NVIDIA CUDA TOOLKIT 8.0

RN-06722-001 _v8.0 | May 2016

Release Notes for Windows, Linux, and Mac OS

TABLE OF CONTENTS

Chapter 1. CUDA Toolkit Major Components 1
Chapter 2. New Features 3
2.1. General CUDA 3
2.2. CUDA Tools 3
2.2.1. CUDA Compilers 3
2.2.2. CUDA Profiler 5
2.2.3. NVIDIA Visual Profiler 5
2.3. CUDA Libraries 5
2.3.1. cuBLAS Library 5
2.3.2. cuFFT Library 6
2.3.3. CUDA Math Library 6
2.3.4. CUDA nvGRAPH Library 6
2.4. CUDA Samples 6
Chapter 3. Unsupported Features 7
Chapter 4. Deprecated Features 8
Chapter 5. Performance Improvements 9
5.1. CUDA Tools 9
5.1.1. CUDA Compilers 9
5.2. CUDA Libraries 9
5.2.1. cuBLAS Library 9
5.2.2. CUDA Math Library 9
Chapter 6. Resolved Issues 10
6.1. CUDA Tools 10
6.1.1. CUDA Compilers 10
Chapter 7. Known Issues 11
7.1. General CUDA 11

Chapter 1. CUDA TOOLKIT MAJOR COMPONENTS

This section provides an overview of the major components of the CUDA Toolkit and points to their locations after installation.

Compiler

The CUDA-C and CUDA-C++ compiler, **`nvcc`**, is found in the **`bin/`** directory. It is built on top of the NVVM optimizer, which is itself built on top of the LLVM compiler infrastructure. Developers who want to target NVVM directly can do so using the Compiler SDK, which is available in the **`nvvm/`** directory.

Tools

The following development tools are available in the **`bin/`** directory (except for Nsight Visual Studio Edition (VSE) which is installed as a plug-in to Microsoft Visual Studio).

- IDEs: **`nsight`** (Linux, Mac), Nsight VSE (Windows)
- Debuggers: **`cuda-memcheck`**, **`cuda-gdb`** (Linux, Mac), Nsight VSE (Windows)
- Profilers: **`nvprof`**, **`nvvp`**, Nsight VSE (Windows)
- Utilities: **`cuobjdump`**, **`nvdisasm`**, **`gwiz`**

Libraries

The scientific and utility libraries listed below are available in the **`lib/`** directory (DLLs on Windows are in **`bin/`**), and their interfaces are available in the **`include/`** directory.

- **`cublas`** (BLAS)
- **`cublas_device`** (BLAS Kernel Interface)
- **`cuda_occupancy`** (Kernel Occupancy Calculation [header file implementation])
- **`cudadevrt`** (CUDA Device Runtime)
- **`cudart`** (CUDA Runtime)
- **`cufft`** (Fast Fourier Transform [FFT])
- **`cupti`** (Profiling Tools Interface)
- **`curand`** (Random Number Generation)
- **`cusolver`** (Dense and Sparse Direct Linear Solvers and Eigen Solvers)
- **`cusparse`** (Sparse Matrix)
- **`npp`** (NVIDIA Performance Primitives [image and signal processing])
- **`nvblas`** ("Drop-in" BLAS)

- **nvcuvid** (CUDA Video Decoder [Windows, Linux])
- **nvgraph** (CUDA nvGRAPH [accelerated graph analytics])
- **nvrtc** (CUDA Runtime Compilation)
- **thrust** (Parallel Algorithm Library [header file implementation])

CUDA Samples

Code samples that illustrate how to use various CUDA and library APIs are available in the **samples/** directory on Linux and Mac, and are installed to **C:\ProgramData\NVIDIA Corporation\CUDA Samples** on Windows. On Linux and Mac, the **samples/** directory is read-only and the samples must be copied to another location if they are to be modified. Further instructions can be found in the *Getting Started Guides* for Linux and Mac.

Documentation

The most current version of these release notes can be found online at http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html. Also, the **version.txt** file in the root directory of the toolkit will contain the version and build number of the installed toolkit.

Documentation can be found in PDF form in the **doc/pdf/** directory, or in HTML form at **doc/html/index.html** and online at http://docs.nvidia.com/cuda/index.html.

CUDA-GDB Sources

CUDA-GDB sources are available as follows:

- For CUDA Toolkit 7.0 and newer, in the installation directory **extras/**. The directory is created by default during the toolkit installation unless the **.rpm** or **.deb** package installer is used. In this case, the **cuda-gdb-src** package must be manually installed.
- For CUDA Toolkit 6.5, 6.0, and 5.5, at https://github.com/NVIDIA/cuda-gdb.
- For CUDA Toolkit 5.0 and earlier, at ftp://download.nvidia.com/CUDAOpen64/.
- Upon request by sending an e-mail to mailto:oss-requests@nvidia.com.

