bfloat16 type and compute precision of FP32 (requires compute capability 8.0 or later). For Bfloat16 support, convolution I/O channels are required to be a multiple of 8.
  - ▶ INT8 and compute precision of INT32 (requires compute capability 7.5 or later) datatype and in NHWC layout. For INT8 support, the convolution I/O channels are required to be a multiple of 16, and unlike the `NCHW_VECT_C` kernels, filter and bias reordering is not required.

  In the fused pointwise/reduction operations, FP32 is the compute precision supported.

- ▶ The cuDNN runtime fusion engine has added experimental Tensor Core kernel generation support for NVIDIA Volta (compute capability 7.0) and NVIDIA Xavier (compute capability

7.2). The supported input tensor data type is FP16, compute precision is FP32, and the supported layout is NHWC. However, reduction fusion is not yet supported and we are working on further generalizing the support.

▶ Equations in the documentation are now supported in Chrome.

▶ The backend API now supports fused convolution-scale-bias-activation with per-channel-scaling by matching the operation graph.

▶ cudnnPoolingBackward() allows both x and y data pointers (together with the related tensor descriptor handles) to be NULL for avg-pooling. This could save memory footprint and bandwidth.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

▶ In some cases, NVIDIA Ampere Architecture users of cuDNN 8.1 cudnnGetConvolutionBackwardFilterAlgorithm_v7() could receive a workspace that was insufficient for computing the calculation with cudnnConvolutionBackwardFilter(). This issue has been fixed in this release.

▶ Many convolution models were experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This included ResNet-50 with up to 2x performance difference and ResNeXt up to 10x the performance difference. Many of these performance issues have been fixed in this release.

▶ Compared to cuDNN version 8.0.5, there was a known 8% performance regression on the SSD ResNet-50 model on the NVIDIA Ampere Architecture. This issue has been fixed in this release.

▶ L4T users of cuDNN could observe CUDNN_STATUS_EXECUTION_FAILED errors in some cases when performing convolutions using CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM. This issue has been fixed in this release.

▶ In cuDNN 8.2.0, if the user runs a Bi-directional RNN network with dropout enabled, the user may see non-deterministic outputs. This issue has been fixed in 8.2.1.

▶ There was a known 18% performance regression for inference on the PyTorch ResNet-50 v1.5 model on the NVIDIA Turing architecture. This issue has been fixed in this release.

▶ Known regressions on certain layers in cuDNN 8 regression in algorithm selection heuristics have been fixed on NVIDIA Volta and NVIDIA Pascal platforms.

▶ In older versions of cuDNN, when calling the API cudnnSetDropoutDescriptor(), a kernel launched by this API used to require a substantial amount of GPU memory for the stack. The memory is released when the kernel finishes and the stack size is changed back in

a way that is not thread safe. Starting in the 8.2.1 release, the extra memory is no longer required, and as a result, the thread safety concern is no longer present.

▶ In cuDNN 8.1.1, compared to cuDNN 8.1.0, there was a known regression in performance of the runtime fusion engine for convolution fused with ReLU in the epilog. This was caused due to the generalized support for parameterized ReLU. This issue has been fixed since the 8.2.0 release.

▶ Since cuDNN 8.0.4 until 8.2.0, certain SKUs of V100 GPU may encounter `CUDNN_STATUS_EXECUTION_FAILED` status or unspecified launch failure in a subsequent call to cudaDeviceSynchronize() when running RNN with cell mode of `CUDNN_LSTM` and `CUDNN_RNN_ALGO_PERSIST_STATIC` algorithm. This issue has been fixed in this release.

▶ Between cuDNN 8.1.0 and 8.2.0, if the user runs `cudnnRNN*()` API under CUDA compute sanitizer with `CUDNN_RNN_ALGO_PERSIST_STATIC_SMALL_H` algorithm, users may see errors like `Invalid __global__ read` reported by the CUDA compute sanitizer. This issue has been fixed in this release.

▶ Compared to cuDNN 8.0.0 preview, there is a known ~12% performance regression on vgg16 when run on Jetson Nano and TX2. This issue has been fixed in this release.

▶ Compared to cuDNN 7.6, there is a significant performance regression on Darknet when run on Jetson Nano. This issue has been fixed in this release.

Known Issues

▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ Some convolution models are experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This includes EfficientNet with up to 6x performance difference, UNet up to 1.6x performance difference and Tacotron up to 1.6x performance difference.

▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

▶ cudnnAddTensor() does not support all tensor shapes even though the cuDNN documentation says otherwise.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

- Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

- Compared to cuDNN 7.6.5, there are known performance regressions on various convolutional models using INT8 data types on NVIDIA Volta GPUs.

- The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet.

- It is possible, starting in cuDNN v7.6 and up to but not including 8.1.1, to leak memory when computing common convolution operations in rare cases.

- There is a known 15% performance regression for inference on the PyTorch WaveGlow model on the NVIDIA Turing architecture.

- There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

- The documentation for cudnnReorderFilterAndBias() needs some corrections for clarity.

- Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

- NVIDIA Turing GTX 16xx users of cuDNN can observe invalid values in convolution output.

- NVIDIA Turing users of cuDNN can observe intermittent illegal memory access errors for some convolution workloads.

- FFT and Winograd based algorithms for convolution do not support graph capture.

- In a multi-GPU setting, with complex scheduling, cuDNN can segfault. It is not clear that this is a cuDNN issue, but the issue is under active investigation so that the known limitations section of this document can be updated in a future release.

- Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case.

- There is a known 60% performance regression for ResNet-50 on the GTX 1080 when run using FP16 data with large batch sizes (over 128).

Limitations

- The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

- Samples can crash unless they are installed in a writable location.

- RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the

same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 📝 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionCheck()) to load the kernels in the sub library before opening graph capture.

▶ Starting in cuDNN version 8.1.0, we are no longer shipping the `libfreeimg` static library with the MNIST sample. Users can follow the instructions in the `readme.txt` file to download and compile the library separately and link with the MNIST sample.

▶ For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.

▶ Users of cuDNN must add the dependencies of cuBLAS to the linkers command explicitly to resolve the undefined symbols from cuDNN static libraries.

Deprecated Features

The following features are deprecated in cuDNN 8.2.1:

▶ Support for Ubuntu 16.04 will be deprecated in cuDNN 8.2.2 for CUDA 11.4. For a list of supported OS, refer to the cuDNN Support Matrix.

▶ Support for RHEL7 for ppc64le will be deprecated in cuDNN 8.2.2 for CUDA 11.4. For a list of supported OS, refer to the cuDNN Support Matrix.

1.10.   cuDNN Release 8.2.0

This is the cuDNN 8.2.0 release notes. This release includes fixes from the previous cuDNN v8.1.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Convolution with the backend API now supports tensor with more than 2**31 elements. The size and stride of each tensor dimension are still limited to 32-bit values.

▶ Convolution Heuristics Generalization has been improved for several GPUs. These improvements can be observed in the legacy API and version 8 API. In the version 8 API, these improvements are available in both `CUDNN_HEUR_MODE_INSTANT` and `CUDNN_HEUR_MODE_B`.

▶ The cuDNN runtime fusion engine now supports generating Tensor Core based fusion kernels in the following scenarios:

  ▶ When there is a scale+bias+ReLU pattern in the graph fused to the x input of a convolution forward operation

  ▶ When the graph contains 3D convolution forward, backward data, or backward filter operation

  ▶ When the graph contains a convolution backward data operation with non-unit convolution strides

  We are working on further generalization of this support.

▶ cuDNN C++ frontend has released the 0.2 version that adds more general support to activation forward and backward operations, matMul operation, and contains various bug fixes and clean ups. A few runtime fusion samples have also been added. For more information, refer to GitHub: cuDNN frontend.

▶ The new `RNN ALGO_STANDARD` implementation and heuristics tuning provides significant speedup (up to 100%), especially when the overall problem size is small (hidden size, batch size, and the number of timesteps).

▶ The RNN dropout implementation has been heavily optimized. The new implementation brings significant speed-up to all RNN algorithms when dropout is enabled.

▶ cuDNN RNN has moved to calling cuBLASLt on newer architectures (compute capability >= 7.0). As a result, the `CUBLAS_WORKSPACE_CONFIG` workaround for cuBLAS non-

deterministic behavior is no longer needed on those architectures. In addition, under repeated CUDA graph capture, cuBLASLt no longer allocates workspace repeatedly like cuBLAS.

▶ In the cuDNN v8 backend API, a new CUDNN_ATTR_ENGINE_BEHAVIOR_NOTE attribute has been added. Users can query the engine behaviors using this attribute similar to the numerical behaviors queried through the CUDNN_ATTR_ENGINE_NUMERICAL_NOTE attribute. Currently, the engine behavior note only shows whether an engine does runtime compilation or not. More behaviors may be added in future releases.

▶ cuDNN API logging for the v8 backend API has been significantly improved. Now more detailed information can be printed from the backend data structures, for example, tensors, operations, engines, and execution plans. We hope this can improve the development and debugging experience of cuDNN. Refer to this link for more instructions of how to enable API logging.

▶ cuDNN now supports SWISH activation in both forward and backward directions. It can be configured for use with cudnnActivationForward() and cudnnActivationBackward() by using CUDNN_ACTIVATION_SWISH with cudnnSetActivationDescriptor(). SWISH activation's parameter, commonly known as beta, may further be set using the newly added API function cudnnSetActivationDescriptorSwishBeta() and queried with cudnnGetActivationDescriptorSwishBeta().

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

▶ There was a performance regression in certain use cases comparing NVIDIA RTX 3090 using cuDNN version 8.x to NVIDIA RTX 2080 Ti using cuDNN version 7.x. This regression has been fixed in this release.

▶ There was a performance regression in the runtime engine for convolution fused with ReLU in the epilog in cuDNN 8.1.1. This regression has been fixed in this release.

▶ Compared to cuDNN version 7.6.5, there was a performance regression in certain grouped ConvolutionBackwardFilter cases on the NVIDIA Volta GPU architecture. This regression has been fixed in this release.

▶ CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT returned an internal error when the number of channels in the filter was greater than or equal to 65536. This issue has been fixed in this release.

▶ Compared to cuDNN version 8.0.2, there was a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case. This issue has been fixed in this release.

▶ Although the overall cuDNN library size has improved in cuDNN 8.1.0 with CUDA Toolkit 11.2 and greater, as compared to cuDNN 8.0.x, the cuDNN library remains large. We have attempted to moderate the severity of this issue in this release.

▶ Many convolution models were experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This included ResNet-50 with up to 2x performance difference, ResNeXt up to 10x performance difference and U-Net up to 3x performance difference. The performance issues have been fixed in this release.

▶ The ResNet-50 native FP32 inference issues have been fixed on NVIDIA Volta, NVIDIA Turing, and NVIDIA Ampere Architecture GPUs.

▶ We have eliminated anonymous structs in cuDNN public headers `cudnn_cnn_infer.h`, `cudnn_cnn_train.h`, and `cudnn_ops_infer.h` to allow forward struct declarations. The following five `typedef-s` were updated: cudnnConvolutionFwdAlgoPerf_t, cudnnConvolutionBwdDataAlgoPerf_t, cudnnConvolutionBwdFilterAlgoPerf_t, cudnnAlgorithm_t, and cudnnDebug_t

▶ cudnnActivationForward() could generate illegal memory access errors for tensors of more than $2^{**}30$ elements in the previous version of cuDNN 8. This issue has been fixed in this release.

▶ In previous releases, cudnnRNNBackwardWeights(), cudnnRNNBackwardWeightsEx(), and cudnnRNNBackwardWeights_v8() may generate wrong and non-deterministic results when dropout is enabled. A stream dependency issue has been fixed in the current release so users will no longer observe this issue.

▶ The heuristics in cudnnConvolutionBackwardFilter() have been improved for generalized cases. We have observed several convolution cases with up to ~100x performance improvements compared to cuDNN version 8.1.

▶ Compared to cuDNN 8.0.4, there was a known ~6% performance regression on ONNX-WaveGlow when run on NVIDIA TITAN RTX. This issue has been fixed in this release.

▶ Compared to cuDNN 7.6, there were known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs. This issue has been fixed in this release.

Known Issues

▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

▶ cudnnAddTensor() does not support all broadcast-able tensor shapes even though the cuDNN documentation says otherwise.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Compared to cuDNN 8.0.0 Preview, there is a known ~12% performance regression on vgg16 when run on Jetson Nano and TX2.

▶ Compared to cuDNN 8.0.4, there is a known ~6% performance regression on ONNX-WaveGlow when run on NVIDIA TITAN RTX.

▶ Compared to cuDNN 7.6, there is a significant performance regression on Darknet when run on Jetson Nano.

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.

▶ L4T users of cuDNN may observe `CUDNN_STATUS_EXECUTION_FAILED` errors in some cases when performing convolutions using `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`. This issue is being investigated.

▶ Users of the static library requiring the best possible convolution performance should use whole-archive linking. This will come at a cost to the binary size that will require resolution in a future release, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.

▶ Compared to cuDNN 7.6.5, there are known performance regressions on various convolutional models using INT8 data types on NVIDIA Volta GPUs.

▶ Compared to cuDNN 8.1.0, there is a known regression in performance of the runtime fusion engine for convolution fused with ReLU in the epilog. This is caused due to the generalized support for parameterized ReLU. Further optimizations are being worked on.

▶ The numeric behavior of INT8 operations including saturation behavior, accumulator data types, and so on, have not been documented as of yet. This is being worked on and will be resolved in a future release.

▶ It is possible, starting in cuDNN v7.6, to leak memory when computing common convolution operations in rare cases.

▶ There is a known 15% performance regression for inference on the PyTorch WaveGlow model on the NVIDIA Turing architecture.

▶ There is a known 25% performance regression for inference on the PyTorch SSD model on the NVIDIA Turing architecture.

▶ There is a known 18% performance regression for inference on the PyTorch ResNet-50 v1.5 model on the NVIDIA Turing architecture.

▶ The documentation for `cudnnReorderFilterAndBias()` needs some corrections for clarity. This will be fixed in a future release.

▶ Compared to cuDNN 8.0.5, there is a known ~17% performance regression on SSD models running on V100.

Limitations

▶ The runtime fusion engine is only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1 or later; it also requires the NVRTC from CUDA 11.2 update 1 or later. If this condition is not satisfied, the error status of `CUDNN_STATUS_NOT_SUPPORTED` or `CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING` will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are

cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

   1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
   2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
   3. Convert all output tensors from a supported type to your desired output type.

> 💬 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

- ▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionChec()) to load the kernels in the sub library before opening graph capture.

- ▶ Starting in cuDNN version 8.1.0, we are no longer shipping the libfreeimg static library with the MNIST sample. Users can follow the instructions in the readme.txt file to download and compile the library separately and link with the MNIST sample.

- ▶ For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.

1.11. cuDNN Release 8.1.1

This is the cuDNN 8.1.1 release notes. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

- ▶ The runtime fusion engine now supports the canonical NCHW/KCRS/NKPQ format for describing a tensor, in addition to the version 8 format that has the explicit group dimension NGCHW/GKCRS/NGKPQ that is already supported.

- ▶ The runtime fusion engine now supports NVIDIA Ampere Architecture cards with compute capability 86 (that is, GA10x) in addition to compute capability 80 (GA100) and compute capability 75 (Tu10x).

- ▶ The runtime fusion engine fully supports fusing configurable ReLU (a generalization of ReLU, clipped ReLU, and leaky ReLU), Tanh, Sigmoid, configurable EluGelu, configurable Softplus, and configurable Swish forward and backward activations into the epilog of a convolution forward, a convolution backward data, or a matrix multiplication operation.

- ▶ The runtime fusion engine now fully supports [N, H, W, C] to [1, 1, 1, C] reduction and [N, H, W, C] to [N, H, W, 1] reduction on the output of a convolution forward, a convolution backward data operation, and [1, M, N] to [1, M, 1] and [1, M, N] to [1, 1, N] reduction in matrix multiplication operations. For convolution backward filter operation, [N, H, W, C] to [1, H, W, C] reduction and [N, H, W, C] to> [N, 1, 1, 1]reduction are supported. The supported reduction operators are CUDNN_REDUCE_TENSOR_ADD, CUDNN_REDUCE_TENSOR_MIN, and CUDNN_REDUCE_TENSOR_MAX.

- ▶ API logging in cudnnBackendExecute() has been greatly improved to print out the internal information in descriptors like operation graphs, engines, and execution plans.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

▶ In some cases, `cudnnConvolutionBackwardData()`, on the NVIDIA Turing and NVIDIA Volta architectures performs operations on the GPU resulting in illegal memory access. This issue was fixed in version 8.0 and subsequent releases.

▶ When running a convolution forward, convolution backward data/weights, or a matrix multiplication fusion with pointwise and reduction operations with engine index `0`, the runtime fusion engine used to be allowed to run on the CUDA Toolkit 10.2. However, not all the features it relies upon are supported in the CUDA Toolkit 10.2. For better stability and targeted optimizations, the engine now requires CUDA Toolkit 11.2 update 1. We have blocked the engine from running in cuDNN built against CUDA Toolkit 10.2. See the *Limitations* section for more details.

▶ The supported check in the runtime fusion engine has been improved to return proper error codes in currently unsupported operation graphs, such as:

  ▶ an operation graph that contains more than one convolution of matrix multiplication operations

  ▶ convolutions with compute type that is not FP32

  ▶ grouped convolutions

  ▶ when the convolution mode is not `CUDNN_CROSS_CORRELATION`

▶ When running a convolution backward data/weights fusion with pointwise and reduction operations with engine index `0`, the runtime fusion engine may be launching kernel with more shared memory specified than necessary, causing sub-optimal performance. This issue has been fixed in this release.

▶ Execution of a plan for convolution forward operation graph, with engine-global index `1` returned `CUDNN_STATUS_INTERNAL_ERROR` when the filter format is NHWC and padding was larger than zero. This issue has been fixed in this release.

▶ Compared to cuDNN version 8.0.5, there was a known performance regression of 10-50% on Tacotron2 and WaveGlow models. This issue has been fixed in this release.

▶ `cudnnConvolutionBiasActivationForward()` does not invoke TF32 Tensor Core kernels when the math type in the convolution descriptor is set to `CUDNN_DEFAULT_MATH`. This leads to suboptimal performance under this math mode. This issue has been fixed in this release.

▶ Fixed an issue where the version 8 graph API's execution plan descriptor may internally refer to a descriptor outside of the data structure, which can cause unexpected errors

when the external descriptors have been destroyed. Now all the information is recorded within the data structure.

▶ There was a performance regression where NHWC was slower than NCHW on 3D convolution up to 40% on V100 and NVIDIA A100 GPUs. This issue has been fixed in this release.

▶ There was a performance regression in certain use cases comparing NVIDIA RTX 3090 using cuDNN version 8.x to NVIDIA RTX 2080 Ti using cuDNN version 7.x. This regression has been fixed in this release.

▶ Compared to cuDNN version 7.6, there were known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Volta and NVIDIA Ampere Architecture GPUs. These regressions has been fixed in this release.

▶ When calling:

  ▶ cudnnRNNForwardInference()

  ▶ cudnnRNNForwardInferenceEx()

  ▶ cudnnRNNForwardTraining()

  ▶ cudnnRNNForwardTrainingEx()

  ▶ cudnnRNNBackwardData()

  ▶ cudnnRNNBackwardDataEx()

  the API will crash when called with `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` algo but the cudnnPersistentRNNPlan_t was not created. This has been fixed in this release.

Known Issues

▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ The ResNet-50 native FP32 inference issues have been fixed on NVIDIA Volta and NVIDIA Turing. Few performance regressions exist in the NVIDIA Ampere Architecture GPUs.

▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

▶ cudnnAddTensor() does not support all broadcast-able tensor shapes even though the cuDNN documentation says otherwise.

▶ cudnnPoolingForward() with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Compared to cuDNN 8.0.0 Preview, there is a known ~12% performance regression on vgg16 when run on Jetson Nano and TX2.

▶ Compared to cuDNN 8.0.4, there is a known ~6% performance regression on ONNX-WaveGlow when run on NVIDIA TITAN RTX.

- Compared to cuDNN 7.6, there is a significant performance regression on Darknet when run on Jetson Nano.
- For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.
- Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single [cudnnConvolutionBackwardFilter()](#) use case. We are not aware of any popular model that uses this unique use case.
- Although the overall cuDNN library size has improved in cuDNN 8.1.0 with CUDA Toolkit 11.2 and greater as compared to cuDNN 8.0.x, the cuDNN library remains large; future releases will attempt to moderate the severity of this issue.
- `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT` returns an internal error when the number of channels in the filter is greater than or equal to 65536.
- Execution of a plan for convolution forward operation graph, with engine-global index `1` returns `CUDNN_STATUS_INTERNAL_ERROR` when the filter format in NHWC and padding is larger than zero.
- Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).
- Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing GPUs.
- `cudnnActivationForward()` can cause an illegal memory access CUDA error for tensors with more than 2**30 elements.
- L4T users of cuDNN may observe `CUDNN_STATUS_EXECUTION_FAILED` errors in some cases when performing convolutions using `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`. This issue is being investigated.
- Users of the static library requiring the best possible convolution performance should use whole-archive linking. This will come at a cost to the binary size that will require resolution in a future release, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.
- Compared to cuDNN 8.1.0, there are known performance regressions on certain `dgrad` NHWC configurations from FastPitch and WaveGlow models on V100 and NVIDIA A100 GPUs.
- Compared to cuDNN 7.6.5, there are known performance regressions on various convolutional models using INT8 data types on NVIDIA Volta GPUs.
- Compared to cuDNN 8.1.0, there is a known regression in performance of the runtime fusion engine for convolution fused with ReLU in the epilog. This is caused due to the generalized support for parameterized ReLU. Further optimizations are being worked on.

Limitations

▶ The runtime fusion engine was only supported in the cuDNN build based on CUDA Toolkit 11.2 update 1; it now requires the NVRTC from CUDA 11.2 update 1. If this condition is not satisfied, the error status of CUDNN_STATUS_NOT_SUPPORTED or CUDNN_STATUS_RUNTIME_PREREQUISITE_MISSING will be returned.

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the CUBLAS_WORKSPACE_CONFIG environmental variable, for example, :16:8 or :4096:2.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is :16:8:4096:2, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the :16:8 non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the _ALGO_0 algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions C, D (if 3D convolution), H, and W is not supported on devices older than NVIDIA Volta. To prevent a

potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs when cudnnConvolutionForward() is used with CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM algorithm and half I/O data types a silent error might occur when the output width Q is 1 and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (CUDNN_DATA_INT8, CUDNN_DATA_INT8x4, CUDNN_DATA_INT8x32 or CUDNN_DATA_INT32). Floating types (particularly CUDNN_DATA_FLOAT) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (CUDNN_DATA_FLOAT is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as CUDNN_DATA_FLOAT.
3. Convert all output tensors from a supported type to your desired output type.

> **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to $W >= (R-1) * dilationW \&\& H >= (S-1) * dilationH$, whereas, in cuDNN v8.0.x, $W == (R-1) * dilationW || H == (S-1) * dilationH$ cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable CUDNN_TEXOFF_DBG. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob CUDNN_KNOB_TYPE_USE_TEX to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1 is not supported when the product (channels * height * width) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns CUDNN_STATUS_NOT_SUPPORT when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionChec()) to load the kernels in the sub library before opening graph capture.

▶ Starting in cuDNN version 8.1.0, we are no longer shipping the `libfreeimg` static library with the MNIST sample. Users can follow the instructions in the `readme.txt` file to download and compile the library separately and link with the MNIST sample.

1.12.　cuDNN Release 8.1.0

This is the cuDNN 8.1.0 release notes. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the [cuDNN Archived Documentation](#).

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ A preview of the cuDNN runtime operation fusion capabilities is included in this release. This feature is exposed as a new backend engine in the version 8.0 graph API. With runtime op fusion, the engine can generate and compile fused tensor-core kernels on the fly for the specified operation graph during the execution plan finalization stage. Some of the operation graph patterns supported in this preview are: convolution or matrix multiplication operation with arbitrary combination of one or more pointwise operations, and reduction operations fused onto the output tensor. Examples include but are not limited to `conv-bias-leaky_relu`, and `gemm-bias-gelu`. This feature is supported on GPUs with compute capability 7.5 and 8.0. The current implementation supports FP16 I/O with FP32 compute or FP32 (TF32) I/O with FP32 compute. In this release, the support for this feature is restricted to Linux on x86-64. We will continue to work on this feature to provide additional support and improved performance. In the meantime, we welcome your feedback. E-mail:cudnn@nvidia.com

▶ We have released our [C++ frontend via GitHub](#) which implements a series of classes wrapping around the v8 backend C API. The user must include a few headers to enjoy the convenience from graph construction, heuristics query to execution. The frontend also implements a significantly improved autotuning feature that can accurately time the executions from a list of functionally equivalent implementations and return the fastest implementation.

▶ Heuristics have been improved for TF32 and `PSEUDO_HALF` (with Tensor Core enabled) convolutions. On one known model, performance was improved 1.3x (when not auto-tuning). On select cases in several models, we have seen performance improvements up to ~50x.

▶ Added support for `PSEUDO_BFLOAT16_CONFIG` on NVIDIA Ampere Architecture GPU for CNNs. While most of the algos/engines that are available for `PSEUDO_HALF_CONFIG` are available for `PSEUDO_BFLOAT16_CONFIG`, a few are not available. The available engines for `PSEUDO_BFLOAT16_CONFIG` can achieve at least 90% performance of

`PSEUDO_HALF_CONFIG` for layers in standard models. There is a known limitation for layers having three or four channels for the filter and convolution of stride 2, such as the first layer of ResNet.

▶ EfficientNet performances have improved. Depthwise convolution is now optimized in NHWC layout in cuDNN 8.1.0. From EfficientNet, we see an average of 2.9x speed-up for 5x5 layers, and 1.7x speed-up for 3x3 layers.

▶ Added support for TF32 engines to compute operation graphs that match the fused conv-bias-activation pattern. TF32 kernels are also supported in cudnnConvolutionBiasActivationForward() API.

▶ Added support for a new RNN algo `CUDNN_RNN_ALGO_PERSIST_STATIC_SMALL_H`, specialized for small hidden sizes. It is expected to be faster than other algos for those small hidden sizes.

▶ The cuDNN build against CUDA Toolkit 11.2 is now backward compatible with earlier CUDA 11 drivers, including 450, 455, in addition to the 460 driver.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

▶ Kernel `calc_bias_diff_nhwc_packed` has a known functional issue from 8.0.2. It happens when running `cudnnConvolutionBackwardBias`(bgrad) with `NHWC/NDHWC` packed tensors and even `C && (C >= 6)`. This issue was fixed in this release.

▶ Kernel `convolve_common_engine_int8` may cause accuracy degradation when running cudnnConvolutionBiasActivationForward() with INT8 in cuDNN version 8.0.5 because there was not a rounding when converting the results from single to INT8. This issue was fixed in this release.

▶ On some NVIDIA Turing GPUs, when users are running persistent RNN with `hiddenSize` greater than or equal to 768 but less than 1024, users may get incorrect results and see CUDA error 719, `cudaErrorLaunchFailure` the next time they call `cudaDeviceSynchronize()`. This bug has been fixed in this release.

▶ Calling cudnnConvolutionBiasActivationForward() or executing a cuDNN backend plan for fused convolution-bias-activation operation graphs, can lead to a memory leak. This issue is fixed in the current release.

▶ On Windows, calling API cudnnGetFoldedConvBackwardDataDescriptors() results in failure to find symbols. This issue has been present in all versions since cuDNN version 8.0.0 and is fixed in this release.

▶ The backend convolution operation had external dependencies on the user created backend tensor descriptors even after finalization. Deletion of the tensor descriptors might

cause the operation to seg-fault when constructing the operation graph. This bug affects all versions since cuDNN version 8.0.0, and has been fixed in 8.1.0.

▶ Many convolution models were experiencing lower performance on NVIDIA RTX 3090 compared to 2080 Ti. This included ResNet-50 with up to 2x performance difference, ResNeXt up to 10x performance difference and U-Net up to 3x performance difference. The performance issues have been fixed in this release.

Known Issues

▶ Users of the static library requiring best possible convolution performance should use whole-archive linking. This will come at a cost to binary size that will require resolution in future releases, either through static sub libraries or relaxing the whole-archive linkage requirement altogether.

▶ The ResNet-50 native FP32 inference issues have been fixed on NVIDIA Volta and NVIDIA Turing. Few performance regressions exist in the NVIDIA Ampere Architecture GPU.

▶ Compared to version 8.0.5, legacy convolution APIs have increased CPU computational costs. On x86, this has been measured to be as high as 10 microseconds.

▶ cudnnAddTensor() does not support all broadcast-able tensor shapes even though the cuDNN documentation says otherwise.

▶ Users have reported that in RNN training with non-zero dropout rate, and if the RNN network is unidirectional, the output of cudnnRNNBackwardWeights() may be non-deterministic. We are still investigating this issue.

▶ cudnnPoolingForward() with pooling mode CUDNN_POOLING_AVG might output NaN for pixel in output tensor outside the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim().

▶ Compared to cuDNN 8.0.0 Preview, there is a known ~12% performance regression on vgg16 when run on Nano and TX2.

▶ Compared to cuDNN 8.0.4, there is a known ~6% performance regression on ONNX-WaveGlow when run on NVIDIA TITAN RTX.

▶ Compared to cuDNN 7.6, there is a significant performance regression on Darknet when run on Nano.

▶ For pre-Volta devices, users should align all buffers to at least 4 bytes; this applies to half-precision data as well.

▶ Compared to cuDNN version 8.0.2, there is a known 3x performance regression for a single cudnnConvolutionBackwardFilter() use case. We are not aware of any popular model that uses this unique use case.

▶ Although the overall cuDNN library size has improved in cuDNN 8.1.0 with CUDA Toolkit 11.2 and greater as compared to cuDNN 8.0.x, the cuDNN library remains large; future releases will attempt to moderate the severity of this issue.

▶ CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT returns an internal error when the number of channels in the filter is greater than or equal to 65536.

▶ Execution of a plan for convolution forward operation graph, with engine-global index `1` returns `CUDNN_STATUS_INTERNAL_ERROR` when the filter format in NHWC and padding is larger than zero.

▶ Convolutions (`ConvolutionForward`, `ConvolutionBackwardData`, and `ConvolutionBackwardFilter`) may experience performance regressions when run with math type `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` on `CUDNN_DATA_FLOAT` data (*input* and *output*).

▶ Compared to cuDNN 7.6, there are known performance regressions up to 2x on select configurations for AlexNet-like models on NVIDIA Turing, NVIDIA Volta, and NVIDIA Ampere Architecture GPUs.

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multistream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are

cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is 1 and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, cudnnTransformTensor() can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 📝 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ cudnnSpatialTfSamplerBackward() returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, cudnnOpsInferVersionChec()) to load the kernels in the sub library before opening graph capture.

▶ Starting in cuDNN version 8.1.0, we are no longer shipping the `libfreeimg` static library with the MNIST sample. Users can follow the instructions in the `readme.txt` file to download and compile the library separately and link with the MNIST sample.

1.13. cuDNN Release 8.0.5

This is the cuDNN 8.0.5 release notes. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ RNN now supports zero-length sequences within the batch when the RNN data layout is `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED`. For more information, see cudnnSetRNNDataDescriptor().

▶ Users can now set the environment variable `CUDNN_CONV_WSCAP_DBG` to a value in MiB to limit the workspace size returned by cudnnConvolutionForwardGetWorkspaceSize(), cudnnConvolutionBackwardDataGetWorkspaceSize(), and cudnnConvolutionBackwardFilterGetWorkspaceSize(). Limiting the workspace might result in performance lost.

▶ Significant performance improvements were made for NVIDIA RTX 3090 for many models on many configurations.

▶ Performance improvements were made:

  ▶ For EfficientNet, when run using NHWC FP16 Tenor Core configurations on V100 and A100 GPU architectures.

  ▶ For PilotNet, AH-Net, MobileNet V3 on V100 and A100 GPU architectures.

  ▶ For various 3D convolution cases on NVIDIA RTX 8000.

▶ Support for the 3D `NDHWC` layout was added in `cudnnConvolutionBackwardFilter()`.

▶ Added instructions for installing cuDNN using the Package Manager for Linux and RHEL users. For step-by-step instructions, see Package Manager Installation in the *cuDNN Installation Guide*.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Fixed Issues

The following issues have been fixed in this release:

- `cudnnBackendFinalize(descriptor)`, where `descriptor` is of type `CUDNN_BACKEND_ENGINE_DESCRIPTOR()` or `CUDNN_BACKEND_EXECUTION_PLAN_DESCRIPTOR()`, might result in a hang if the operation graph has backward filter operation and the user links against `libcudnn.so` (`cudnn64.dll` on Windows). This issue has been fixed in this release.

- Call to `cudnnConvolutionBiasActivationForward()` might result in a memory leak in release 8.0.1. This issue has been fixed.

- Performance regression on the U-Net Industrial network on NVIDIA Volta for certain batch sizes has been fixed.

- `cudnnRNN*()` with LSTM mode may produce incorrect results on the `cy` outputs when clipping is enabled on all GPUs. This issue also exists in previous cuDNN releases since version 7.2.1. This issue has been fixed in this release.

- `cudnnRNNForward*` with LSTM mode may produce incorrect results in case of clipping when `CUDNN_RNN_ALGO_PERSIST_STATIC` is used. This issue also exists in previous cuDNN releases since version 7.2.1. This issue has been fixed in this release.

- In previous cuDNN versions, `cudnnRNNBackwardData()` or `cudnnRNNBackwardDataEx()` may produce non-deterministic outputs when running configurations such as `hiddenSize=128` or less, LSTM cell type, and FP32 with `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`. This issue has been fixed in this release.

- Compared to cuDNN 7.6, there was a known ~6% performance regression on Inception V3 and ResNet-50 models when run using NHWC FP16 configurations on various NVIDIA Turing and NVIDIA TITAN V architectures. This issue has been fixed in this release.

- Compared to cuDNN v8.0.3, there was a known ~18% performance regression on the U-Net Industrial model when run using NCHW TF32 configurations on V100 and A100 GPU architectures. This issue has been fixed in this release.

- *Updated: November 25, 2020*

  When calling `cudnnConvolutionBiasActivationForward()` with INT8x4 or INT8x32 I/O tensors, it could result in `CUDNN_STATUS_BAD_PARAM` in 8.0.4. This issue has been fixed in this release.

Known Issues

- When using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) not being multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users encountered a

return status of `CUDNN_STATUS_EXECUTION_FAILED` in cudnn built against CUDA 11.0. This issue has been fixed with cuDNN built against CUDA 11.1.

▶ The ResNet-50 native FP32 inference issues have been fixed on NVIDIA Volta and NVIDIA Turing. Few performance regressions exist in the NVIDIA Ampere Architecture GPU.

▶ [cudnnAddTensor()](#) does not support all broadcast-able tensor shapes even though the cuDNN documentation says otherwise.

▶ Users have reported that in RNN training with non-zero dropout rate, and if the RNN network is unidirectional, the output of `cudnnRNNBackwardWeights()` may be non-deterministic. We are still investigating this issue.

▶ `cudnnPoolingForward()` with pooling mode `CUDNN_POOLING_AVG` might output NaN for pixel in output tensor outside the value recommended by `cudnnGetPoolingNdForwardOutputDim()` or `cudnnGetPooling2dForwardOutputDim()`.

▶ Compared to cuDNN 8.0.0 Preview, there is a known ~12% performance regression on vgg16 when run on Nano and TX2.

▶ Compared to cuDNN 8.0.4, there is a known ~6% performance regression on ONNX-WaveGlow when run on NVIDIA TITAN RTX.

▶ Compared to cuDNN 7.6, there is a significant performance regression on Darknet when run on Nano.

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the [Results Reproducibility](#) section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 16 byte boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.x compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.x users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, `cudnnTransformTensor()` can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 🗩 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures.

Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

▶ `cudnnSpatialTfSamplerBackward()` returns `CUDNN_STATUS_NOT_SUPPORT` when the number of channels exceeds 1024.

▶ When using graph-capture, users should call the sub library version check API (for example, `cudnnOpsInferVersionChec()`) to load the kernels in the sub library before opening graph capture.

1.14. cuDNN Release 8.0.4

This is the cuDNN 8.0.4 release notes. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the [cuDNN Archived Documentation](#).

Key Features and Enhancements

The following features and enhancements have been added to this release:

GA102 support with improved convolution performance
Now includes convolution heuristics targeting the NVIDIA GA102 GPU. *(not applicable for Jetson platforms)*

RNN API v8 sample
The new RNN sample illustrating the usage of the new RNN version 8 API has been added. The sample's workflow consists of the several routines to create RNN descriptors, create RNN data descriptors, set up weight space, and compute routines. The sample takes several input parameters that can set up different RNN configurations and input data specifications (data type, cell mode, bias mode, and so on).

RNN functional and performance improvements

▶ When using RNN with padded data, the RNN padding and padding removal kernel at the beginning and at end of the computation may achieve up to 4x speedup, and has been generalized to work with any batch sizes.

▶ Updated the following API functions return codes:

▶ [cudnnBuildRNNDynamic()](#)
▶ [cudnnCreatePersistentRNNPlan()](#)

▶ [cudnnRNNBackwardData_v8()](#)

▶ [cudnnRNNBackwardData()](#)

▶ [cudnnRNNBackwardDataEx()](#)

▶ [cudnnSetDropoutDescriptor()](#)

ARM Server Base System Architecture (SBSA)
Added support for ARM SBSA for Linux.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the [cuDNN Support Matrix for 8.x.x](#).

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 8.0.4 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the [Results Reproducibility](#) section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN 8.0.4 now require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 128-bit boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.4 compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.4 users can target the backend API to query the

numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when [cudnnConvolutionForward()](#) is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, `cudnnTransformTensor()` can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

  1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
  2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
  3. Convert all output tensors from a supported type to your desired output type.

> 📝 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

Deprecated Features

The following features are deprecated in cuDNN 8.0.4:

▶ Support for Ubuntu 18.04 ppc64le builds will be dropped post cuDNN 8.0.4.

Fixed Issues

▶ `cudnnConvolutionBackwardFilter()` and `cudnnGetConvolutionBackwardFilterWorkspaceSize()` can result in a segmentation fault in multi-threaded usage due to a race condition. This issue has been fixed in this release.

▶ The `libfreeimage.a` library in the RHEL 8 ppc64le RPM package was for the wrong architecture. This issue has been fixed in this release.

▶ In previous cuDNN versions, `cudnnRNNBackwardData()` or `cudnnRNNBackwardDataEx()` may return `CUDNN_STATUS_INTERNAL_ERROR`, NaN-s, or non-deterministic finite values when `CUDNN_RNN_ALGO_PERSIST_STATIC` was selected. These issues occurred mainly on smaller GPUs, such as NVIDIA Turing with 30 or 36 SMs and smaller `hiddenSize` values. Most of those issues have been fixed in this release. However, configurations such as `hiddenSize=128`, LSTM, FP32 with `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` may still output non-deterministic results.

▶ There was an issue in upgrading the cuDNN version using the RPM and Debian packages in the 8.0.3 version. This issue has been fixed in this release.

▶ The ResNet-50 native FP32 inference issues have been fixed on NVIDIA Volta and NVIDIA Turing. Few performance regressions exist in the NVIDIA Ampere Architecture GPU.

▶ cuDNN exhibited performance regressions for GoogLeNet and U-Net on V100. This issue has been fixed in this release.

▶ cuDNN exhibited performance regressions for VGG16 on GA100. This issue has been fixed in this release.

▶ The performance regression across Tacotron2 and WaveGlow seen on the NVIDIA Turing architecture have been fixed.

▶ The performance regressions in the FastPitch network seen on the NVIDIA Volta and NVIDIA Turing architecture have been fixed.

▶ The cuDNN API unconditionally triggers CUDA context initialization. This causes unnecessary host-side performance overhead. This is an issue that was introduced in cuDNN version 8.0.2. This issue has been fixed in this release.

▶ Some ResNet-50 and SSD mixed precision inference use-cases may have performance regressions compared to cuDNN 7.6 on V100. V-Net 3D models might have performance regressions on NVIDIA Turing based architectures. This issue has been fixed in this release.

▶ Previous cuDNN 8 releases exhibited performance regressions when compared to version 7.6, for some important convolutional networks on the NVIDIA Pascal GPU architecture. In

particular, the performance regressions of ResNet-50 seen previously on NVIDIA Pascal with cuDNN versions 8.0.3 and earlier, are fixed with this release.

▸ `cudnnConvolutionBiasActivationForward()` could result in incorrect results when the `alpha2` value is zero and the device buffer `zData` contains NaN. This issue has been fixed in this release.

▸ When using `cudnnRNN*Ex()` APIs, if the layout of RNN data is `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED`, and if the batch size is larger than 6144 on NVIDIA Volta or NVIDIA Ampere Architecture A100 GPUs, or larger than 4096 on NVIDIA Turing GPUs, `CUDNN_STATUS_EXECUTION_FAILED` would be returned. This issue has been fixed in this release. cuDNN supports arbitrary batch size.