Chapter 2. NEW FEATURES

2.1. General CUDA

- CUDA 8.0 adds support for GPUDirect Async, which improves application throughput by eliminating the CPU as the critical path in GPU-initiated data transfers. The GPU now directly triggers data transfers without CPU coordination, unblocking the CPU to perform other tasks.
- The NVLink high-speed interconnect is now supported.
- Added new RPM packages that help automate the deployment of CUDA installations in large-scale cluster environments, using tools such as Puppet.
- Added absolute GPU numbering in NVML and NVIDIA-SMI that is based on the order of the GPUs on the PCI bus, so that the numbering matches the **`/dev/nvidiaX`** indices.
- Unified Memory is now supported with OS X 10.11 for Mac.

2.2. CUDA Tools

2.2.1. CUDA Compilers

- Intel C++ Compilers 16.0 and 15.0.4 are now supported.
- POWER8 IBM XL compiler 13.1.3 is now supported.
- The clang release 3.7 LLVM-based C and C++ compilers are now supported as host compilers by **`nvcc`** on Linux operating systems.
- The **`-std=c++11`** option for **`nvcc`** is now supported when IBM xlC compiler version 13.1 (and above) is used as the host compiler.
- The **`-std=c++11`** option for **`nvcc`** is now supported when Intel ICC compiler 15 (and above) is used as the host compiler. Note that the CUDA extended lambda feature is not supported with the Intel ICC compiler.
- For debug compilations, the compiler now maintains the live ranges of variables that are in-scope according to C language rules. This feature allows users to inspect

variables they expect to be live and reduces "value optimized out" errors during debugging.

- Improved 32-bit overflow checking when detecting common sub-expressions in array index operations.
- Improved the loop unroller with respect to nested loops. There are cases when the inner loop's trip count may depend on the outer loop's induction variable; after the outer loop is fully unrolled, the inner loop's trip count may become compile-time constants and thus become candidates for complete unrolling.
- Improved loop unrolling in the presence of **unroll pragma** information.
- The argument to the **unroll pragma** is now allowed to be any integral constant expression (as defined by the C++ standard). This includes integral template arguments and **constexpr** function calls (in modes where **constexpr** is enabled).
- The **nvcc** compiler defines the macros **__CUDACC_EXTENDED_LAMBDA__** and **__CUDACC_RELAXED_CONSTEXPR__** when the **--expt-extended-lambda** and **--expt-relaxed-constexpr** flags are specified, respectively.
- For the function **nvrtcCreateProgram()**, the type of the **headers** and **includeNames** parameters has been changed from **const char **** to **const char * const ***. For the function **nvrtcCompileProgram()**, the type of the **options** parameter has been changed from **const char **** to **const char * const ***. These changes are intended to facilitate easier use of the NVRTC API, such as using the pointer returned by the **std::initializer_list<const char *>::begin()** function.
- To reduce compile time, the compiler may remove unused **__device__** functions before generating PTX in whole-program compilation mode. The unused **__device__** functions are not removed when compiling in debug mode (**-G**) or in separate compilation mode. Note that a **__device__** function is considered unused if it has been fully inlined into its callers and has no other references.
- Within the body of a **__device__**, **__global__**, or **__device__ __host__** function, variables without any device memory qualifiers can be declared as static storage. They have the same restrictions as **__device__** variables defined in namespace scope.
- The **nvstd::function** implementation has been enhanced to allow **operator()** to be invoked from **__host__** and **__host__ __device__** functions as well as from **__device__** functions. The intent is to allow the **nvstd::function** to be usable in both host and device code. Please see the "C/C++ Language Support" section of the *CUDA Programming Guide* for more information.
- The compiler now supports instantiating **__global__** function templates with closure types of extended **__host__ __device__** lambdas defined in host code. This functionality is enabled when the **--expt-extended-lambda** flag is passed to **nvcc**. Please see the "C/C++ Language Support" section of the *CUDA Programming Guide* for more information.
- The compiler now provides the following type traits to detect closure types of extended **__device__** and extended **__host__ __device__** lambdas:
  - **__nv_is_extended_device_lambda_closure_type(T)**
  - **__nv_is_extended_host_device_lambda_closure_type(T)**

Please see the "C/C++ Language Support" section of the *CUDA Programming Guide* for more information.