▸ When the user upgraded from cuDNN 8.0.2 to 8.0.3 through the Debian or RPM package, users had to manually uninstall the old `libcudnn8-doc` package before they installed `libcudnn8-samples_*.deb/rpm`, otherwise a file conflict could happen. This has been fixed and is no longer the case in the 8.0.4 release.

▸ Performance regressions on NVIDIA Turing, NVIDIA Volta, and NVIDIA Pascal architectures for True Half convolutions have been resolved.

▸ When using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) not being multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users encountered a return status of `CUDNN_STATUS_EXECUTION_FAILED in cudnn built against cuda 11.0`. This issue has been fixed with cuDNN built against CUDA 11.1.

Known Issues

▸ When using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) not being multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users encountered a return status of `CUDNN_STATUS_EXECUTION_FAILED`. This issue affects earlier cuDNN 8.0.1 Preview and cuDNN 8.0.2 releases built against CUDA 11.0.

▸ There is a known minor performance regression on small batch sizes for ResNet-50 native FP32 inference that exists on the NVIDIA Ampere Architecture GPU.

1.15.　 cuDNN Release 8.0.3

This is the cuDNN 8.0.3 release notes. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

cuDNN backend API

Documentation for the cuDNN backend API has been included in this release. Users specify the computational case, set up an execution plan for it, and execute the computation using numerous descriptors. The typical use pattern for a descriptor with attributes consists of the following sequence of API calls:

1. cudnnBackendCreateDescriptor() creates a descriptor of a specified type.
2. cudnnBackendSetAttribute() sets the values of a settable attribute for the descriptor. All required attributes must be set before the next step.
3. cudnnBackendFinalize() finalizes the descriptor.
4. cudnnBackendGetAttribute() gets the values of an attribute from a finalized descriptor.

For more information, refer to the cuDNN Backend API section in the *cuDNN API Reference*.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 8.0.3 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the CUBLAS_WORKSPACE_CONFIG environmental variable, for example, :16:8 or :4096:2.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is :16:8:4096:2, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the :16:8 non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN 8.0.3 now require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 128-bit boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.3 compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.3 users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions C, D (if 3D convolution), H, and W is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width Q is 1 and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, `cudnnTransformTensor()` can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 💬 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to W >= (R-1) * dilationW && H >= (S-1) * dilationH, whereas, in cuDNN v8.0.x, W == (R-1) * dilationW || H == (S-1) * dilationH cases are no longer supported.

▶ In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures.

Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to `1` for engines that support texture-based load instructions.

▶ In the backend API, convolution forward engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX=1` is not supported when the product (`channels * height * width`) of the input image exceeds 536,870,912 that is 2^29.

Fixed Issues

▶ For cudnnConvolutionBackwardFilter, the 3D convolution table, `wDesc: _NCHW`, `_ALGO_1` and `FFT_TILING` had incorrect data fields. This has been fixed in this release.

▶ In prior versions of cuDNN, cudnnPoolingForward() with pooling mode `CUDNN_POOLING_MAX` might return incorrect result when one of the spatial dimensions has negative padding and the output tensor is larger than the value recommended by cudnnGetPoolingNdForwardOutputDim() or cudnnGetPooling2dForwardOutputDim(). This issue has been fixed in this release.

▶ In cudnnPoolingForward() with average-pooling, when the output tensor data is INT8 type, it is possible for some pixels result to be off by 1. Note that `cudnnPoolingForward()` rounds to the nearest-even integer. This issue has been fixed in this release.

▶ The performance of cudnnConvolutionBiasActivationForward() for INT8x4 use cases on NVIDIA Volta and NVIDIA Turing, INT8x32 use cases on NVIDIA Turing, FP32, and pseudo-FP16 use cases on NVIDIA Volta, NVIDIA Turing, and NVIDIA Ampere Architecture GPU have been improved.

▶ We have updated our public headers to fully reflect the documented dependencies between the six sub libraries.

▶ There were `libcudnn_ops/cnn/adv_infer/train_static.a` binaries in the cuDNN Debian and tgz packages. Users were advised not to link against those and link against `libcudnn_static.a` instead. Those binaries have been removed from the release packages.

▶ On NVIDIA Volta and NVIDIA Pascal architectures, performance regressions were present for various `TRUE_HALF` convolutions. This has been fixed in this release.

▶ In prior versions of cuDNN, API functions `cudnnGetConvolution*Algorithm_v7()` return a workspace size in the result for `algo1` that is inconsistent with the result of the corresponding `cudnnGet*Workspace()` calls if the math type of the convolution descriptor is set to `CUDNN_FMA_MATH`. This issue has been fixed in this release.

▶ The new RNN APIs: `cudnnRNNForward()`, `cudnnRNNBackwardData_v8()`, and `cudnnRNNBackwardWeights_v8()` were available as a preview in the cuDNN 8.0.2 release. They no longer hold preview status.

▶ When using `cudnnRNN*Ex()` APIs, if the user planned to use `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or

`CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED` as the layout of the RNN data descriptors, the user would have had to call `cudnnSetRNNPaddingMode()` to set the mode to `CUDNN_RNN_PADDED_IO_ENABLED` after initializing an `RNNDescriptor` but before calling `cudnnGetRNNWorkspaceSize()`. Not doing this would result in `CUDNN_STATUS_EXECUTION_FAILED`. We have added internal checks to return `CUDNN_STATUS_BAD_PARAM` to prevent hitting `EXECUTION_FAILED`.

▶ When cudnnBatchNormalizationForwardTrainingEx() is called with NHWC tensors with pseudo-half configuration, under rare occasions the kernel would produce incorrect results, including possible NaNs in the results. This has been fixed in this release. This issue affects earlier releases since 7.4.1.

▶ Fused convolution-scale-bias-activation with per-channel α1 and α2 scaling gives incorrect results when the reorder type in the convolution descriptor is set to `CUDNN_NO_REORDER`. This is an issue in cuDNN version 8.0.2 This issue has been fixed in this release.

▶ On NVIDIA Ampere Architecture GA100, cudnnConvolutionBackwardData() for Tensor Core enabled problems with half input and output could, in rare cases, could produce incorrect results; the same could happen for users of cudnnBackendExecute() using engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` 57 for backward data. This has been fixed in this release. *(not applicable for Jetson platforms)*

▶ There was a performance regression in MaskRCNN inference with automatic mixed precision on V100. This has been fixed in this release.

▶ Two-dimensional forward convolutions using algo1 may segfault when the filter size is large. For example, we have observed this issue when the filter width and height are more than or equal to 363. This has been fixed in this release.

▶ For some 3D spatial non-Tensor-Core convolutions on Maxwell, NVIDIA Pascal, NVIDIA Volta, and NVIDIA Turing architectures, cudnnBackwardFilter() can return incorrect results when the convolution width padding exceeds the value (`filterWidth – 1)/2`. Likewise, users of cudnnBackendExecute() can experience the same issue when using the engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` 32 for backward filter. The issue affecting cudnnBackwardFilter() has been fixed in this release. With cudnnBackendFinalize(), an engine descriptor with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` 32 and a backward filter operation that satisfies the previous condition will return `CUDNN_STATUS_NOT_SUPPORTED`.

Known Issues

▶ Occasionally, inaccurate results were observed in outputs of the `cudnnRNNBackwardWeights()` and `cudnnRNNBackwardWeightsEx()` functions when the RNN cell type was GRU and the NVIDIA Ampere Architecture GPU was used with FP32 I/O and `mathType` of `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH`. Users may switch to `CUDNN_FMA_MATH` as a temporary workaround. This issue is being investigated.

▶ `cudnnRNN*()` with LSTM mode may produce inaccurate results on the `cy` outputs when clipping is enabled on all GPUs. This issue exists in previous cuDNN releases as well.

- On NVIDIA Volta and NVIDIA Pascal architectures, performance regressions may be present for various `TRUE_HALF` convolutions.

- When the user is using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) being not multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users may encounter a return status of `CUDNN_STATUS_EXECUTION_FAILED`. This issue also affects earlier releases cuDNN 8.0.1 Preview and cuDNN 8.0.2.

- Some ResNet-50 and SSD mixed precision inference use-cases may have performance regressions compared to cuDNN 7.6 on V100. V-Net 3D models might have performance regressions on NVIDIA Turing based architectures.

- When using `cudnnRNN*Ex()` APIs, if the user used `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED` as the layout of the RNN data descriptors, and if the batch size is larger than 6144 on NVIDIA Volta or NVIDIA Ampere Architecture A100 GPUs, or larger than 4096 on NVIDIA Turing GPUs, `CUDNN_STATUS_EXECUTION_FAILED` would be returned.

- Documentation of the backend API is not complete. The `CUDNN_BACKEND_OPERATION_GEN_STATS_DESCRIPTOR` and `CUDNN_BACKEND_OPERATION_POINTWISE_DESCRIPTOR` descriptor types will be documented in a future release.

- The `conv_sample_v8.0` sample is not included in the Debian and RPM packages. This will be fixed in a future release.

- The `libfreeimage.a` library in the RHEL 8 ppc64le RPM is for the wrong architecture. This will be fixed in a future release.

- When the user is upgrading from cuDNN 8.0.2 to 8.0.3 through the Debian or RPM package, before installing `libcudnn8-samples_*.deb/rpm`, users should manually uninstall the old `libcudnn8-doc` package, otherwise a file conflict may happen.

1.16.   cuDNN Release 8.0.2

This is the cuDNN 8.0.2 release notes and first GA release of cuDNN 8.x. This release includes fixes from the previous cuDNN v8.0.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the [cuDNN Archived Documentation](#).

Key Features and Enhancements

cuDNN 8.0.1 Preview and 8.0.0 Preview

The key features mentioned in cuDNN [8.0.1 Preview](#) and [8.0.0 Preview](#) are now GA quality in this release.

Added new API functions to the documentation

`cudnnRNNBackwardData_v8()` and `cudnnRNNBackwardWeights_v8()` are now documented in the cudnn_adv_train.so Library. For a list of functions and data types that were added in this release, see API Changes For cuDNN 8.0.2.

TF32 performance

▶ TF32 for 3D convolutions and deconvolution performance is significantly better, up to 3.9x, compared to cuDNN 8.0.1.

▶ TF32 for grouped convolutions on A100 were improved up to 1.5x performance compared to cuDNN 8.0.1 on ResNext convolution layers and up to 3x the performance compared to V100 with cuDNN v7.6. *(not applicable for Jetson platforms)*

The above performance improvements were measured using only cuDNN operations. The observed performance improvements will depend on a number of factors, such as non-cuDNN operations, kernel run time, and model architecture type.

Performance improvements

This release includes performance improvements on all architectures for 2D and 3D grouped convolutions compared with version 7.6. Additionally, we improved kernel selection heuristics on several known Deep Learning GitHub Examples (also known as model scripts).

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.x.x.

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 8.0.2 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer

configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN 8.0.2 now require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 128-bit boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.2 compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.2 users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in NVIDIA Volta and above, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, `cudnnTransformTensor()` can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 🗨 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

- In INT8x32 Tensor Core cases, the parameters supported by cuDNN v7.6 are limited to `W >= (R-1) * dilationW && H >= (S-1) * dilationH`, whereas, in cuDNN v8.0.x, `W == (R-1) * dilationW || H == (S-1) * dilationH` cases are no longer supported.

- In prior versions of cuDNN, some convolution algorithms can use texture-based load structure for performance improvements particularly in older hardware architectures. Users can opt out of using texture using the environmental variable `CUDNN_TEXOFF_DBG`. In cuDNN 8.x, this variable is removed. Texture loading is turned off by default. Users who want to continue to use texture-based load, can adapt the new backend API, and toggle the engine knob `CUDNN_KNOB_TYPE_USE_TEX` to 1 for engines that support texture-based load instructions.

Fixed Issues

The following issues have been fixed in this release:

- The implementation of `cuDNNLRNCrossChannelBackward()` for even-sized normalization windows was incorrect in all previous releases. This issue has been fixed in this release.

- There is not a dedicated API to query the supported or the most performant algo for `cudnnConvolutionBiasActivationForward()` in cuDNN. It is not recommended to query w using `cudnnGetConvolutionForwardAlgorithm_v7`. Instead, we recommend using the cuDNN version 8 backend API. The number of supported engines can be queried using enum `CUDNN_ATTR_OPERATIONGRAPH_ENGINE_GLOBAL_COUNT` from an operation graph descriptor using `cudnnBackendGetAttribute()`.

- A `memcheck` error may have occurred on cuDNN version 7.x builds when calling `cudnnConvolutionBackwardFilter()` on NVIDIA Volta or NVIDIA Turing GPUs. This issue has been fixed in this release.

- Various convolutions that exhibited sub-optimal performance on GA100 GPUs are now achieving ideal performance. *(not applicable for Jetson platforms)*

- `cudnnCnnTrainVersionCheck()` and `cudnnCnnInferVersionCheck()` were missing in past releases. This issue has been fixed in this release.

- Documentation of RNN new APIs and deprecations is not complete. The `cudnnRNNBackwardData_v8()` and `cudnnRNNBackwardWeights_v8()` have been added to this release.

- cuDNN 8.0.1 built with Windows and CUDA 11.0 RC had reduced performance on 2D, 3D, and grouped convolutions compared to Linux. This issue has been fixed in this release. *(not applicable for Jetson platforms)*

- There was a known issue in cuDNN 8.0.1 when linking statically to cuDNN and using the library's 3D algo1 backward filter convolutions. Users would see that the library emits an internal error or incorrectly state that a shared library was missing. This issue has been fixed in this release.

- When using an RPM file on RedHat for installation, upgrading from cuDNN v7 to cuDNN v8 directly or indirectly using TensorRT 7.1.3 would cause installation errors. This issue has been fixed in this release.

▶ The implementation of `cuDNNLRNCrossChannelBackward` was inconsistent with the implementation of `cuDNNLRNCrossChannelForward` and returned incorrect results when the normalization window was even. This issue has been fixed in this release.

▶ RNN APIs in cuDNN v8.0.1, compiled with CUDA 11.0, used an incorrect default down-conversion on GPUs with CUDA SM version SM80 (NVIDIA Ampere Architecture GPU family) when supplied input data and weights have the `CUDNN_DATA_FLOAT` type and `cudnnMathType_t` set using `cudnnSetRNNMatrixMathType()` is `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH`. Instead of using the default TF32 computation when Tensor Cores are used, a down conversion to FP16 (half-precision) was performed; same as in the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` mode. This introduced a lower dynamic range of intermediate data but possibly faster execution. To disable the automatic down conversion of `CUDNN_DATA_FLOAT` weights and data in RNN APIs, the user needed to set the environmental variable `NVIDIA_TF32_OVERRIDE` to `0` (notice this would have disabled the use of TF32 in the entire library, which might have had a performance impact on CNNs that are not affected by this issue). Another workaround was to assign the `CUDNN_FMA_MATH` mode to the `cudnnMathType_t` argument in `cudnnSetRNNMatrixMathType()`. Due to this, the A100 GPU TF32 feature was not accessible for RNNs in cuDNN v8.0.1. This issue has been fixed in this release. *(not applicable for Jetson platforms)*

▶ cuDNN convolution APIs may return `CUDNN_STATUS_EXECUTION_FAILED` when the number of input or output channels equals to or exceeds 2097152. This issue exists for all cuDNN 8.0.x releases. This issue has been fixed in this release.

▶ Since version 8.0.0 Preview, `cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()` erroneously returned `CUDNN_STATUS_INTERNAL_ERROR` when the workspace size argument value was less than the required workspace size as returned by their respective `cudnnGetWorkspace()` API. This issue has been fixed and `CUDNN_STATUS_BAD_PARAMS` is returned as documented.

Known Issues

▶ In this release, the performance of [cudnnConvolutionBiasActivationForward()](#) for true-half use cases on NVIDIA Pascal, INT8x4 use cases on NVIDIA Volta, and NVIDIA Turing, compared to version 7.6 is still lower. In addition, FP32 and pseudo-FP16 performance on NVIDIA Volta, NVIDIA Turing, and the NVIDIA Ampere Architecture GPU is still not fully optimized.

▶ The new RNN APIs: `cudnnRNNForward()`, `cudnnRNNBackwardData_v8()`, and `cudnnRNNBackwardWeights_v8()` are available as a preview in the cuDNN 8.0.2 release.

▶ Occasionally, inaccurate results were observed in outputs of the `cudnnRNNBackwardWeights()` and `cudnnRNNBackwardWeightsEx()` functions when the RNN cell type was GRU and the NVIDIA Ampere Architecture GPU was used with FP32 I/O and `mathType` of `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH`. Users may switch to `CUDNN_FMA_MATH` as a temporary workaround. This issue is being investigated.

▶ `cudnnRNN*()` with LSTM mode may produce inaccurate results on the `cy` outputs when clipping is enabled on all GPUs. This issue exists in previous cuDNN releases as well.

▶ On NVIDIA Volta and NVIDIA Pascal architectures, performance regressions may be present for `TRUE_HALF` convolution backward filter.

▶ When using `cudnnRNN*Ex()` APIs, if the user uses `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED` as the layout of the RNN data descriptors, and if the batch size is larger than 6144 on NVIDIA Volta or NVIDIA Ampere Architecture A100 GPUs, or larger than 4096 on NVIDIA Turing GPUs, `CUDNN_STATUS_EXECUTION_FAILED` may be returned.

▶ Currently, there are `libcudnn_ops/cnn/adv_infer/train_static.a` binaries in the cuDNN Debian and tgz packages. Users are advised not to link against those and link against `libcudnn_static.a` instead. Those binaries will be removed from the release packages in the next release.

▶ When using `cudnnRNN*Ex()` APIs, if the user plans to use `CUDNN_RNN_DATA_LAYOUT_SEQ_MAJOR_UNPACKED` or `CUDNN_RNN_DATA_LAYOUT_BATCH_MAJOR_UNPACKED` as the layout of the RNN data descriptors, the user should call `cudnnSetRNNPaddingMode()` to set the mode to `CUDNN_RNN_PADDED_IO_ENABLED` after initializing an `RNNDescriptor` but before calling `cudnnGetRNNWorkspaceSize()`. Not doing this may result in `CUDNN_STATUS_EXECUTION_FAILED`.

▶ *Updated: August 24, 2020*

Fused convolution-scale-bias-activation with per-channel α1 and α2 scaling gives incorrect results when the reorder type in the convolution descriptor is set to `CUDNN_NO_REORDER`.

▶ *Updated: August 24, 2020*

When the user is using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) being not multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users may encounter a return status of `CUDNN_STATUS_EXECUTION_FAILED`.

▶ *Updated: August 24, 2020*

For some 3D spatial non-Tensor-Core convolutions on Maxwell, NVIDIA Pascal, NVIDIA Volta, and NVIDIA Turing architectures, cudnnBackwardFilter() can return incorrect results when the convolution width padding exceeds the value (`filterWidth – 1)/2`. Likewise, users of cudnnBackendExecute() can experience the same issue when using the engine with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` 32 for backward filter. The issue affecting cudnnBackwardFilter() has been fixed in this release. With cudnnBackendFinalize(), an engine descriptor with `CUDNN_ATTR_ENGINE_GLOBAL_INDEX` 32 and a backward filter operation that satisfies the above condition will return `CUDNN_STATUS_NOT_SUPPORTED`.

1.17.   cuDNN Release 8.0.1 Preview

> **!** **ATTENTION:** This is the cuDNN 8.0.1 Preview release. This Preview release is for early testing and feedback, therefore, for production use of cuDNN, continue to use cuDNN 7.6.5. This release is subject to change based on ongoing performance tuning and functional testing. For feedback on the new backend API and deprecations, e-mail cudnn@nvidia.com.

These release notes are applicable to NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

▶ Added new kernels to improve the performance of fusion.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.0.1.

Limitations

▶ Samples can crash unless they are installed in a writable location.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 8.0.1 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ Some data types are not widely supported by all cuDNN API. For example, `CUDNN_DATA_INT8x4` is not supported by many functions. In such cases, support is available by using cudnnTransformTensor() to transform the tensors from the desired type to a type supported by the API. For example, a user is able to transform input tensors from `CUDNN_DATA_INT8x4` to `CUDNN_DATA_INT8`, run the desired API and then transform output tensors from `CUDNN_DATA_INT8` to `CUDNN_DATA_INT8x4`. Note that this transformation will incur an extra round trip to memory.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN 8.0.1 now require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 128-bit boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.1 compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.1 users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

▶ For the `_ALGO_0` algorithm of convolution backward data and backward filter, grouped convolution with groups larger than 1 and with odd product of dimensions `C`, `D` (if 3D convolution), `H`, and `W` is not supported on devices older than NVIDIA Volta. To prevent a potential illegal memory access by an instruction that only has a 16-bit version in Volta and later, pad at least one of the dimensions to an even value.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur when the output width `Q` is `1` and both height and width padding are zero.

Fixed Issues

The following issues have been fixed in this release:

▶ The `dimA` and `strideA` parameters in cudnnSetTensorNdDescriptor() do not document the tensor layout. The documentation has been updated to include this information.

▶ cuDNN 8.0.0 Preview will not work with GA10x NVIDIA Ampere Architecture GPUs. This has been fixed in 8.0.1 Preview.

▶ cuDNN 8.0.0 Preview removed a restriction on convolution backward filter for output filter with odd products of dimensions (`N*C*D*H*W`) for a kernel in `algo0` for pre-Volta GPUs. This can potentially lead to an illegal memory access error. This restriction is restored in cuDNN 8.0.1 Preview. cuDNN will use a kernel that does not have this restriction for this computation case.

▶ Fixed performance issues for pre-Vola architectures for convolutions (except when the compute type is half).

▶ Mitigated the performance regression to less than 10% end to end.

Known Issues

▶ On pre-Volta, there are significant performance issues on convolution layers when the compute type is half.

▶ Sub-optimal performance is present in this release for all INT8 convolutions for all GPUs.

▶ The performance of cudnnConvolutionBiasActivationForward() is slower than v7.6 in most cases. This is being actively worked on and performance optimizations will be available in the upcoming releases.

▶ There are some peer-to-peer documentation links that are broken within the cuDNN API Reference. These links will be fixed in the next release.

▶ `cudnnCnnTrainVersionCheck()` and `cudnnCnnInferVersionCheck()` are missing in this release and will be added in the GA release.

▶ Documentation of RNN new APIs and deprecations is not complete. The `cudnnRNNBackwardData_v8()` and `cudnnRNNBackwardWeights_v8()` functions will be implemented in the next release.

▶ cuDNN 8.0.1 Preview build with Windows and CUDA 11.0 RC has reduced performance on 2D, 3D, and grouped convolutions compared to Linux.

▶ There is a known issue in cuDNN 8.0.1 when linking statically to cuDNN and using the library's 3D algo1 backward filter convolutions. Users will see that the libraries emit an internal error or incorrectly state that a shared library is missing. This is a bug that will be fixed in a future release.

▶ When using an RPM file on RedHat for installation, installing cuDNN v8 directly or using TensorRT 7.1.3 will enable users to build their application with cuDNN v8. However, in order for the user to compile an application with cuDNN v7 after cuDNN v8 is installed, the user must perform the following steps:

 1. Issue `sudo mv /usr/include/cudnn.h /usr/include/cudnn_v8.h`.

 2. Issue `sudo ln -s /etc/alternatives/libcudnn /usr/include/cudnn.h`.

 3. Switch to cuDNN v7 by issuing `sudo update-alternatives --config libcudnn` and choose cuDNN v7 from the list.

 Steps 1 and 2 are required for the user to be able to switch between v7 and v8 installations. After steps 1 and 2 are performed once, step 3 can be used repeatedly and the user can choose the appropriate cuDNN version to work with. For more information, refer to the Installing From An RPM File and Upgrading From v7 To v8 sections in the *cuDNN Installation Guide*.

▶ When FFT Tiled aglo (that is, `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` in forward convolution or `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING` for backward data) is used for 3D convolution, an intermittent silent failure might happen due to an incorrect

stream used for kernel execution. In some cases, this might be manifested as undefined values seen in the output.

▶ The implementation of `cuDNNLRNCrossChannelBackward` is inconsistent with the implementation of `cuDNNLRNCrossChannelForward` and returns incorrect results when the normalization window is even. This will be fixed in a future release.

▶ RNN APIs in cuDNN v8.0.1, compiled with CUDA 11.0, use an incorrect default down-conversion on GPUs with CUDA SM version SM80 (NVIDIA Ampere Architecture GPU family) when supplied input data and weights have the `CUDNN_DATA_FLOAT` type and `cudnnMathType_t` set using `cudnnSetRNNMatrixMathType()` is `CUDNN_DEFAULT_MATH` or `CUDNN_TENSOR_OP_MATH`. Instead of using the default TF32 computation when Tensor Cores are used, a down conversion to FP16 (half-precision) is performed; same as in the `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` mode. This introduces a lower dynamic range of intermediate data but possibly faster execution. To disable the automatic down conversion of `CUDNN_DATA_FLOAT` weights and data in RNN APIs, set the environmental variable `NVIDIA_TF32_OVERRIDE` to `0` (notice this will disable the use of TF32 in the entire library, which might have a performance impact on CNNs that are not affected by this issue). Another workaround is to assign the `CUDNN_FMA_MATH` mode to the `cudnnMathType_t` argument in `cudnnSetRNNMatrixMathType()`. Due to this, the A100 TF32 feature is not accessible for RNNs in cuDNN v8.0.1.

▶ Several cuDNN APIs are unable to directly support computations using integer types (`CUDNN_DATA_INT8`, `CUDNN_DATA_INT8x4`, `CUDNN_DATA_INT8x32` or `CUDNN_DATA_INT32`). Floating types (particularly `CUDNN_DATA_FLOAT`) are much more widely supported. If an API does not support the desired type, `cudnnTransformTensor()` can be used to support the use case by converting to/from a supported type and the desired type. Here are the steps for doing so:

1. Convert all input tensors from their native type to a supported type (`CUDNN_DATA_FLOAT` is recommended).
2. Run cuDNN API using the converted input tensors and output tensor descriptors set as `CUDNN_DATA_FLOAT`.
3. Convert all output tensors from a supported type to your desired output type.

> 💬 **Note:** This will require extra memory use for the temporary buffers. Further, this will introduce an additional round trip to memory that might noticeably impact performance.

▶ *Updated: August 24, 2020*

cuDNN convolution APIs may return `CUDNN_STATUS_EXECUTION_FAILED` when the number of input or output channels equals to or exceeds 2097152.

▶ *Updated: August 24, 2020*

When the user is using `cudnnRNN*` APIs with the problem sizes (input size, hidden size) being not multiples of 16 for FP16 tensors or multiples of 8 for FP32 tensors, users may encounter a return status of `CUDNN_STATUS_EXECUTION_FAILED`.

1.18. cuDNN Release 8.0.0 Preview

> **!** **ATTENTION:** This is the cuDNN 8.0.0 Preview release. This Preview release is for early testing and feedback, therefore, for production use of cuDNN, continue to use cuDNN 7.6.5. This release is subject to change based on ongoing performance tuning and functional testing. For feedback on the new backend API and deprecations, e-mail cudnn@nvidia.com.

These release notes are applicable to NVIDIA JetPack users of cuDNN unless appended specifically with *(not applicable for Jetson platforms)*.

> **📧** **Note:** cuDNN 8.0.0 passed GA quality testing and validation for TensorRT and JetPack users.

For previous cuDNN documentation, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

cuDNN library

- ▶ The cuDNN library has been split into the following libraries:

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

  - ▶ `cudnn_ops_train` - This entity contains common training routines and algorithms, such as batch normalization, softmax, dropout, and so on. The `cudnn_ops_train` library depends on `cudnn_ops_infer`.

  - ▶ `cudnn_cnn_infer` - This entity contains all routines related to convolutional neural networks needed at inference time. The `cudnn_cnn_infer` library depends on `cudnn_ops_infer`.

  - ▶ `cudnn_cnn_train` - This entity contains all routines related to convolutional neural networks needed during training time. The `cudnn_cnn_train` library depends on `cudnn_ops_infer`, `cudnn_ops_train`, and `cudnn_cnn_infer`.

  - ▶ `cudnn_adv_infer` - This entity contains all other features and algorithms. This includes RNNs, CTC loss, and multihead attention. The `cudnn_adv_infer` library depends on `cudnn_ops_infer`.

  - ▶ `cudnn_adv_train` - This entity contains all the training counterparts of `cudnn_adv_infer`. The `cudnn_adv_train` library depends on `cudnn_ops_infer`, `cudnn_ops_train`, and `cudnn_adv_infer`.

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

▶ cuDNN does not support mixing sub library versions. If there is a mismatch in the cuDNN version numbers in the cuDNN sub library header files, the build will crash. The versions must match on the major number and minor number, as well as the patch level.

▶ The cuDNN sub libraries must be installed under a single directory.

Multiple dynamic libraries

In order to link against a subset of cuDNN, you must know which subset of the API you are using and then link against the appropriate cuDNN sub components. The cuDNN sub components are as follows:

▶ `cudnn_ops_infer.so`

▶ `cudnn_ops_train.so`

▶ `cudnn_cnn_infer.so`

▶ `cudnn_cnn_train.so`

▶ `cudnn_adv_infer.so`

▶ `cudnn_adv_train.so`

cuDNN linking options

There are two different linking options:

▶ Linking against individual sub libraries: Users who link against individual sub libraries must be able to identify the API exposed by each cuDNN sub library. Users also must know the hierarchy of the different cuDNN sub libraries. Each `.so` or `.a` needs to be specified explicitly in the user's linking command, as well as any external dependencies cuDNN require. For more information, refer to the *Limitations* section below.

▶ Linking against the full cuDNN (compatibility option): This would allow users to use `-lcudnn`. `libcudnn.so` is provided as a shim layer that would open the appropriate cuDNN sub-library for any particular cuDNN API call. While `libcudnn.a` is largely unchanged, it is a statically linked file for all of cuDNN.

cuDNN loading options

For users who want a smaller memory footprint, there are two ways of loading the library.

▶ Cherry-pick loading: Each sub library is loaded only when accessed. This will cause the first reference to that sub library to take a long time but will ensure the user isn't loading more libraries than they need.

▶ All access loading: All available cuDNN sub libraries are loaded early during runtime.

New API functions

For a list of functions and data types that were added in this release, see API Changes For cuDNN 8.0.0.

General Support of CUDA Graph Capture

CUDA Graphs are now supported for all functions in this release; with the following restrictions.

▶ CUDA Toolkit 10.2 or higher is required.

▶ cuDNN 8.0.0 graphs are captured using the CUDA graph-capture APIs.

▶ any non-default use of textures by users of cuDNN must be disabled prior to capture

cuDNN 8.0.0 does not at this time offer API support to add operations to an existing CUDA graph directly; however, the captured graph may be added to an existing graph through the existing CUDA Graphs API.

Regarding texture usage, cuDNN 8.0.0 by default will not enable texture usage; expert users may enable texture usage where allowed, but that usage will prevent a successful CUDA Graph capture until disabled. In order for cuDNN 8.0.0 to be graph-capture compatible library-wide, the cuDNN 8.0.0 CTC API was updated as described elsewhere.

The usual restrictions for CUDA Graphs apply in addition to these restrictions here.

New APIs for convolution

A new set of API functions to provide a brand new approach to cuDNN that offers more fine-grain control of performance, numerical properties, and so on for convolution. Using this API, users directly access various engines that compute convolution forward propagation, backward data, backward filter, and generic support for fusion starting with a limited support in this cuDNN 8.0.0 release and expanding support in follow-up releases. Each engine has performance-tuning knobs such as GEMM tiling and split-K. Users can use this API to fine-tune their network by querying cuDNN's heuristics, or doing their own, to find the most optimal engine configuration with which cuDNN computes each network layer.

NVIDIA Ampere Architecture GPU support *(not applicable for Jetson platforms)*

▶ Added support for A100 GPU based on NVIDIA Ampere Architecture.

▶ cuDNN 8.0.0 has seen significant improvements when using A100 GPUs compared to NVIDIA Volta V100 with cuDNN 7.6.

▶ Added support for Tensor Float 32 (TF32) for 1D and 2D convolutions. Full support for TF32 will come in future releases such as grouped convolutions and 3D convolutions in addition to further performance tuning.

▶ Increased performance for the legacy Tensor Cores (mixed precision for 1D, 2D, 3D, and grouped convolutions.

NVIDIA Turing and NVIDIA Volta architecture improvements

▶ New kernels for Tensor Cores and heuristics update for 1D convolution resulting in performance improvements for speech networks such as Jasper and Tacotron2 and WaveGlow, in addition to support for grouped 1D convolution (QuartzNet).

▶ Added 3D convolutions support of NHWC and improved heuristics and kernels for Tensor Cores in NCHW resulting in performance improvements for VNet, UNet-Medical, and UNet-Industrial. Additionally, FP16 3D convolutions are supported as well.

▶ Better utilization of Tensor Cores and heuristics for grouped convolutions result in improvements for ResNext.

▶ More tuning for vision networks like ResNet-50 ([MXNet] [PyTorch] [TensorFlow]) and SSD ([PyTorch] [TensorFlow]) with new updated heuristics.

Operation fusion

Operation fusion can be achieved using the backend API. The general workflow is similar to running unfused operations, except that instead of creating a single operation Operation Graph, the user may specify a multi-operation Operation Graph. For more information, see Operation Fusion Via The Backend API in the *cuDNN Developer Guide*.

Depthwise convolution extension

We have extended the `fprop` and `dgrad` NHWC depthwise kernels to support more combinations (filter sizes/strides) such as 5x5/1x1, 5x5/2x2, 7x7/1x1, 7x7/2x2 (in addition to what we already have, 1x1/1x1, 3x3/1x1, 3x3/2x2), which provides good performance.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for 8.0.0.

Limitations

▶ Samples must be installed in a writable location, otherwise the samples can crash.

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 8.0.0 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

▶ Some data types are not widely supported by all cuDNN API. For example, `CUDNN_DATA_INT8x4` is not supported by many functions. In such cases, support is available by using cudnnTransformTensor() to transform the tensors from the desired type to a type supported by the API. For example, a user is able to transform input tensors from `CUDNN_DATA_INT8x4` to `CUDNN_DATA_INT8`, run the desired API and then transform output tensors from `CUDNN_DATA_INT8` to `CUDNN_DATA_INT8x4`. Note that this transformation will incur an extra round trip to memory.

▶ The tensor pointers and the filter pointers require at a minimum 4-byte alignment, including INT8 data in the cuDNN library.

▶ Some computational options in cuDNN 8.0.0 now require increased alignment on tensors in order to run efficiently. As always, cuDNN recommends users to align tensors to 128-bit boundaries that will be sufficiently aligned for any computational option in cuDNN. Doing otherwise may cause performance regressions in cuDNN 8.0.0 compared to cuDNN v7.6.

▶ For certain algorithms, when the computation is in float (32-bit float) and the output is in FP16 (half float), there are cases where the numerical accuracy between the different algorithms might differ. cuDNN 8.0.0 users can target the backend API to query the numerical notes of the algorithms to get the information programmatically. There are cases where algo0 and algo1 will have a reduced precision accumulation when users target the legacy API. In all cases, these numerical differences are not known to affect training accuracy even though they might show up in unit tests.

Deprecated Features

The following features are deprecated in cuDNN 8.0.0:

▶ Support for Ubuntu 14.04 has been deprecated in this release. Upgrade to 16.04 or 18.04 for continued support.

▶ Support for Mac OS X has been deprecated in this release. Linux and Windows OS are currently supported.

▶ cuDNN version 8 introduces a new API deprecation policy to enable a faster pace of innovation. A streamlined, two-step, deprecation policy will be used for all API changes starting with cuDNN version 8. For details about this new deprecation policy, see Backward Compatibility And Deprecation Policy in the *cuDNN Developer Guide*.

▶ Removed and deprecated API changes. For a list of removed and deprecated APIs, see API Changes For cuDNN 8.0.0.

Fixed Issues

The following issues have been fixed in this release:

- ► There is a known issue in that `cudnnDestroy()` does not destroy all that `cudnnCreate()` created. Calling `cudnnDestroy()` after `cudnnCreate()` has a memory leak in some tests of about 1.6 MB on host memory. This issue has been fixed in cuDNN 8.0.0.

- ► Starting in cuDNN 7.6.1, when using the experimental multihead attention API, it is possible that the forward and backward paths produce different results for the BERT model, when the batch size is greater than one and the number of heads is greater than one. This issue has been fixed in cuDNN 8.0.0.

- ► The description of `cudnnSetCTCLossDescriptorEx()` is not clear. This issue has been fixed in cuDNN 8.0.0.

- ► Documentation affecting 1x1 convolution functions are not clear, for example `cudnnFindConvolutionBackwardDataAlgorithm()`. This issue has been fixed in cuDNN 8.0.0.

- ► cuDNN forward convolution with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` does not propagate NANs in weights. This issue has been fixed in cuDNN 8.0.0.

- ► Document mathematical definitions of all operations in cuDNN. We include full mathematical descriptions for the convolution functions.

- ► The functions `cudnnGetConvolutionForwardAlgorithm_v7()` and `cudnnGetConvolutionForwardWorkspaceSize()` may return `CUDNN_STATUS_SUCCESS` while the execution of the same convolution returns `CUDNN_STATUS_NOT_SUPPORTED`. Similar issues may also happen for `convolutionBackwardData()` and `convolutionBackwardFilter()`. This issue is present in cuDNN 7.2.2 library and later versions. This has been fixed in cuDNN 8.0.0.

- ► Algorithms returned by `cudnnGetConvolution*Algorithm()` may, in some limited use cases, fail to execute when they are actually run. This is a cuDNN library-wide issue and applies for convolution forward, convolution backward data, and convolution backward filter operations. This issue is also present in versions before cuDNN 8.0.0 EA.

- ► cuDNN does not support CUDA graphs. When launching a CUDA graph constructed using a stream capture that includes a `cudnnConvolutionForward()` operation, you may see `cudaErrorLaunchFailure` error. This is because CUDA graphs were not supported. The user can proceed.

- ► There was a known performance drop in 3D convolutions for some cases on NVIDIA Turing GPUs since cuDNN 7.4.2. This has been fixed on T4. *(not applicable for Jetson platforms)*

- ► There are rare cases where `cudnnConvolution*` will return `STATUS_NOT_SUPPORTED` when `cudnn*GetWorkspaceSize` might return success for a given algorithm. This has been fixed in cuDNN 8.0.0.

▶ In previous versions of cuDNN,
`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` did not propagate NaN values in some cases. This is fixed in the current release. Users desiring the old behavior can configure ReLU activation and set the floor to be `-Inf`.

▶ The `multiHeadAttention` sample code was added to the cuDNN 7.6.3 release. The sample code includes a simple NumPy/Autograd reference model of the multihead attention block that computes the forward response and all derivatives. The test code demonstrates how to use the multihead attention API, access attention weights, and sequence data.

▶ *Updated: July 22, 2020*

  In version 7.6.x, `cudnnConvolutionBackwardData()` with `PSEUDO_HALF_CONFIG` with `CUDNN_TENSOR_OP_MATH` or `FLOAT_CONFIG` with `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` returns incorrect results in 3D convolution when the filter size of the `w` dimension is `1` and padding of the `w` dimension is `0`. This issue has been fixed in this release.

Known Issues

▶ Performance regressions on V100 are observed in this release on SSD inference use cases if not using TensorRT.

▶ There are significant performance regressions on pre-Volta GPUs and some NVIDIA Turing GPUs based on the TU102 architecture. This performance regression is not applicable to T4, NVIDIA JetPack, and NVIDIA Tegra.

▶ Sub-optimal performance is present in this release for all INT8 convolutions for all GPUs.

▶ The performance of cudnnConvolutionBiasActivationForward() is slower than v7.6 in most cases. This is being actively worked on and performance optimizations will be available in the upcoming releases.

▶ On K80 GPUs, when cudnnConvolutionForward() is used with `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm and half I/O data types a silent error might occur.

▶ There are some peer-to-peer documentation links that are broken within the cuDNN API Reference. These links will be fixed in the next release.

▶ `cudnnCnnTrainVersionCheck()` and `cudnnCnnInferVersionCheck()` are missing in this release and will be added in the GA release.

▶ Documentation of RNN new APIs and deprecations is not complete. The `cudnnRNNBackwardData_v8()` and `cudnnRNNBackwardWeights_v8()` functions will be implemented in the next release.

▶ cuDNN 8.0.0 Preview will not work with GA10x NVIDIA Ampere Architecture GPUs. This will be fixed in the next release.

▶ cuDNN 8.0.0 Preview build with Windows and CUDA 11.0 RC has reduced performance on 2D, 3D, and grouped convolutions compared to Linux.

▶ *Updated: June 12, 2020*

There is a known issue in cuDNN 8.0.0 when linking statically to cuDNN and using the library's 3D algo1 backward filter convolutions. Users will see that the libraries emit an internal error or incorrectly state that a shared library is missing. This is a bug that will be fixed in a future release.

▶ *Updated: June 25, 2019*

There is a known issue in cuDNN 8.0.0 when linking statically to cuDNN and using the library's 3D algo1 backward filter convolutions. Users will see that the library emit an internal error or incorrectly state that a shared library is missing. This is a bug that will be fixed in a future release.