- The NVRTC API has been enhanced for easier integration with templated host code. The following functions have been added:
  - **`nvrtcAddNameExpression()`** and **`nvrtcGetLoweredName()`**. This pair of functions can be used to extract the mangled (lowered) names of **`__global__`** functions and function template instantiations, given high-level source expressions denoting the addresses of the functions. This is useful in using the CUDA Driver API to look up the kernel functions in the generated PTX.
  - **`template <typename T> nvrtcResult nvrtcGetTypeName()`**. This function uses host platform mechanisms such as **`abi::__cxa_demangle()`** to extract the string name corresponding to the type argument **`T`**. The type name string can be incorporated into a **`__global__`** template instantiationin the NVRTC source string, thus allowing templated host code functions to create customized **`__global__`** template instantiations at run time.

  Please see the NVRTC documentation for details and runnable examples.
- The limit on the number of variables that can be captured by extended lambdas has been increased from 30 to 1023.

2.2.2. CUDA Profiler

- CUDA 8.0 provides CPU profiling to identify hot-spot regions in the code, along with call-graph and source-level drill-down.
- The dependency analysis feature enables optimization of the program runtime and the concurrency of applications using multiple CPU threads and CUDA streams. It allows computing the critical path of a specific execution, detecting waiting time, and inspecting dependencies among activities executing in different threads or streams.
- Enabled support for mixed precision (**`fp16`**) in the CUDA debugger and profiler.
- Enabled profiling of NVLink, including topology, bandwidth, and throughput.

2.2.3. NVIDIA Visual Profiler

- CUDA 8.0 adds tools support for the profiling of OpenACC applications.

2.3. CUDA Libraries

2.3.1. cuBLAS Library

- The cuBLAS library now supports a Gaussian implementation for the GEMM, SYRK, and HERK operations on complex matrices.
- New routines for batched GEMMs, **`cublas<T>gemmStridedBatch()`**, have been added. These routines implement a new batch API for GEMMs that is easier to set up. The routines are optimized for performance on GPU architectures **`sm_5x`** or greater.

- The cublasXt API now accepts matrices that are resident in GPU memory.

2.3.2. cuFFT Library

- The cuFFT library now includes half-precision floating point datatype (**fp16**) FFT functions for transform sizes that are powers of two. For FFT size 2, real-to-complex and complex-to-real transforms in **fp16** are currently not supported.
- Improvements were made to cuFFT to take advantage of multi-GPU configurations. The cuFFT library was also optimized for different NVLink topologies.
- In cuFFT 8.0, compatibility modes different from **CUFFT_COMPATIBILITY_FFT_PADDING** are no longer supported. Function **cufftSetCompatibilityMode()** no longer accepts the following values for the mode parameter: **CUFFT_COMPATIBILITY_NATIVE**, **CUFFT_COMPATIBILITY_FFTW_ALL**, and **CUFFT_COMPATIBILITY_FFT_ASYMMETRIC**. The error code **CUFFT_NOT_SUPPORTED** is returned in each case.

2.3.3. CUDA Math Library

- Support for half-precision floating point (**fp16**) has been added to the CUDA math library.
- CUDA 8.0 introduces a new built-in for **fp64 atomicAdd()**. Note that this built-in cannot be overridden with a custom function declared by the user (even if the code is not specifically being compiled for targets other than **sm_60**).

2.3.4. CUDA nvGRAPH Library

- CUDA 8.0 introduces nvGRAPH, a new library that is a collection of routines to process graph problems on GPUs. It includes implementations of the semi-ring SPMV, single source shortest path (SSSP), Widest Path, and PageRank algorithms. The nvGraph library will undergo some changes for the CUDA 8.0 final release:
  - The **naga.h** header will be renamed **nvgraph.h**.
  - Library names (***.so, *.a, *.dll**) will be changed from **libnaga*** to **libnvgraph***.
  - The signature of **nvgraphGetGraphStructure()** will be changed to

    ```
    nvgraphStatus_t NVGRAPH_API nvgraphGetGraphStructure (
        nvgraphHandle_t handle, nvgraphGraphDescr_t descrG,
        void* topologyData, nvgraphTopologyType_t* TType);
    ```

    Its functionality will be updated to return more information to the user.

2.4. CUDA Samples

- CUDA samples were added to illustrate usage of the cuSOLVER library.

Chapter 3. UNSUPPORTED FEATURES

The following features are officially unsupported in the current release. Developers must employ alternative solutions to these features in their software.

General CUDA

- **cuFFT Compatibility Modes.** Compatibility modes different from `CUFFT_COMPATIBILITY_FFT_PADDING` are no longer supported.

CUDA Tools

- **Legacy Command Line Profiler.** The legacy Command Line Profiler was deprecated in CUDA 7.0 and is now removed in CUDA Toolkit 8.0. This notice only applies to the legacy Command Line Profiler controlled by the `COMPUTE_PROFILE` environment variable; there is no impact on the newer `nvprof` profiler, which is also controlled from the command line, or on the Visual Profiler.