▶ *Updated: June 25, 2019*

When using an RPM file on RedHat for installation, installing cuDNN v8 directly or using TensorRT 7.1.3 will enable users to build their application with cuDNN v8. However, in order for the user to compile an application with cuDNN v7 after cuDNN v8 is installed, the user must perform the following steps:

1. Issue `sudo mv /usr/include/cudnn.h /usr/include/cudnn_v8.h`.
2. Issue `sudo ln -s /etc/alternatives/libcudnn /usr/include/cudnn.h`.
3. Switch to cuDNN v7 by issuing `sudo update-alternatives --config libcudnn` and choose cuDNN v7 from the list.

Steps 1 and 2 are required for the user to be able to switch between v7 and v8 installations. After steps 1 and 2 are performed one time, step 3 can be used repeatedly and the user can choose the appropriate cuDNN version to work with. For more information, refer to the [Installing From An RPM File](#) and [Upgrading From v7 To v8](#) sections in the *cuDNN Installation Guide*.

▶ *Updated: July 22, 2020*

`cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()` erroneously returns `CUDNN_STATUS_INTERNAL_ERROR` when the workspace size argument value is less than the required workspace size as returned by their respective `cudnnGetWorkspace()` API.

▶ *Updated: August 24, 2020*

cuDNN convolution APIs may return `CUDNN_STATUS_EXECUTION_FAILED` when the number of input or output channels equals to or exceeds 2097152.

Chapter 2. cuDNN Release 7.x.x

2.1. cuDNN Release 7.6.5

This is the cuDNN 7.6.5 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN release notes, refer to the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Made performance improvements to several APIs including `cudnnAddTensor`, `cudnnOpTensor`, `cudnnActivationForward`, and `cudnnActivationBackward`.

▶ Separated the cuDNN datatype references and APIs from the *cuDNN Developer Guide* into a new cuDNN API.

▶ Published Best Practices For Using cuDNN 3D Convolutions.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for v7.6.5.

Limitations

Updated: June 5, 2020

▶ RNN and multihead attention API calls may exhibit non-deterministic behavior when the cuDNN 7.6.5 library is built with CUDA Toolkit 10.2 or higher. This is the result of a new buffer management and heuristics in the cuBLAS library. As described in the Results Reproducibility section in the *cuBLAS Library User's Guide*, numerical results may not be deterministic when cuBLAS APIs are launched in more than one CUDA stream using the same cuBLAS handle. This is caused by two buffer sizes (16 KB and 4 MB) used in the default configuration.

When a larger buffer size is not available at runtime, instead of waiting for a buffer of that size to be released, a smaller buffer may be used with a different GPU kernel. The kernel selection may affect numerical results. The user can eliminate the non-deterministic behavior of cuDNN RNN and multihead attention APIs, by setting a single buffer size in the `CUBLAS_WORKSPACE_CONFIG` environmental variable, for example, `:16:8` or `:4096:2`.

The first configuration instructs cuBLAS to allocate eight buffers of 16 KB each in GPU memory while the second setting creates two buffers of 4 MB each. The default buffer configuration in cuBLAS 10.2 and 11.0 is `:16:8:4096:2`, that is, we have two buffer sizes. In earlier cuBLAS libraries, such as cuBLAS 10.0, it used the `:16:8` non-adjustable configuration. When buffers of only one size are available, the behavior of cuBLAS calls is deterministic in multi-stream setups.

Fixed Issues

The following issues have been fixed in this release:

▶ Corrected the documentation for `cudnnBatchNormalization*` API functions, clarifying which are optional arguments and when the user must pass them to the API.

▶ Fixed a lack-of-synchronization issue when `cudnnRNNBackwardData()` and `cudnnRNNBackwardDataEx()` calls a kernel that is not synchronized back to the application's stream. This issue only appears when users are using bidirectional RNN using algo of `CUDNN_RNN_ALGO_STANDARD`. This issue affects cuDNN versions 5 through 7.6.4.

▶ Corrected-supported tensor format tables for `cudnnConvolutionForward()`.

▶ `cudnnConvolutionBackwardData` used to give wrong answers when the kernel size was >=30 in any dimension and the stride is 2 in that dimension; with the algorithm set to `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING`. This has been fixed.

▶ Fixed an issue where if the user uses `cudnnBatchNormalizationForwardInference` with the mode of `CUDNN_BATCHNORM_SPATIAL_PERSISTENT`, the API will return `CUDNN_STATUS_NOT_SUPPORTED` and not fall back to `CUDNN_BATCHNORM_SPATIAL` mode. Now, it falls back correctly similar to the behavior of the other batch normalization APIs including `cudnnBatchNormalizationForwardTraining`, `cudnnBatchNormalizationForwardTrainingEx`, `cudnnBatchNormalizationBackward`, and `cudnnBatchNormalizationBackwardEx`.

▶ Previously, when cuDNN invoked `convolve_common_engine_int8_NHWC kernel` for NHWC format, irrespective of the output data precision, the output values were clipped to be in the range from -128 to 127. In this release, we have fixed the issue. As a result, output values are clipped only for INT8 precision. Whereas if the output data is float precision, the values are not clipped.

Known Issues

▶ *Updated: August 24, 2020*

Two-dimensional forward convolutions using algo1 may segfault when the filter size is large. For example, we have observed this issue when the filter width and height are more than or equal to 363.

▶ *Updated: September 28, 2020*

`cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()` calls with `algo0` or `algo1` can result in an illegal memory access for `PSEUDO_HALF_CONFIG` data configuration when the number of elements in the output tensor is odd. This can be mitigated by allocating one extra element in the output buffer.

2.2.   cuDNN Release 7.6.4

This is the cuDNN 7.6.4 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

For previous cuDNN release notes, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Gained significant speed-up in multihead-attention forward training and inference.

Compatibility

For the latest compatibility software versions of the OS, CUDA, the CUDA driver, and the NVIDIA hardware, see the cuDNN Support Matrix for v7.6.4.

Limitations

▶ When launching a CUDA graph constructed using a stream capture that includes a `cudnnConvolutionForward` operation, the subsequent synchronization point reports a `cudaErrorLaunchFailure` error. This error appears when cuDNN is set to use a non-default stream.

Fixed Issues

The following issues have been fixed in this release:

▶ Earlier versions of cuDNN v7.6 contained symbols that would conflict with those of in TensorRT 5.1 and later. In some cases, these conflicts could lead to application crashes when applications linked against cuDNN and TensorRT. This issue is fixed in cuDNN 7.6.4.

▶ Addressed the regressions that were introduced in the `cudnnConvolutionBiasActivationForward` function in cuDNN 7.6.3. Previously, if this API had different values in destination data buffer and zData buffer, then incorrect results

were computed. This issue has been resolved and now the API will compute correct results even if users provide an arbitrary set of values to the destination data and zData.

▶ Multihead attention will now return `CUDNN_STATUS_ARCH_MISMATCH` for true-half configuration on devices with compute capability less than 5.3 (for example, most of Maxwell and all of NVIDIA Kepler, and so on), which do not have native hardware support for true half computation. Previously, an error like `CUDNN_STATUS_EXECUTION_FAILED` may be triggered or inaccurate results may be produced.

2.3. cuDNN Release 7.6.3

This is the cuDNN 7.6.3 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes. These release notes are applicable to both cuDNN and NVIDIA JetPack users unless appended specifically with *(not applicable for Jetson platforms)*.

For previous cuDNN release notes, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ The cuDNN 7.6.3 library now supports auto-padding for NHWC layout. The functional behavior, and the benefits of auto-padding as follows: *(not applicable for Jetson platforms)*

   ▶ For use cases where C and K dimensions of input and filter Tensors are not multiples of 8, the auto-padding feature increases the Tensor size so that the Tensor dimensions are multiples of eight.

   ▶ With auto-padding, the cuDNN library invokes faster kernels, improving the performance.

   ▶ With auto-padding, the performance with NHWC data layout is now comparable to that of the NCHW layout.

▶ Added support for `dataType=CUDNN_DATA_HALF` and `computePrec=CUDNN_DATA_HALF` in multihead attention forward (https://docs.nvidia.com/deeplearning/sdk/cudnn-api/index.html#cudnnMultiHeadAttnForward) and backward (gradient) (cudnnMultiHeadAttnBackwardData() and cudnnMultiHeadAttnBackwardWeights()) API functions. *(not applicable for Jetson platforms)*

▶ Multihead attention API now supports bias after the projections on **Q**, **K**, **V**, and **O** in the cudnnMultiHeadAttnForward() call (backward bias gradient is not yet supported). *(not applicable for Jetson platforms)*

   The new feature required a small API change in cudnnSetAttnDescriptor(): the `cudnnAttnQueryMap_t queryMap` argument is replaced with `unsigned attnMode` to pass various on and off options. This change is backward compatible with earlier API versions. *(not applicable for Jetson platforms)*

▶ Significantly improved the performance in typical multihead attention use cases in forward inference and training, especially when the vector length of each head is a multiple of 32 up to 128. *(not applicable for Jetson platforms)*

▶ Tensor Core support is added for true half and single-precision use cases in multihead attention. Users may use it by setting the `mathType` argument in [cudnnSetAttnDescriptor()](#) to `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`. *(not applicable for Jetson platforms)*

▶ The `multiHeadAttention` sample code is added. The sample code includes a compact NumPy/Autograd reference model of the multihead attention block that computes the forward response and all first-order derivatives. The test code demonstrates how to use the multihead attention API, access attention weights, and sequence data. *(not applicable for Jetson platforms)*

▶ Improved depth-wise convolution for forward, `dgrad`, and `wgrad` under the following conditions:

  ▶ Algorithm is algo1.

  ▶ Tensor format for filter is NCHW (`wgrad` supports NHWC also).

  ▶ Input and outputs are in FP16 and computation is in FP32.

  ▶ Filter size: 1x1, 3x3, 5x5, 7x7 (`dgrad` only supports stride 1).

  ▶ Math type is CUDNN_DEFAULT_MATH.

▶ Improved-grouped convolution for `cudnnConvolutionBackwardFilter()` in the configuration under the following conditions:

  ▶ Algorithm is `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`.

  ▶ Math type is `CUDNN_DEFAULT_MATH`.

  ▶ Tensor format for filter is NCHW.

  ▶ Input and outputs are in FP16 and computation is in FP32.

  ▶ Filter size: 1x1, 3x3, 5x5, 7x7

▶ Improved the performance of grouped convolution, for `cudnnConvolutionForward()` in the configuration under the following conditions:

  ▶ Algorithm is `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`

  ▶ Math type is `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOROP_MATH_ALLOW_CONVERSION`

  ▶ Tensor format for filter is NHWC.

  ▶ Input and outputs are in FP16/ FP32.

  ▶ Per group C and K == 4/8/16/32

  ▶ Filter size: 3x3

▶ Improved the performance of grouped convolution, for `cudnnConvolutionBackwardFilter()` in the configuration under the following conditions:

  ▶ Algorithm is `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`

- ▶ Math type is `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOROP_MATH_ALLOW_CONVERSION`
- ▶ Tensor format for filter is NHWC.
- ▶ Input and outputs are in FP16 and computation is in FP32.
- ▶ On NVIDIA Volta (compute capability 7.0)
- ▶ Per group C and K == 4/8/16/32
- ▶ Filter size: 1x1, 3x3

Fixed Issues

The following issues have been fixed in this release:

- ▶ Fixed an issue where cudnnMultiHeadAttnBackwardData() was producing incorrect results when K sequence length is longer than 32.
- ▶ Fixed a race condition in cudnnMultiHeadAttnBackwardData() that was producing intermittent incorrect results.
- ▶ The function cudnnCTCLoss() produced incorrect gradient result for label whose length is smaller than the maximal sequence length in the batch. This is fixed in cuDNN 7.6.3.

2.4.   cuDNN Release 7.6.2

This is the cuDNN 7.6.2 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

For previous cuDNN release notes, see the cuDNN Archived Documentation.

Key Features and Enhancements

The following features and enhancements have been added to this release:

- ▶ Enhanced the performance of 3D deconvolution using cudnnConvolutionBackwardData(), for the following configuration:
  - ▶ 2x2x2 filter and 2x2x2 convolution stride.
  - ▶ For FP16 for data input and output, and for accumulation.
  - ▶ For FP32 for data input and output, and for accumulation.
- ▶ Enhanced the performance of 3D convolution using cudnnConvolutionForward(), for the following configuration:
  - ▶ Tensor Core for FP16 for data input and output and FP32 accumulation when CUDNN_TENSOR_OP_MATH is set.
  - ▶ Tensor Core for FP32 for data input and output and FP32 accumulation when CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION is set.
- ▶ Enhanced the functionality of the data type cudnnFusedOps_t by adding the following three enums:

▶ `CUDNN_FUSED_CONV_SCALE_BIAS_ADD_ACTIVATION`

▶ `CUDNN_FUSED_SCALE_BIAS_ADD_ACTIVATION_GEN_BITMASK`, and

▶ `CUDNN_FUSED_DACTIVATION_FORK_DBATCHNORM`

Fixed Issues

The following issues have been fixed in this release:

▶ In cuDNN 7.6.1, on NVIDIA Volta architecture only, there may be a performance degradation when the function cudnnConvolutionBackwardFilter() is used for 3D convolutions with CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1. This is fixed in cuDNN 7.6.2.

▶ In cuDNN 7.6.1, on NVIDIA Turing and NVIDIA Pascal architectures, performance may be degraded for cudnnConvolutionBackwardData(), when used with the following conditions:

   ▶ CUDNN_CONVOLUTION_BWD_DATA_ALGO_0 for 3D convolutions

   ▶ `wDesc`, `dyDesc`, and `dxDesc` are all in NCDHW.

   ▶ Data type configuration is `FLOAT_CONFIG` (that is, single-precision data and compute).

   This is fixed in cuDNN 7.6.2.

▶ In cuDNN 7.6.1, in some cases the function cudnnConvolutionBackwardData() may fail with "disallowed mismatches" error on NVIDIA Turing (T4) and NVIDIA Volta (V100) architectures, when used with the following configuration:

   ▶ Algorithm is CUDNN_CONVOLUTION_BWD_DATA_ALGO_1

   ▶ Math type is `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOROP_MATH_ALLOW_CONVERSION`

   ▶ Tensor format for filter is NCHW.

   ▶ Input and outputs are in FP16 and computation is in FP32.

   This is fixed in cuDNN 7.6.2.

2.5.   cuDNN Release 7.6.1

This is the cuDNN 7.6.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ Performance is enhanced for 3D convolutions using Tensor Core for FP16 input and output data types, whenever they are supported. Moreover, for single-precision (FP32) I/O, cuDNN 7.6.1 will use these enhanced kernels whenever possible, and only when cudnnMathType_t

is set to `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`. See [cudnnConvolutionForward()](#) and [cudnnConvolutionBackwardData()](#) and [cudnnConvolutionBackwardFilter()](#).

▶ On Maxwell and NVIDIA Pascal architectures only, the performance of 3D convolutions with the kernel size of 128^3, when used with `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`, is enhanced.

▶ [API logging](#) is fully implemented for the experimental multihead attention API, namely, for the following functions:

  ▶ [cudnnCreateAttnDescriptor()](#)

  ▶ [cudnnDestroyAttnDescriptor()](#)

  ▶ [cudnnSetAttnDescriptor()](#)

  ▶ [cudnnGetAttnDescriptor()](#)

  ▶ [cudnnGetMultiHeadAttnBuffers()](#)

  ▶ [cudnnGetMultiHeadAttnWeights()](#)

  ▶ [cudnnMultiHeadAttnForward()](#)

  ▶ [cudnnMultiHeadAttnBackwardData()](#)

  ▶ [cudnnMultiHeadAttnBackwardWeights()](#)

  ▶ [cudnnSetSeqDataDescriptor()](#)

  ▶ [cudnnGetSeqDataDescriptor()](#)

  ▶ [cudnnCreateSeqDataDescriptor()](#)

  ▶ [cudnnDestroySeqDataDescriptor()](#)

▶ Performance of the experimental multihead attention forward API is enhanced. See [cudnnMultiHeadAttnForward()](#).

▶ Performance is enhanced for the fused convolution and fused `wgrad` fallback path. See [cudnnFusedOps_t](#).

Fixed Issues

The following issues have been fixed in this release:

▶ In cuDNN 7.6.0, the function [cudnnGetConvolutionBackwardDataWorkspaceSize()](#) returns a value for which [cudnnConvolutionBackwardData()](#), when used with `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0`, returns `CUDNN_STATUS_NOT_SUPPORTED`. This is fixed in cuDNN 7.6.1 so that now `cudnnGetConvolutionBackwardDataWorkspaceSize()` returns a proper value for `cudnnConvolutionBackwardData()`.

▶ In cuDNN 7.6.0 and earlier versions, when all the following conditions are true,

  ▶ RNN model is bi-directional,

  ▶ Cell type is LSTM,

  ▶ `cudnnRNNAlgo_t= CUDNN_RNN_ALGO_STANDARD`, and

  ▶ Dropout probability was greater than zero,

then the cudnnRNNBackwardWeights() function produces inaccurate and occasionally non-deterministic results.

This is fixed in cuDNN 7.6.1.

An underlying issue, where the same buffer was used for left to right and right-to-left directions when re-computing forward dropout results passed from one RNN layer to the next, was the cause of the bug.

▶ A bug in cuDNN 7.6.0 and earlier versions, in the cudnnRNNForwardTraining() function, related to dropout, is fixed in cuDNN 7.6.1.

When all the following conditions are true:

  ▶ cudnnRNNAlgo_t=CUDNN_RNN_ALGO_PERSIST_STATIC,

  ▶ cudnnMathType_t is CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION, and

  ▶ input data type is CUDNN_DATA_FLOAT,

then the FP32-to-FP16 conversion might be applied as a performance optimization.

When this down conversion is scheduled, a GPU kernel invoked by cudnnDropoutForward() would crash due to incorrect parameters being passed. In this case CUDA runtime reports the "misaligned address" error when reading the data from global memory.

▶ In cuDNN 7.6.0, on RHEL7 only, the /usr/src/cudnn_samples_v7/samples_common.mk file is missing. This requires a workaround to compile the cuDNN samples. This is fixed in cuDNN 7.6.1 and the workaround is not needed for cuDNN 7.6.1.

▶ In cuDNN 7.6.0, on pre-Volta hardware only, the function cudnnGetConvolutionBackwardFilterWorkspaceSize() can erroneously return CUDNN_STATUS_SUCCESS for cudnnConvolutionBackwardFilter() for 3D convolutions, using CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1 with NDHWC layout. When this occurs, the cudnnConvolutionBackwardFilter() function will process the data using a kernel that expects the data in NCDHW layout (the only format supported by wDesc in this case), leading to incorrect results. In cuDNN 7.6.1, this is fixed so that cudnnGetConvolutionBackwardFilterWorkspaceSize() will now return CUDNN_STATUS_NOT_SUPPORTED.

▶ In cuDNN 7.5.x and 7.6.0 for Jetson platform, in some cases the function cudnnConvolutionBackwardData() , when used with CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD, might return incorrect results. This is fixed in cuDNN 7.6.1.

▶ When the data type configuration is FLOAT_CONFIG, then cudnnGetConvolution*Algorithm(), for a few convolution sizes, incorrectly returns a slow algorithm for the NVIDIA Pascal architecture. This is fixed in cuDNN 7.5.0 and later versions.

▶ When using the `fusedOps` API with the enum
`CUDNN_FUSED_SCALE_BIAS_ACTIVATION_CONV_BNSTATS` or
`CUDNN_FUSED_SCALE_BIAS_ACTIVATION_WGRAD`, and when input tensor is in NCHW
format or is not fully packed, then incorrect results may be produced. This is now fixed in
cuDNN 7.6.1.

Known Issues

The following issues and limitations exist in this release:

▶ Algorithms returned by `cudnnGetConvolution*Algorithm()` may, in some limited use
cases, fail to execute when they are actually run. This is a cuDNN library-wide issue and
applies for convolution forward, convolution backward data, and convolution backward
filter operations. This issue is also present in versions before cuDNN 7.6.1.

▶ When the input and output tensors are in NHWC and the filter is 1x1 and NCHW, the
performance of the function cudnnConvolutionBackwardData() might be degraded.

▶ In cuDNN 7.6.1, when using the experimental multihead attention API, it is possible that
the forward and backward paths produce different results for the BERT model, when the
batch size is greater than one and the number of heads is greater than one.

▶ In cuDNN 7.6.1, on NVIDIA Volta architecture only, there may be a performance
degradation when the function cudnnConvolutionBackwardFilter() is used for 3D
convolutions with `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`.

▶ In cuDNN 7.6.1, on NVIDIA Turing and NVIDIA Pascal architectures, performance may be
degraded for cudnnConvolutionBackwardData(), when used with the following conditions:

  ▶ `CUDNN_CONVOLUTION_BWD_DATA_ALGO_0` for 3D convolutions.

  ▶ `wDesc`, `dyDesc`, and `dxDesc` are all in NCDHW.

  ▶ Data type configuration is `FLOAT_CONFIG` (that is, single-precision data and compute).

2.6.   cuDNN Release 7.6.0

This is the cuDNN 7.6.0 release notes. This release includes fixes from the previous cuDNN
v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ A new API is introduced for fused ops, which can accelerate many use cases in ResNet-
like networks. With this new API, it is now possible to execute various fused operations
such as apply per channel scale and bias, perform activation, compute convolution, and
generate batchnorm statistics. Below is a list of supported datatype and functions in this
API:

Datatypes:

- ► `cudnnFusedOpsVariantParamPack_t`
- ► `cudnnFusedOpsConstParamPack_t`
- ► `cudnnFusedOpsPlan_t`
- ► `cudnnFusedOps_t`
- ► `cudnnFusedOpsConstParamLabel_t`
- ► `cudnnFusedOpsPointerPlaceHolder_t`
- ► `cudnnFusedOpsVariantParamLabel_t`

Functions:

- ► `cudnnCreateFusedOpsConstParamPack`
- ► `cudnnDestroyFusedOpsConstParamPack`
- ► `cudnnSetFusedOpsConstParamPackAttribute`
- ► `cudnnGetFusedOpsConstParamPackAttribute`
- ► `cudnnCreateFusedOpsVariantParamPack`
- ► `cudnnDestroyFusedOpsVariantParamPack`
- ► `cudnnSetFusedOpsVariantParamPackAttribute`
- ► `cudnnGetFusedOpsVariantParamPackAttribute`
- ► `cudnnCreateFusedOpsPlan`
- ► `cudnnDestroyFusedOpsPlan`
- ► `cudnnMakeFusedOpsPlan`
- ► `cudnnFusedOpsExecute`

► Improved the performance of grouped convolution layers in ResNeXt-50, for `cudnnConvolutionBackwardData()` in the configuration below:

- ► On NVIDIA Volta (compute capability 7.0)
- ► Algorithm is `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`
- ► Stride of 1
- ► Math type is `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOROP_MATH_ALLOW_CONVERSION`
- ► Tensor format for filter is NHWC.
- ► Input and outputs are in FP16 and computation is in FP32.

► A new API is introduced to enhance the inference time. With this new API, it is now possible to separate the filter layout transformation that was applied on every call, which in turn leads to inference time enhancement. Below is a list of supported datatype and functions in this API.

- ► `cudnnReorderType_t`
- ► `cudnnReorderFilterAndBias`
- ► `cudnnSetConvolutionReorderType`
- ► `cudnnGetConvolutionReorderType`

▶ Performance is enhanced (by selecting a faster kernel) on NVIDIA T4 cards for INT8x4 and INT8x32.

Fixed Issues

The following issues have been fixed in this release:

▶ In cuDNN 7.5.0 and cuDNN 7.5.1, a bug in the `cudnnRNNBackwardData()` function affected the thread synchronization. This effect is limited to only the first iteration of the loop, and only in some paths. This occurs when using the function with the CUDNN_RNN_ALGO_PERSIST_STATIC method. This is fixed in cuDNN 7.6.0.

Known Issues

The following issues and limitations exist in this release:

▶ The `cudnnConvolutionBackwardData()` function for CUDNN_CONVOLUTION_BWD_DATA_ALGO_0 fails with CUDNN_STATUS_NOT_SUPPORTED when the input size is large.

▶ A general known issue for cuDNN library: the Tensor pointers and the filter pointers require at a minimum 4-byte alignment, including for FP16 or INT8 data.

▶ On RHEL7 only, the `/usr/src/cudnn_samples_v7/samples_common.mk` file is missing. This will prevent compiling the cuDNN samples. The workaround is to copy the below contents into "samples_common.mk" text file and place this file in the "/usr/src/cudnn_samples_v7/" directory, so that the `/usr/src/cudnn_samples_v7/samples_common.mk` file exists.

```
# Setting SMS for all samples
# architecture

ifneq ($(TARGET_ARCH), ppc64le)
CUDA_VERSION := $(shell cat $(CUDA_PATH)/include/cuda.h |grep "define
 CUDA_VERSION" |awk '{print $$3}')
else
CUDA_VERSION := $(shell cat $(CUDA_PATH)/targets/ppc64le-linux/include/cuda.h |
grep "define CUDA_VERSION" |awk '{print $$3}')
endif

#Link against cublasLt for CUDA 10.1 and up.
CUBLASLT:=false
ifeq ($(shell test $(CUDA_VERSION) -ge 10010; echo $$?),0)
CUBLASLT:=true
endif
$(info Linking agains cublasLt = $(CUBLASLT))

ifeq ($(CUDA_VERSION),8000 )
SMS_VOLTA =
else
ifneq ($(TARGET_ARCH), ppc64le)
ifeq ($(CUDA_VERSION), $(filter $(CUDA_VERSION), 9000 9010 9020))
SMS_VOLTA ?= 70
else
ifeq ($(TARGET_OS), darwin)
SMS_VOLTA ?= 70
else
SMS_VOLTA ?= 70 72 75
```

```
endif #ifneq ($(TARGET_OS), darwin)
endif #ifeq ($(CUDA_VERSION), $(filter $(CUDA_VERSION), 9000 9010 9020))
else
SMS_VOLTA ?= 70
endif #ifneq ($(TARGET_ARCH), ppc64le)
endif #ifeq ($(CUDA_VERSION),8000 )
SMS ?= 30 35 50 53 60 61 62 $(SMS_VOLTA)
```

2.7.   cuDNN Release 7.5.1

This is the cuDNN 7.5.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ The function `cudnnMultiHeadAttnForward()` is now enabled to sweep through all the time steps in a single API call. This is indicated by a negative value of the `currIdx` argument in the inference mode, that is, when `reserveSpace=NULL` so that either `cudnnMultiHeadAttnBackwardData()` or `cudnnMultiHeadAttnBackwardWeights()` will not be invoked. This sweep mode can be used to implement self-attention on the encoder side of the transformer model.

Fixed Issues

The following issues have been fixed in this release:

▶ In cuDNN 7.5.0, using the static link for `cudnnConvolutionBiasActivationForward()` function may result in CUDNN_STATUS_NOT_SUPPORTED error message. The workaround is to perform a whole-archive link. This issue is fixed in cuDNN 7.5.1.

▶ In cuDNN 7.5.0 and 7.4.x, in some cases of input images with large dimensions, the 3D forward convolution operations with CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM will cause a crash with "illegal memory access" error. This is fixed in cuDNN 7.5.1.

▶ In cuDNN 7.5.0, setting `attnDropoutDesc=NULL` in `cudnnSetAttnDescriptor()` triggered a segmentation fault in `cudnnMultiHeadAttnForward()`, even though the user is required to set it to NULL in the inference mode. This is fixed in cuDNN 7.5.1.

Known Issues

The following issues and limitations exist in this release:

▶ In cuDNN7.5 and cudnn7.5.1, image size smaller than filter size is unsupported, even with sufficient padding.

2.8. cuDNN Release 7.5.0

This is the cuDNN 7.5.0 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following features and enhancements have been added to this release:

▶ In `cudnnConvolutionForward()` for 2D convolutions, for `wDesc` NCHW, the `IMPLICIT_GEMM` algorithm (algo 0) now supports the data type configuration of `INT8x4_CONFIG` and `INT8x4_EXT_CONFIG`.

▶ A new set of APIs is added to provide support for multihead attention computation. The following is a list of the new functions and data types:

Datatypes:

  ▶ `cudnnSeqDataAxis_t`
  ▶ `cudnnMultiHeadAttnWeightKind_t`
  ▶ `cudnnSeqDataDescriptor_t`
  ▶ `cudnnWgradMode_t`
  ▶ `cudnnAttnQueryMap_t`
  ▶ `cudnnAttnDescriptor_t`

Functions:

  ▶ `cudnnCreateAttnDescriptor`
  ▶ `cudnnDestroyAttnDescriptor`
  ▶ `cudnnSetAttnDescriptor`
  ▶ `cudnnGetAttnDescriptor`
  ▶ `cudnnGetMultiHeadAttnBuffers`
  ▶ `cudnnGetMultiHeadAttnWeights`
  ▶ `cudnnMultiHeadAttnForward`
  ▶ `cudnnMultiHeadAttnBackwardData`
  ▶ `cudnnMultiHeadAttnBackwardWeights`
  ▶ `cudnnSetSeqDataDescriptor`
  ▶ `cudnnGetSeqDataDescriptor`
  ▶ `cudnnCreateSeqDataDescriptor`
  ▶ `cudnnDestroySeqDataDescriptor`

▶ A new set of APIs for general tensor folding is introduced. The following is a list of the new functions and data types:

Datatypes:

- ► `cudnnTensorTransformDescriptor_t`
- ► `cudnnFoldingDirection_t`

Functions:

- ► `cudnnTransformTensorEx`
- ► `cudnnCreateTensorTransformDescriptor`
- ► `cudnnDestroyTensorTransformDescriptor`
- ► `cudnnInitTransformDest`
- ► `cudnnSetTensorTransformDescriptor`
- ► `cudnnGetTensorTransformDescriptor`

► A new set of APIs, and enhancements for the existing APIs, are introduced for RNNs. The following is the list of the new and enhanced functions and data types:

Datatypes:

- ► `cudnnRNNBiasMode_t` (new)
- ► `cudnnRNNMode_t` (enhanced)

Functions:

- ► `cudnnSetRNNBiasMode` (new)
- ► `cudnnGetRNNBiasMode` (new)
- ► `cudnnGetRNNLinLayerBiasParams` (enhanced)

► All `cudnnRNNForward/Backward*` functions are enhanced to support FP16 math precision mode when both input and output are in FP16. To switch to FP16 math precision, set the `mathPrec` parameter in `cudnnSetRNNDescriptor` to CUDNN_DATA_HALF. To switch to FP32 math precision, set the `mathPrec` parameter in `cudnnSetRNNDescriptor` to CUDNN_DATA_FLOAT. This feature is only available for `CUDNN_ALGO_STANDARD` and for the compute capability 5.3 or higher.

► Added support for INT8x4 and INT8x32 data type for `cudnnPoolingForward`. Using these will provide improved performance over scalar data type.

Fixed Issues

The following issues have been fixed in this release:

► When the following is true for the `cudnnConvolutionBackwardData()` function:

- ► used with CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING, and
- ► `convDesc`'s vertical stride is exactly 2, and
- ► the vertical padding is a multiple of 2, and
- ► the filter height is a multiple of 2

OR

- ▶ used with CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING, and
- ▶ convDesc's horizontal stride is exactly 2, and
- ▶ the horizontal padding is a multiple of 2, and
- ▶ the filter width is a multiple of 2

then the resulting output is incorrect. This issue was present in cuDNN 7.3.1 and later. This is fixed in cuDNN 7.5.0.

▶ The mathPrec parameter in cudnnSetRNNDescriptor is reserved for controlling math precision in RNN, but was not checked or enforced. This parameter is now strictly enforced. As a result, the following applies:

- ▶ For the I/O in FP16, the parameter mathPrec can be CUDNN_DATA_HALF or CUDNN_DATA_FLOAT.
- ▶ For the I/O in FP32, the parameter mathPrec can only be CUDNN_DATA_FLOAT.
- ▶ For the I/O in FP64, double type, the parameter mathPrec can only be CUDNN_DATA_DOUBLE.

▶ Users upgrading to cuDNN 7.4 may see insufficiently small values returned from the function cudnnGetConvolutionBackwardFilterWorkspaceSize () for dimensions 5 and greater, resulting in a CUDNN_STATUS_EXECUTION_FAILED error message. In cuDNN 7.4, the workaround for this issue is to calculate the workspace by using the formula below:

```
Let M be the product of output tensor (gradDesc) dimensions starting at 1.
Let N be the output tensor dimension 0.
Let Mp = (M+31)/32
Let Np = (N+31)/32
W = 2 * M * N * sizeof(int) is the workspace that should be used.
```

This is fixed.

▶ In earlier cuDNN versions, when all the conditions below are true:

- ▶ 3D convolution
- ▶ Batch size > 1
- ▶ Algorithm is CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1
- ▶ convDesc's dataType is CUDNN_DATA_HALF, then, calls to cudnnConvolutionBackwardFilter() may produce incorrect (and non-deterministic) results. This is fixed in cuDNN 7.5.0.

▶ In cuDNN 7.4.2, for some cases the 3D convolution resulted in a reduced performance on NVIDIA Turing GPUs, compared to the previous cuDNN releases. This is fixed.

▶ For int8x32 datatype, the function cudnnSetTensor4dDescriptorEx erroneously returns CUDNN_STATUS_BAD_PARAM. Now it is fixed in cuDNN 7.5 so it no longer returns bad param.

► In cuDNN 7.4.1 and 7.4.2, when `cudnnBatchNormMode_t` is set to `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` and the I/O tensors are in NHWC format and of `CUDNN_DATA_HALF` datatype, then, on Windows only, the `cudnnBatchNormalization*Ex` functions are supported only with the device in TCC mode. Refer to [Tesla Compute Cluster Mode for Windows](#) for more information.

Starting with cuDNN 7.5.0, the following checks are added for the driver mode on Windows. If on Windows and not in TCC mode:

   ► The functions fallback to a slower implementation if `bnOps` in the `cudnnBatchNormalization*Ex` function is set to CUDNN_BATCHNORM_OPS_BN.

   ► If `bnOps` is set to `CUDNN_BATCHNORM_OPS_BN_ACTIVATION`, or `CUDNN_BATCHNORM_OPS_BN_ADD_ACTIVATION`, the `CUDNN_STATUS_NOT_SUPPORTED` is returned.

► In cuDNN 7.4.2, in some cases the `cudnnConvolutionBackwardData()` function, when used with NHWC tensor format, resulted in the "disallowed mismatches" error. This is fixed.

► In some cases, using `cudnnConvolutionBiasActivationForward()` with `GroupCount()` > 1 and `xDesc`'s data type is `CUDNN_DATA_HALF` will produce incorrect results for all groups except the first. This is fixed.

► When using cuDNN 7.3.1 on Quadro P4000, when calling the `cudnnConvolutionForward()` function with `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED` algorithm, there was a small chance of seeing intermittent inaccurate results. This is fixed.

► When `cudnnConvolutionForward()` is called with these settings:

   ► The datatype is `CUDNN_DATA_INT8x4`.
   ► The convolution is 2D.
   ► The architecture is `sm_61`.
   ► The filter size is larger than 8x8.

Then, incorrect results and potential illegal memory access errors occur. This is fixed.

► For `sm_72` and `sm_75`, the function `cudnnConvolutionBiasActivationForward()`, when used with INT8x32, failed to run. This is fixed.

► In the function `cudnnSetRNNDataDescriptor`, if API logging is turned on, the `seqLengthArray` field in the log may not display the correct number of array elements. This is fixed.

► For the batchNorm functions `cudnnBatchNormalization{Backward|BackwardEx|ForwardInference|ForwardTraining|ForwardTrainingEx}`, the value of `epsilon` is required to be greater or equal to `CUDNN_BN_MIN_EPSILON` that was defined in the `cudnn.h` file to the value 1e-5. This threshold value is now lowered to 0.0 to allow a wider range of `epsilon` value. However, users should still choose the `epsilon` value carefully,

since a too small a value of `epsilon` may cause batchNormalization to overflow when the input data's standard deviation is close to 0.

▶ Some Grouped Convolutions (particularly those used in Depthwise-Separable convolutions) may return INTERNAL_ERROR if they have all inputs/outputs as NHWC-packed and do not match one of the following criteria:

  ▶ filter_height = 1, filter_width = 1, vertical_conv_stride = 1, horizontal_conv_stride = 1

  ▶ filter_height = 3, filter_width = 3, vertical_conv_stride = 1, horizontal_conv_stride = 1

  ▶ filter_height = 3, filter_width = 3, vertical_conv_stride = 2, horizontal_conv_stride = 2

Known Issues

The following issues and limitations exist in this release:

▶ The RNN persist-static algorithm returns incorrect results for GRU problems in backwards mode, when the hidden size is greater than 1024. Due to this, RNN persist-static algorithm is disabled in cuDNN 7.5.0. Users with such GRU problems are advised to use the standard or persist-dynamic RNN algorithms. See `cudnnRNNAlgo_t`. This note applies to all previous cuDNN 7 releases.

▶ The function `cudnnConvolutionBackwardFilter()`, when used with CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1, returns the error "`Uninitialized __global__ memory read of size 4`".

2.9.   cuDNN Release 7.4.2

This is the cuDNN 7.4.2 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Fixed Issues

The following issues have been fixed in this release:

▶ In some cases when the data is in `CUDNN_DATA_HALF` and NHWC, illegal memory access may occur for `cudnnBatchNormalization*` functions in the cuDNN 7.4.1 library. This is now fixed.

▶ When the data is in `CUDNN_DATA_HALF` and NHWC, for `cudnnBatchNormalization*` functions when (N*H*W) is large and odd number, the output may contain wrong results. This is fixed.

▶ When calling the `cudnnConvolutionBiasActivationForward()` function with the `algo` parameter set to CUDNN_CONVOLUTION_FWD_ALGO_FFT and the `activationDesc` parameter set to CUDNN_ACTIVATION_RELU and sufficiently large inputs, the ReLU operation is not applied and negative values are passed through to the output. This issue is now fixed. This issue was present in all previous cuDNN versions.

▶ Performance regression was introduced in cuDNN 7.4.1 for `cudnnConvolutionBwdFilterAlgo_t()` function with `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` algorithm. This is fixed.

Known Issues

The following issues and limitations exist in this release:

▶ When `cudnnBatchNormMode_t` is set to `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` and the I/O tensors are in NHWC format and of `CUDNN_DATA_HALF` datatype, then, **on Windows only**, the `cudnnBatchNormalization*Ex` functions are supported only with the device in TCC mode. See Tesla Compute Cluster Mode for Windows. This issue is not present on Linux systems. This issue is present in cuDNN 7.4.1 and this current version.

▶ In some cases, the 3D convolution will have a reduced performance on NVIDIA Turing GPUs, compared to the previous cuDNN releases.

▶ The functions `cudnnGetConvolutionForwardAlgorithm_v7()` and `cudnnGetConvolutionForwardWorkspaceSize()` will return `CUDNN_STATUS_SUCCESS`, but the execution of the convolution returns `CUDNN_STATUS_NOT_SUPPORTED`. This issue is present in cuDNN 7.2.2 library and later versions.

2.10.   cuDNN Release 7.4.1

This is the cuDNN 7.4.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ Added a new family of fast NHWC batch normalization functions. Refer to the following five new functions and one new type descriptor:

   ▶ `cudnnGetBatchNormalizationForwardTrainingExWorkspaceSize()` function

   ▶ `cudnnBatchNormalizationForwardTrainingEx` function

   ▶ `cudnnGetBatchNormalizationBackwardExWorkspaceSize()` function

   ▶ `cudnnBatchNormalizationBackwardEx()` function

   ▶ `cudnnGetBatchNormalizationTrainingExReserveSpaceSize()` function

   ▶ `cudnnBatchNormOps_t` type descriptor

▶ For API Logging, a conversion specifier for the process id is added. With this, the process id can be included in the log file name. Refer to API Logging for more information.

▶ Performance of `cudnnPoolingBackward()` is enhanced for the average pooling when using NHWC data format-for both the `CUDNN_POOLING_AVERAGE_COUNT_INCLUDE_PADDING` and `CUDNN_POOLING_AVERAGE_COUNT_EXCLUDE_PADDING` cases of `cudnnPoolingMode_t`.

▶ Performance of the strided convolution in `cudnnConvolutionBackwardData()` is enhanced when the filter is in NHWC format and the data type is `TRUE_HALF_CONFIG`, `PSEUDO_HALF_CONFIG`, or `FLOAT_CONFIG`. For strides `u,v < r,s` the performance is further enhanced.

▶ Significantly improved the performance of `cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()` functions on RCNN models such as Fast RCNN, Faster RCNN, and Mask RCNN.

Fixed Issues

The following issues have been fixed in this release:

▶ The following set-up was giving "Misaligned Address" error in cuDNN 7.3.x. This is fixed in cuDNN 7.4.1: For the `cudnnConvolutionForward()` function with the `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM` algorithm, in the data type configuration of `PSEUDO_HALF_CONFIG`, when the input and output tensors are in NHWC and the filter is 1x1 and NCHW, and Tensor Op is enabled.