Chapter 4. DEPRECATED FEATURES

The following features are deprecated in the current release of the CUDA software. The features still work in the current release, but their documentation may have been removed, and they will become officially unsupported in a future release. We recommend that developers employ alternative solutions to these features in their software.

General CUDA

- **Redundant Device Functions.** Five redundant device functions —`synchthreads()`, `trap()`, `brkpt()`, `prof_trigger()`, and `threadfence()`— have been deprecated because their respective functionalites are identical to `__syncthreads()`, `__trap()`, `__brkpt()`, `__prof_trigger()`, and `__threadfence()`.
- **Fermi Architecture Support.** Fermi architecture support is being deprecated in the CUDA 8.0 toolkit, which will be the last toolkit release to support it. Future versions of the CUDA toolkit will not support the architecture and are not guaranteed to work on that platform.
- **Windows Server 2008 R2 Support.** Support for Windows Server 2008 R2 is now deprecated and will be removed in a future version of the CUDA toolkit.

Chapter 5. PERFORMANCE IMPROVEMENTS

5.1. CUDA Tools

5.1.1. CUDA Compilers

- In CUDA 8.0, some 64-bit integer divisions by zero are converted to 32-bit divisions to improve performance. A CUDA 8.0 result may now differ from a CUDA 7.5 one and is still undefined.
- The performance of single-precision square root (**`sqrt`**), single-precision reciprocal (**`rcp`**), and double-precision division (**`div`**) has nearly doubled.

5.2. CUDA Libraries

5.2.1. cuBLAS Library

- The cuBLAS library now supports high-performance SGEMM routines on Maxwell for handling problem sizes where **`m`** and **`n`** are not necessarily a multiple of the computation tile size. This leads to much smoother and more predictable performance.

5.2.2. CUDA Math Library

- Performance for more than 15 double-precision instructions was improved, with the most significant improvements in division, exponents, and logarithms. Performance and accuracy were improved for following single-precision functions: **`log1pf()`**, **`log2f()`**, **`logf()`**, **`acoshf()`**, **`asinhf()`**, and **`atanhf()`**.

Chapter 6. RESOLVED ISSUES

6.1. CUDA Tools

6.1.1. CUDA Compilers

- The alignment of the built-in **`long2`** and **`ulong2`** types on 64-bit Linux and Mac OS X systems has been changed to 16 bytes (from 8 bytes). It now matches the alignment computed when compiling CUDA code with **`nvcc`** on these systems.
- When C++11 code (**`-std=c++11`**) is compiled on Linux with **`gcc`** as the host compiler, invoking **`pow()`** or **`std::pow()`** from device code with **`(float, int)`** or **`(double, int)`** arguments now compiles successfully.

Chapter 7. KNOWN ISSUES

7.1. General CUDA

- Installation of CUDA 8.0 on an Ubuntu 14.04.4 system requires a system reboot to avoid an **`nvidia-nvlink`** error and system sluggishness.
- Enabling per-thread synchronization behavior [http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html#stream-sync-behavior__per-thread-default-stream] does not work correctly with the following CUDA runtime routines, which use the legacy synchronization behavior [http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html#stream-sync-behavior__legacy-default-stream]:

  - **`cudaMemcpyPeer()`**
  - **`cudaMemcpyPeerAsync()`** with a **`NULL`** stream argument
  - **`cudaGraphicsMapResources()`** with a **`NULL`** stream argument
  - **`cudaGraphicsUnmapResources()`** with a **`NULL`** stream argument

  To work around this issue, use **`cudaMemcpyPeerAsync()`** with the **`cudaPerThreadStream`** stream argument instead of **`cudaMemcpyPeer()`**, and pass the **`cudaPerThreadStream`** argument instead of **`NULL`** to the other routines.
- If the Windows toolkit installation fails, it may be because Visual Studio, **`Nvda.Launcher.exe`**, **`Nsight.Monitor.exe`**, or **`Nvda.CrashReporter.exe`** is running. Make sure these programs are closed and try to install again.
- Peer access is disabled between two devices if either of them is in SLI mode.
- Unified memory is not currently supported with IOMMU. The workaround is to disable IOMMU in the BIOS. Please refer to the vendor documentation for the steps to disable it in the BIOS.

Acknowledgment

NVIDIA extends thanks to Professor Mike Giles of Oxford University for providing the initial code for the optimized version of the device implementation of the double-precision **exp()** function found in this release of the CUDA toolkit.

Notice

Trademarks

Copyright