▶ For a few convolution sizes for `ALGO_0` and `ALGO_1`, the performance of the function `cudnnConvolutionBackwardFilter()` was degraded in cuDNN 7.3.1. This is now fixed.

▶ Fixed. In cuDNN 7.3.1, the function `cudnnAddTensor` was computing incorrect results when run on GPUs with the compute capability < 6.0 (before NVIDIA Pascal).

Known Issues

The following issues and limitations exist in this release:

▶ When calling the `cudnnConvolutionBiasActivationForward()` function with the `algo` parameter set to `CUDNN_CONVOLUTION_FWD_ALGO_FFT` and the `activationDesc` parameter set to `CUDNN_ACTIVATION_RELU` and sufficiently large inputs, the ReLU operation is not applied and negative values are passed through to the output. This issue is present in all previous cuDNN versions.

2.11.   cuDNN Release 7.3.1

This is the cuDNN 7.3.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ The FFT tiling algorithms for convolution have been enhanced to support strided convolution. In specific, for the algorithms `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` and `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT_TILING`, the `convDesc`'s vertical and horizontal filter stride can be 2 when neither the filter width nor the filter height is 1.

▶ The `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD` algorithm for `cudnnConvolutionForward()` and `cudnnConvolutionBackwardData()` now give superior performance for NVIDIA Volta architecture. In addition, the mobile version of this algorithm in the same functions gives superior performance for Maxwell and NVIDIA Pascal architectures.

▶ Dilated convolutions now give superior performance for `cudnnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()` on NVIDIA Volta architecture, in some cases.

Known Issues and Limitations

The following issues and limitations exist in this release:

▶ For the `cudnnConvolutionForward()`, when using a 1x1 filter with input and output tensors of `NHWC` format and of `CUDNN_DATA_HALF` (half precision) type, and the filter format is `NCHW`, with compute type of float, cuDNN will generate incorrect results.

▶ On Quadro P4000, when calling `cudnnConvolutionForward()` function with `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED` algorithm, there may be a small chance of seeing intermittent inaccurate results.

▶ When using `cudnnConvolutionBackwardFilter()` with `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0` in mixed precision computation, with I/O in `CUDNN_DATA_HALF` (half precision) and compute type of float, when the number of batches (N) is larger than 1 the results might include INF due to an intermediate down convert to half float. In other words, with an accumulation of float for all intermediate values (such as in `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`) the result will be a finite half precision float. This limitation also exists in all previous cuDNN versions.

Fixed Issues

The following issues have been fixed in this release:

▶ Fixed a pointer arithmetic integer overflow issue in RNN forward and backward functions, when sequence length and mini-batch size are sufficiently large.

▶ When tensor cores are enabled in cuDNN 7.3.0, the `cudnnConvolutionBackwardFilter()` calculations were performing an illegal memory access when K and C values are both non-integral multiples of 8. This issue is fixed.

▶ For the `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` algorithm in `cudnnConvolutionBackwardFilter()`, on NVIDIA Volta, the tensor operations were occasionally failing when the filter-spatial size (filter `h` * filter `w`) was greater than 64. This issue is fixed.

▶ While running cuDNN 7.3.0 on NVIDIA Turing with CUDA 10.0, r400 driver, the functions `cudnnRNNForwardTraining(Ex)` and `cudnnRNNForwardInference(Ex)` errored out returning `CUDNN_STATUS_NOT_SUPPORTED`. This issue is fixed.

▶ In cuDNN 7.3.0, when using `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` with tensor data or filter data in `NHWC` format, the function might have resulted in a silent failure. This is now fixed.

2.12.   cuDNN Release 7.3.0

This is the cuDNN 7.3.0release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ Support is added to the following for the dilated convolution, for `NCHW` and `NHWC` filter formats:

  ▶ `cudnnConvolutionForward()` for 2D

  ▶ `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`

  ▶ `cudnnConvolutionBackwardData()` for 2D

  ▶ `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`

  ▶ `cudnnConvolutionBackwardFilter()` for 2D

  ▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`

  For these supported cases, the dilated convolution is expected to offer superior speed, compared to the existing dilated convolution with algo 0.

▶ Grouped convolutions for depth-wise separable convolutions are optimized for the following NHWC formats: HHH (input: Half, compute: Half, output: Half), HSH, and SSS.

▶ While using `CUDNN_TENSOR_OP_MATH` or `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION`, with the tensor cores, the c, and k dimensions of the tensors are now padded to multiples of 8 (as needed), to allow a tensor core kernel to run.

▶ The `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` algo is enhanced in `cudnnBatchNormalizationForwardTraining()` and `cudnnBatchNormalizationBackward()` to propagate NaN-s or Inf-s as in a pure floating point implementation (the "persistent" flavor of the batch normalization is optimized for speed and it uses integer atomics for inter thread-block reductions). In earlier versions of cuDNN, we recommended invoking `cudnnQueryRuntimeError()` to ensure that no overflow was encountered. When it happened, the best practice was to discard the results, and use `CUDNN_BATCHNORM_SPATIAL` instead, as some results generated by `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` could be finite but invalid. This behavior is now corrected: NaN-s and Inf-s are consistently output when intermediate results are out of range. The refined implementation simulates math operations on special floating point values, for example, +Inf-Inf=NaN.

Known Issues and Limitations

Following issues and limitations exist in this release:

▶ When tensor cores are enabled in cuDNN 7.3.0, the wgrad calculations will perform an illegal memory access when K and C values are both non-integral multiples of 8. This will not likely produce incorrect results, but may corrupt other memory depending on the user buffer locations. This issue is present on NVIDIA Volta and NVIDIA Turing architectures.

▶ Using `cudnnGetConvolution*_v7` routines with `cudnnConvolutionDescriptor_t` set to `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` leads to incorrect outputs. These incorrect outputs will consist only of `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` cases, instead of also returning the performance results for both `DEFAULT_MATH` and `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` cases.

Fixed Issues

The following issues have been fixed in this release:

▶ Using `cudnnConvolutionBackwardData()` with `CUDNN_CONVOLUTION_BWD_DATA_ALGO_WINOGRAD` algorithm produced incorrect results due to an incorrect filter transform. This issue was present in cuDNN 7.2.1.

▶ For INT8 type, with `xDesc` and `yDesc` of NHWC format, the `cudnnGetConvolutionForwardAlgorithm_v7` function was incorrectly returning `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM` as a valid algorithm. This is fixed.

▶ `cudnnConvolutionForward()` using `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD` intermittently produced incorrect results in cuDNN 7.2, due to a race condition. This issue is fixed.

▶ When running `cudnnConvolutionBackwardFilter()` with NHWC filter format, when `n`, `c`, and `k` are all multiple of 8, and when the `workSpace` input is exactly as indicated by `cudnnGetConvolutionBackwardFilterWorkspaceSize()`, leads to error in cuDNN 7.2. This is fixed.

▶ When the user runs `cudnnRNNForward*` or `cudnnRNNBackward*` with FP32 I/O on sm_70 or sm_72, with RNN descriptor's `algo` field set to `CUDNN_RNN_ALGO_PERSIST_STATIC`, and `cudnnMathType_t` type set to `CUDNN_TENSOR_OP_MATH` using `cudnnSetRNNMatrixMathType`, then the results were incorrect. This is fixed.

▶ When the user runs `cudnnRNNForward*` or `cudnnRNNBackward*` with FP32 I/O on sm_70 or sm_72, with RNN descriptor's `algo` field set to `CUDNN_RNN_ALGO_PERSIST_STATIC`, and `cudnnMathType_t` type set to `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` using `cudnnSetRNNMatrixMathType`, then the resulting performance was suboptimal. This is fixed.

▶ Convolution routines with filter format as NHWC require both input and output formats to be NHWC. However, in cuDNN 7.2 and earlier, this condition was not being checked for, as

a result of which silent failures may have occurred. This is fixed in 7.3.0 to correctly return `CUDNN_STATUS_NOT_SUPPORTED`.

2.13.   cuDNN Release 7.2.1

This is the cuDNN 7.2.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ The following new functions are added to provide support for the padding mask for the `cudnnRNN*` family of functions:

  ▶ `cudnnSetRNNPaddingMode()`: Enables/disables the padded RNN I/O.

  ▶ `cudnnGetRNNPaddingMode()`: Reads the padding mode status.

  ▶ `cudnnCreateRNNDataDescriptor()` and `cudnnDestroyRNNDataDescriptor()`: Creates and destroys, respectively, `cudnnRNNDataDescriptor_t`, an RNN data descriptor.

  ▶ `cudnnSetRNNDataDescriptor()` and `cudnnGetRNNDataDescriptor()`: Initializes and reads, respectively, the RNN data descriptor.

  ▶ `cudnnRNNForwardTrainingEx()`: An extended version of the `cudnnRNNForwardTraining()` to allow for the padded (unpacked) layout for the I/O.

  ▶ `cudnnRNNForwardInferenceEx()`: An extended version of the `cudnnRNNForwardInference()` to allow for the padded (unpacked) layout for the I/O.

  ▶ `cudnnRNNBackwardDataEx()`: An extended version of the `cudnnRNNBackwardData()` to allow for the padded (unpacked) layout for the I/O.

  ▶ `cudnnRNNBackwardWeightsEx()`: An extended version of the `cudnnRNNBackwardWeights()` to allow for the padded (unpacked) layout for the I/O.

▶ Added support for cell clipping in cuDNN LSTM. The following new functions are added:

  ▶ `cudnnRNNSetClip()` and `cudnnRNNGetClip()`: Sets and retrieves, respectively, the LSTM cell clipping mode.

▶ Accelerate your convolution computation with this new feature: When the input channel size `c` is a multiple of 32, you can use the new data type `CUDNN_DATA_INT8x32` to accelerate your convolution computation.

> 🗩 **Note:** This new data type `CUDNN_DATA_INT8x32` is only supported by sm_72.

▶ Enhanced the family of `cudnnFindRNN*` functions. The `findIntensity` input to these functions now enables the user to control the overall runtime of the RNN find algorithms, by selecting a percentage of a large Cartesian product space to be searched.

▶ A new mode `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is added to `cudnnMathType_t`. The computation time for FP32 tensors can be reduced by selecting this mode.

▶ The functions `cudnnRNNForwardInference()`, `cudnnRNNForwardTraining()`, `cudnnRNNBackwardData()`, and `cudnnRNNBackwardWeights()` will now perform down conversion of FP32 I/O only when `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` is set.

▶ Improved the heuristics for `cudnnGet*Algorithm()` functions.

Known Issues and Limitations

Following issues and limitations exist in this release:

▶ For FP16 inputs, the functions `cudnnGetConvolutionForwardAlgorithm()`, `cudnnGetConvolutionBackwardDataAlgorithm()`, and `cudnnGetConvolutionBackwardFilterAlgorithm()` will obtain a slower algorithm.

▶ For cases where `beta` is not equal to zero, and when the input channel size is greater than 65535, then the below `cudnnConvolutionBackwardFilter()` algorithms may return `EXECUTION_FAILED` error:

   ▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0`

   ▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`

   ▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_3`

▶ **This is a rare occurrence:** When `beta` is not equal to zero, the function `cudnnFindConvolutionBackwardFilterAlgorithm()` may not return the fastest algorithm available for `cudnnConvolutionBackwardFilter()`.

▶ Grouped convolutions are not supported in the `TRUE_HALF_CONFIG` (convDesc is `CUDNN_DATA_HALF`) data type configuration. As a workaround, the `PSEUDO_HALF_CONFIG` (convDesc is `CUDNN_DATA_FLOAT`) data type configuration can be used without losing any precision.

▶ For the `cudnnConvolutionBiasActivationForward()` function, if the input `cudnnActivationMode_t` is set to enum value `CUDNN_ACTIVATION_IDENTITY`, then the input `cudnnConvolutionFwdAlgo_t` must be set to the enum value `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`.

▶ When the user runs `cudnnRNNForward*` or `cudnnRNNBackward*` with FP32 I/O, on sm_70 or sm_72, with RNN descriptor's `algo` field set to `CUDNN_RNN_ALGO_PERSIST_STATIC`, and math type set to `CUDNN_TENSOR_OP_MATH` using `cudnnSetRNNMatrixMathType()`, then the results are incorrect.

▶ When the user runs `cudnnRNNForward*` or `cudnnRNNBackward*` with FP32 I/O, on sm_70 or sm_72, with RNN descriptor's `algo` field set to `CUDNN_RNN_ALGO_PERSIST_STATIC`, and math type set to `CUDNN_TENSOR_OP_MATH_ALLOW_CONVERSION` using `cudnnSetRNNMatrixMathType()`, then the resulting performance is suboptimal.

Fixed Issues

The following issues have been fixed in this release:

▶ The `cudnnConvolutionBackwardData()` function produced incorrect result under these conditions:

   ▶ The `algo` input is set to `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1` in `cudnnConvolutionBwdDataAlgo_t`, and

   ▶ `CUDNN_TENSOR_OP_MATH` is selected.

   Under these conditions, the dgrad computation was giving incorrect results when the data is not packed and the data format is NCHW. This is fixed.

▶ When the `cudnnConvolutionFwdAlgo_t()` was set to `CONVOLUTION_FWD_ALGO_FFT_TILING` then the function `cudnnConvolutionForward()` was leading to illegal memory access. This is now fixed.

▶ `cudnnPoolingBackward()` was failing when using a large kernel size used for 'global_pooling' with NHWC I/O layout. This is fixed.

▶ The below two items are fixed: If you set RNN mathtype to `CUDNN_TENSOR_OP_MATH`, and run RNN on **sm6x** or earlier hardware:

   ▶ You may have received `CUDNN_STATUS_NOT_SUPPORTED` when algo selected is `CUDNN_RNN_ALGO_STANDARD` or `CUDNN_RNN_ALGO_PERSIST_STATIC`.

   ▶ You may have received incorrect results when algo selected is `CUDNN_RNN_ALGO_PERSIST_DYNAMIC`.

▶ If you passed in variable sequence length input tensor to `cudnnRNNForwardInference()`, `cudnnRNNForwardTraining()`, `cudnnRNNBackwardData()`, and used `CUDNN_RNN_ALGO_PERSIST_STATIC` or `CUDNN_RNN_ALGO_PERSIST_DYNAMIC`, then you may have received incorrect results. Now this is being checked, and `CUDNN_STATUS_NOT_SUPPORTED` will be returned.

2.14. cuDNN Release 7.1.4

This is the cuDNN 7.1.4 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ Improved performance for some cases of data-gradient convolutions and maxpooling. This is expected to improve performance of ResNet-50 like networks.

▶ The runtime of the RNN Find algorithm suite is improved in v7.1.4 resulting in slightly improved runtime of `cudnnFindRNN***AlgorithmEx`.

Known Issues

Following are known issues in this release:

▶ `cudnnGet` picks a slow algorithm that does not use Tensor Cores on NVIDIA Volta when inputs are FP16 and it is possible to do so.

▶ The `cudnnConvolutionBackwardFilter()` function may output incorrect results for `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING` when the convolution mode is `CUDNN_CONVOLUTION`. This function should not be used in this mode.

Fixed Issues

The following issues have been fixed in this release:

▶ `cudnnAddTensorNd` might cause a segmentation fault if called with bad arguments (for example, null pointer). This issue is in 7.1.3 only and fixed in 7.1.4.

▶ `cudnnRNNBackwardData` LSTM cell with FP16 (half) inputs might generate wrong values (silently). This issue exists in cuDNN 7.1.3 binaries compiled with CUDA Toolkit 9.0 and 9.2. This issue does not exist in cuDNN 7.1.3 binaries compiled with CUDA Toolkit 9.1.

▶ `cudnnGetRNNLinLayerMatrixParams` wrongly returns `CUDNN_STATUS_BAD_PARAM` when `cudnnSetRNNDescriptor` is called with `dataType == CUDNN_DATA_FLOAT`. This is an issue in 7.1.3 only and will be fixed in 7.1.4. The `dataType` argument as of today supports only `CUDNN_DATA_FLOAT`. We plan to support additional compute types in the future.

▶ There is a small memory leak issue when calling `cudnnRNNBackwardData` with `CUDNN_RNN_ALGO_STANDARD`. This issue also affects previous cuDNN v7 releases. This is fixed in 7.1.4.

▶ RNN with half-precision returns `CUDNN_EXECUTION_FAILED` on NVIDIA Kepler GPU in 7.1.3. This is fixed in 7.1.4.

▶ The RNN Find algorithm suite mistakenly did not test `CUDNN_RNN_ALGO_PERSIST_STATIC` and `CUDNN_RNN_ALGO_PERSIST_DYNAMIC` kernels with tensor operations enabled when it was possible to do so. This is fixed in v7.1.4.

2.15.   cuDNN Release 7.1.3

This is the cuDNN 7.1.3 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Known Issues

Following are known issues in this release:

▶ `cudnnGet` picks a slow algorithm that does not use Tensor Cores on NVIDIA Volta when inputs are FP16 and it is possible to do so.

▶ The `cudnnConvolutionBackwardFilter()` function may output incorrect results for `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING` when the convolution mode is `CUDNN_CONVOLUTION` and the product n*k (n - batch size, k - number of output feature maps) is large, that is, several thousand or more. It appears that the `CUDNN_CROSS_CORRELATION` mode is not affected by this bug.

▶ There is a small memory leak issue when calling `cudnnRNNBackwardData` with `CUDNN_RNN_ALGO_STANDARD`. This issue also affects previous cuDNN v7 releases.

▶ RNN with half precision will not work on NVIDIA Kepler GPUs and will return `CUDNN_EXECUTION_FAILED`. This will be fixed in future releases to return `CUDNN_STATUS_UNSUPPORTED`.

Fixed Issues

The following issues have been fixed in this release:

▶ `cudnnRNNbackwardData` for LSTM with recurrent projection in half-precision may fail in rare cases with misaligned memory access on NVIDIA Pascal and Maxwell.

▶ `cudnnRNNbackwardData` for bidirectional LSTM with recurrent projection may produce inaccurate results or `CUDNN_STATUS_UNSUPPORTED`.

▶ Algo 1 for forward convolution and dgrad may produce erroneous results when the filter size is greater than the input size. This issue is fixed in 7.1.3.

▶ For very large RNN networks, the function `cudnnGetRNNWorkspaceSize` and `cudnnGetRNNTrainingReserveSize` may internally overflow and give incorrect results.

▶ The small performance regression on multi-layer RNNs using the STANDARD algorithm and Tensor Core math in 7.1.2, as compared to 7.0.5, is fixed in this release.

▶ Fixed an issue with persistent LSTM backward pass with a hidden state size in the range 257 to 512 on GPUs with number of SMs between 22 and 31 might hang. This issue also exists in 7.1.1. This is fixed in 7.1.3.

▶ Fixed an issue persistent GRU backward pass with a hidden state size in the range 513->720 on GPUs with exactly 30 SMs would hang. This issue also exists in 7.1.1. This is fixed in 7.1.3.

2.16. cuDNN Release 7.1.2

This is the cuDNN 7.1.2 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ RNN search API extended to support all RNN algorithms.

▶ Newly added projection layer supported for inference bidirectional RNN cells and for backward data and gradient.

▶ Support IDENTITY Activation for all `cudnnConvolutionBiasActivationForward` data types for `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`.

▶ Added documentation to clarify RNN/LSTM weight formats.

Known Issues

Following are known issues in this release:

▶ `cudnnGet` picks a slow algorithm that does not use Tensor Cores on NVIDIA Volta when inputs are FP16 and it is possible to do so.

▶ There may be a small performance regression on multi-layer RNNs using the STANDARD algorithm with Tensor Core math in this release compared to v7.0.5.

▶ LSTM projection dgrad half precision may fail in rare cases with misaligned memory access on NVIDIA Pascal and Maxwell.

▶ Dgrad for bidirectional LSTM with projection should not be used, may produce inaccurate results, or `CUDNN_STATUS_UNSUPPORTED`.

▶ The `cudnnConvolutionBackwardFilter()` function may output incorrect results for `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT_TILING` when the convolution mode is `CUDNN_CONVOLUTION` and the product $n*k$ (n - batch size, k - number of output feature maps) is large, that is, several thousand or more. It appears that the `CUDNN_CROSS_CORRELATION` mode is not affected by this.

▶ Persistent LSTM backward passes with a hidden state size in the range 257 to 512 on GPUs with number of SMs between 22 and 31 might hang. This issue also exists in 7.1.1 and will be fixed in 7.1.3.

▶ Persistent GRU backward passes with a hidden state size in the range 513 to 720 on GPUs with exactly 30 SMs would hang. This issue also exists in 7.1.1 and will be fixed in 7.1.3.

▶ Algo 1 for forward convolution and dgrad may produce erroneous results when the filter size is greater than the input size.

Fixed Issues

The following issues have been fixed in this release:

▶ The `uint8` input for convolution is restricted to NVIDIA Volta and later. We added support for older architectures, for algo: `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`.

▶ In some cases when algorithm `CUDNN_CONVOLUTION_BWD_FILTER_ALGO1` was selected, the routine `cudnnConvolutionBackwardFilter` could fail at runtime and return `CUDNN_STATUS_EXECUTION_FAILED`. It now returns `CUDNN_STATUS_NOT_SUPPORTED`.

▶ `cudnnSetRNNDescriptor` no longer needs valid Dropout Descriptor in inference mode, user can pass NULL for Dropout Descriptor in inference mode.

2.17. cuDNN Release 7.1.1

This is the cuDNN 7.1.1 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

The following enhancements have been added to this release:

▶ Added new API `cudnnSetRNNProjectionLayers` and `cudnnGetRNNProjectionLayers` to support Projection Layer for the RNN LSTM cell. In this release, only the inference use case will be supported. The bi-directional and the training forward and backward for training is not supported in 7.1.1 but will be supported in the upcoming 7.1.2 release without API changes. For all the unsupported cases in this release, `CUDNN_NOT_SUPPORTED` is returned when projection layer is set and the RNN is called.

▶ The `cudnnGetRNNLinLayerMatrixParams()` function was enhanced and a bug was fixed without modifying its prototype. Specifically:

   ▶ The `cudnnGetRNNLinLayerMatrixParams()` function was updated to support the RNN projection feature. An extra `linLayerID` value of eight can be used to retrieve the address and the size of the "recurrent" projection weight matrix when "mode" in `cudnnSetRNNDescriptor()` is configured to `CUDNN_LSTM` and the recurrent projection is enabled using `cudnnSetRNNProjectionLayers()`.

   ▶ Instead of reporting the total number of elements in each weight matrix in the `linLayerMatDesc` filter descriptor, the `cudnnGetRNNLinLayerMatrixParams()` function returns the matrix size as two dimensions: rows and columns. This allows the user to easily print and initialize RNN weight matrices. Elements in each weight matrix are arranged in the row-major order. Due to historical reasons, the minimum number of dimensions in the filter descriptor is three. In previous versions of the cuDNN library, `cudnnGetRNNLinLayerMatrixParams()` returned the total number of weights as follows: `filterDimA[0]=total_size, filterDimA[1]=1, filterDimA[2]=1`. In v7.1.1, the format was changed to: `filterDimA[0]=1, filterDimA[1]=rows, filterDimA[2]=columns`. In both cases, the "format" field of the filter descriptor should be ignored when retrieved by `cudnnGetFilterNdDescriptor()`.

   ▶ A bug in `cudnnGetRNNLinLayerMatrixParams()` was fixed to return a zeroed filter descriptor when the corresponding weight matrix does not exist. This occurs, for example, for `linLayerID` values of 0-3 when the first RNN layer is configured to exclude matrix multiplications applied to RNN input data (`inputMode=CUDNN_SKIP_INPUT` in `cudnnSetRNNDescriptor()` specifies implicit, fixed identity weight matrices for RNN input). Such cases in previous versions of the cuDNN library caused `cudnnGetRNNLinLayerMatrixParams()` to return corrupted filter descriptors with some entries from the previous call. A workaround was to create a new filter descriptor for every invocation of `cudnnGetRNNLinLayerMatrixParams()`.

▶ The `cudnnGetRNNLinLayerBiasParams()` function was updated to report the bias column vectors in `linLayerBiasDesc` in the same format as `cudnnGetRNNLinLayerMatrixParams()`. In previous versions of the cuDNN library, `cudnnGetRNNLinLayerBiasParams()` returned the total number of adjustable bias parameters as follows: `filterDimA[0]=total_size, filterDimA[1]=1, filterDimA[2]=1`. In v7.1.1, the format was changed to: `filterDimA[0]=1, filterDimA[1]=rows, filterDimA[2]=1` (number of columns). In both cases, the `format` field of the filter descriptor should be ignored when retrieved by `cudnnGetFilterNdDescriptor()`. The recurrent projection GEMM does not have a bias so the range of valid inputs for the `linLayerID` argument remains the same.

▶ Added support for use of Tensor Core for the `CUDNN_RNN_ALGO_PERSIST_STATIC`. This required cuDNN v7.1 built with CUDA 9.1 and 387 or higher driver. It will not work with CUDA 9.0 and 384 driver.

▶ Added RNN search API that allows the application to provide an RNN descriptor and get a list of possible algorithm choices with performance and memory usage, to allow applications to choose between different implementations. For more information, refer to the documentation of: `cudnnFindRNNForwardInferenceAlgorithmEx`, `cudnnFindRNNForwardTrainingAlgorithmEx`, `cudnnFindRNNBackwardDataAlgorithmEx`, and `cudnnFindRNNBackwardWeightsAlgorithmEx`. In this release, the search will operate on STANDARD algorithm and will not support PERSISTENT algorithms of RNN.

▶ Added `uint8` for support for the input data for `cudnnConvolutionBiasActivationForward` and `cudnnConvolutionForward`. Currently, the support is on NVIDIA Volta (sm 70) and later architectures. Support for older architectures will be gradually added in the upcoming releases.

▶ Support for `CUDNN_ACTIVATION_IDENTITY` is added to `cudnnConvolutionBiasActivationForward`. This allows users to perform Convolution and Bias without Activation.

▶ All API functions now support logging. User can trigger logging by setting environment variable `CUDNN_LOGINFO_DBG=1` and `CUDNN_LOGDEST_DBG= <option>` where `<option>` (that is, the output destination of the log) can be chosen from `stdout`, `stderr`, or a file path. User may also use the new `Set`/`GetCallBack` functions to install their customized callback function. Log files can be added to the reported bugs or shared with us for analysis and future optimizations through partners.nvidia.com.

▶ Improved performance of 3D convolution on NVIDIA Volta architecture.

▶ The following algo-related functions have been added for this release: `cudnnGetAlgorithmSpaceSize, cudnnSaveAlgorithm, cudnnRestoreAlgorithm, cudnnCreateAlgorithmDescriptor, cudnnSetAlgorithmDescriptor, cudnnGetAlgorithmDescriptor, cudnnDestroyAlgorithmDescriptor, cudnnCreateAlgorithmPerformance, cudnnSetAlgorithmPerformance, cudnnGetAlgorithmPerformance, cudnnDestroyAlgorithmPerformance`.

▶ All algorithms for convolutions now support groupCount > 1. This includes `cudnConvolutionForward()`, `cudnnConvolutionBackwardData()`, and `cudnnConvolutionBackwardFilter()`.

Known Issues

Following are known issues in this release:

▶ RNN search Algorithm is restricted to STANDARD algorithm.

▶ Newly added projection Layer supported for inference and one directional RNN cell.

▶ uint8 input for convolution is restricted to NVIDIA Volta and later.

▶ `cudnnGet` picks a slow algorithm that does not use Tensor Cores on NVIDIA Volta when inputs are FP16 and it is possible to do so.

▶ There may be a small performance regression on multi-layer RNNs using the STANDARD algorithm with Tensor Core math in this release compared to 7.0.5.

Fixed Issues

The following issues have been fixed in this release:

▶ 3D convolution performance improvements for NVIDIA Volta.

▶ Added support for Algorithm 0 data gradients to cover cases previously not supported.

▶ Removed the requirement for dropout Descriptor in RNN inference. Before application had to set a non-point for the dropout Descriptor that was not used.

▶ Use of `CUDNN_TENSOR_NCHW_VECT_C` with non-zero padding resulted in a return status of `CUDNN_STATUS_INTERNAL_ERROR`. This issue is now fixed.

2.18.   cuDNN Release 7.0.5

This is the cuDNN 7.0.5 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Known Issues

Following are known issues in this release:

▶ cuDNN library may trigger a CPU floating point exception when FP exceptions are enabled by user. This issue exists for all 7.0.x releases.

▶ There are heavy use cases of RNN layers that might hit a memory allocation issue in the CUDA driver when using cuDNN v7 with CUDA 8.0 and R375 driver on pre-Pascal architectures (NVIDIA Kepler and Maxwell). In these cases, subsequent CUDA kernels may fail to launch with an Error Code 30. To resolve the issue, it is recommended to use the latest R384 driver (from NVIDIA driver downloads) or to ensure that the persistence daemon is started. This behavior is observed on all 7.0.x releases.

▶ When using `TENSOR_OP_MATH` mode with `cudnnConvolutionBiasActivationForward`, the pointer to the bias must be aligned to 16 bytes and the size of allocated memory must be multiples of 256 elements. This behavior exists for all 7.0.x releases.

Fixed Issues

The following issues have been fixed in this release:

▶ Corrected the algorithm fallback behavior in RNN when user set to use `CUDNN_TENSOR_OP_MATH` when using compute card without Tensor Cores. Instead of returning `CUDNN_STATUS_NOT_SUPPORTED`, the RNN algorithm will now continue to run using `CUDNN_DEFAULT_MATH`. The correct behavior is to fall back to using default math when Tensor Core is not supported. Fixed to the expected behavior.

▶ On NVIDIA Volta hardware, `BWD_FILTER_ALGO_1` and `BWD_DATA_ALGO_1` convolutions using a number of filter elements greater than 512 were causing `CUDA_ERROR_ILLEGAL_ADDRESS` and `CUDNN_STATUS_INTERNAL_ERROR` errors. Logic was added to fall back to a generic kernel for these filter sizes.

▶ cuDNN v7 with CUDA 8.0 produced erroneous results on NVIDIA Volta for some common cases of Algo 1. Logic was added to fall back to a generic kernel when cudnn v7 with CUDA 8.0 is used on NVIDIA Volta.

2.19. cuDNN Release 7.0.4

This is the cuDNN 7.0.4 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

Performance improvements for grouped convolutions when input channels and output channels per group are one, two, or four for the following algorithms:

▶ `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`

▶ `CUDNN_CONVOLUTION_BWD_DATA_ALGO0`

▶ `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`

▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_0`

▶ `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`

Known Issues

Following are known issues in this release:

▶ The CUDA 8.0 build of cuDNN may produce incorrect computations when run on NVIDIA Volta.

▶ cuDNN library triggers CPU floating point exception when FP exceptions are enabled by user. This issue exists for all 7.0.x releases.

▶ There are heavy use cases of RNN layers that might hit a memory allocation issue in the CUDA driver when using cuDNN v7 with CUDA 8.0 and R375 driver on pre-Pascal architectures (NVIDIA Kepler and Maxwell). In these cases, subsequent CUDA kernels may fail to launch with an Error Code 30. To resolve the issue, it is recommended to use the latest R384 driver (from NVIDIA driver downloads) or to ensure that the persistence daemon is started. This behavior is observed on all 7.0.x releases.

▶ When using `TENSOR_OP_MATH` mode with `cudnnConvolutionBiasActivationForward`, the pointer to the bias must be aligned to 16 bytes and the size of allocated memory must be multiples of 256 elements. This behavior exists for all 7.0.x releases.

Fixed Issues

The following issues have been fixed in this release:

▶ Fixed out-of-band global memory accesses in the 256-point 1D FFT kernel. The problem-affected convolutions with 1x1 filters and tall but narrow images, for example, 1x500 (WxH). In those cases, the workspace size for the `FFT_TILING` algo was computed incorrectly. There was no error in the FFT kernel.

▶ Eliminated a source of floating point exceptions in the `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED` algorithm. The host code to generate a negative infinity-floating point value was substituted with a different logic. By default, FP exceptions are disabled. However, a user program enabled them by invoking `feenableexcept()`. There are at least two other sources of FP exceptions in the cuDNN library, affecting for example `BATCHNORM_SPATIAL_PERSISTENT`. Those sources of FP exceptions will be eliminated in future releases of the cuDNN library.

2.20.   cuDNN Release 7.0.3

This is the cuDNN 7.0.3 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

Performance improvements for various cases:

▶ Forward-grouped convolutions where input channel per groups is one, two, or four and hardware is NVIDIA Volta or NVIDIA Pascal.

▶ `cudnnTransformTensor()` where input and output tensor is packed.

> 🗨 **Note:** This is an improved fallback, improvements will not be seen in all cases.

Known Issues

The following are known issues in this release:

▶ `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` may cause `CUDA_ERROR_ILLEGAL_ADDRESS`. This issue affects input images of just one pixel in width and certain `n`, `c`, `k`, `h` combinations.

Fixed Issues

The following issues have been fixed in this release:

▶ `AddTensor` and `TensorOp` produce incorrect results for half and INT8 inputs for various use cases.

▶ `cudnnPoolingBackward()` can produce incorrect values for rare cases of non-deterministic MAX pooling with `window_width > 256`. These rare cases are when the maximum element in a window is duplicated horizontally (along width) by a stride of `256*k` for some `k`. The behavior is now fixed to accumulate derivatives for the duplicate that is left most.

▶ `cudnnGetConvolutionForwardWorkspaceSize()` produces incorrect workspace size for algorithm `FFT_TILING` for 1d convolutions. This only occurs for large sized convolutions where intermediate calculations produce values greater than 2^31 (2 to the power of 31).

▶ `CUDNN_STATUS_NOT_SUPPORTED` returned by `cudnnPooling*()` functions for small `x` image (`channels * height * width < 4`).

2.21. cuDNN Release 7.0.2

This is the cuDNN 7.0.2 release notes. This release includes fixes from the previous cuDNN v7.x.x releases as well as the following additional changes.

Key Features and Enhancements

This is a patch release of cuDNN 7.0 and includes bug fixes and performance improvements mainly on NVIDIA Volta.

Algo 1 Convolutions Performance Improvements
Performance improvements were made to `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`, `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1`, and `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`. These improvements consist of new SASS kernels and improved heuristics. The new kernels implement convolutions over various data sizes and tile sizes. The improved heuristics take advantage of these new kernels.

Known Issues

The following are known issues in this release:

▶ `cudnnGetConvolutionForwardWorkspaceSize()` returns overflowed `size_t` value for certain input shape for `CUDNN_CONVOLUTION_*_ALGO_FFT_TILING`.

▶ `cudnnPoolingBackward()` fails for pooling window size > 256.

Fixed Issues

The following issues have been fixed in this release:

▶ Batch Norm `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` might get into race conditions in certain scenarios.

▶ cuDNN convolution layers using `TENSOR_OP_MATH` with FP16 inputs and outputs and FP32 compute will use "round to nearest" mode instead of "round to zero" mode as in 7.0.1. This rounding mode has proven to achieve better results in training.

▶ Fixed synchronization logic in the `CUDNN_CTC_LOSS_ALGO_DETERMINISTIC` algo for CTC. The original code would hang in rare cases.

▶ Convolution algorithms using `TENSOR_OP_MATH` returned a workspace size from `*GetWorkspaceSize()` smaller than actually necessary.

▶ The results of INT8 are inaccurate in certain cases when calling `cudnnConvolutionForward()` in convolution layer.

▶ `cudnnConvolutionForward()` called with `xDesc`'s `channel = yDesc`'s `channel = groupCount` could compute incorrect values when vertical padding > 0.

2.22.   cuDNN Release 7.0.1

This is the cuDNN 7.0.1 release notes. This release includes the following changes.

cuDNN v7.0.1 is the first release to support the NVIDIA Volta GPU architecture. In addition, cuDNN v7.0.1 brings new layers, grouped convolutions, and improved convolution find as error query mechanism.

Key Features and Enhancements

This cuDNN release includes the following key features and enhancements.

Tensor Cores
Version 7.0.1 of cuDNN is the first to support the Tensor Core operations in its implementation. Tensor Cores provide highly optimized matrix multiplication building blocks that do not have an equivalent numerical behavior in the traditional instructions, therefore, its numerical behavior is slightly different.

`cudnnSetConvolutionMathType, cudnnSetRNNMatrixMathType,` and `cudnnMathType_t`
The `cudnnSetConvolutionMathType` and `cudnnSetRNNMatrixMathType` functions enable you to choose whether or not to use Tensor Core operations in the convolution and RNN layers respectively by setting the math mode to either `CUDNN_TENSOR_OP_MATH` or `CUDNN_DEFAULT_MATH`.

Tensor Core 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.

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 whereas, the Tensor Core is a parallelized operation, therefore, the two might result in slightly 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.

cudnnSetConvolutionGroupCount

A new interface that allows applications to perform convolution groups in the convolution layers in a single API call.

cudnnCTCLoss

`cudnnCTCLoss` provides a GPU implementation of the Connectionist Temporal Classification (CTC) loss function for RNNs. The CTC loss function is used for phoneme recognition in speech and handwriting recognition.

CUDNN_BATCHNORM_SPATIAL_PERSISTENT

The `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` function is a new batch normalization mode for `cudnnBatchNormalizationForwardTraining` and `cudnnBatchNormalizationBackward`. This mode is similar to `CUDNN_BATCHNORM_SPATIAL`, however, it can be faster for some tasks.

cudnnQueryRuntimeError

The `cudnnQueryRuntimeError` function reports error codes written by GPU kernels when executing `cudnnBatchNormalizationForwardTraining` and `cudnnBatchNormalizationBackward` with the `CUDNN_BATCHNORM_SPATIAL_PERSISTENT` mode.

cudnnGetConvolutionForwardAlgorithm_v7

This new API returns all algorithms sorted by expected performance (using internal heuristics). These algorithms are output similarly to `cudnnFindConvolutionForwardAlgorithm`.

cudnnGetConvolutionBackwardDataAlgorithm_v7

This new API returns all algorithms sorted by expected performance (using internal heuristics). These algorithms are output similarly to `cudnnFindConvolutionBackwardAlgorithm`.

cudnnGetConvolutionBackwardFilterAlgorithm_v7

This new API returns all algorithms sorted by expected performance (using internal heuristics). These algorithms are output similarly to `cudnnFindConvolutionBackwardFilterAlgorithm`.

CUDNN_REDUCE_TENSOR_MUL_NO_ZEROS

The `MUL_NO_ZEROS` function is a multiplication reduction that ignores zeros in the data.

CUDNN_OP_TENSOR_NOT

The `OP_TENSOR_NOT` function is a unary operation that takes the negative of (alpha*A).

cudnnGetDropoutDescriptor

The `cudnnGetDropoutDescriptor` function allows applications to get dropout values.

Using cuDNN v7.0.1

Ensure you are familiar with the following notes when using this release.

- Multi-threading behavior has been modified. Multi-threading is allowed only when using different cuDNN handles in different threads.

- In `cudnnConvolutionBackwardFilter`, dilated convolution did not support cases where the product of all filter dimensions was odd for half precision-floating point. These are now supported by `CUDNN_CONVOLUTION_BWD_FILTER_ALGO1`.

- Fixed bug that produced a silent computation error for when a batch size was larger than 65536 for `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_PRECOMP_GEMM`.

- In `getConvolutionForwardAlgorithm`, an error was not correctly reported in v5 when the output size was larger than expected. In v6 the `CUDNN_STATUS_NOT_SUPPORTED`, error message displayed. In v7, this error is modified to `CUDNN_STATUS_BAD_PARAM`.

- In `cudnnConvolutionBackwardFilter`, cuDNN now runs some exceptional cases correctly where it previously erroneously returned `CUDNN_STATUS_NOT_SUPPORTED`. This impacted the algorithms `CUDNN_CONVOLUTION_BWD_FILTER_ALGO0` and `CUDNN_CONVOLUTION_BWD_FILTER_ALGO3`.

Deprecated Features

The following routines have been removed:

- `cudnnSetConvolution2dDescriptor_v4`
- `cudnnSetConvolution2dDescriptor_v5`
- `cudnnGetConvolution2dDescriptor_v4`
- `cudnnGetConvolution2dDescriptor_v5`

> 🗨 **Note:** Only the non-suffixed versions of these routines remain.

The following routines have been created and have the same API prototype as their non-suffixed equivalent from cuDNN v6:

▶ `cudnnSetRNNDescriptor_v5` - The non-suffixed version of the routines in cuDNN v7.0.1 are now mapped to their `_v6` equivalent.

> ⚠ **ATTENTION:** It is strongly advised using the non-suffixed version as the `_v5` and `_v6` routines will be removed in the next cuDNN release.

▶ `cudnnGetConvolutionForwardAlgorithm`, `cudnnGetConvolutionBackwardDataAlgorithm`, and `cudnnGetConvolutionBackwardFilterAlgorithm` - A `_v7` version of this routine has been created. For more information, see the *Backward compatibility and deprecation policy* chapter of the cuDNN documentation for details.

Known Issues

▶ cuDNN pooling backwards fails for pooling window size > 256.

Trademarks

NVIDIA, the NVIDIA logo, and CUDA, DRIVE, JetPack, Kepler, Maxwell, Pascal, Turing, Volta and Xavier 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

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com