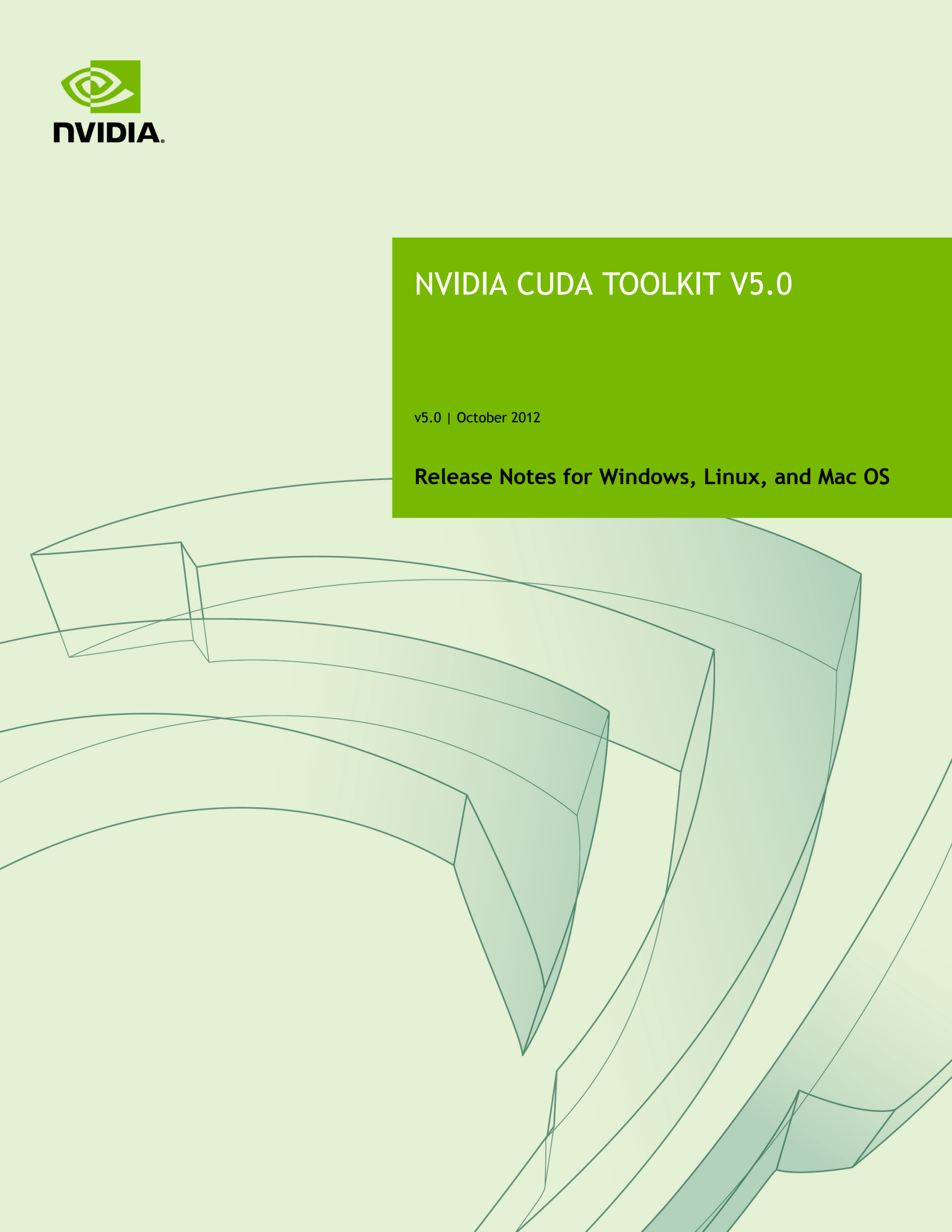

NVIDIA CUDA TOOLKIT V5.0

v5.0 | October 2012

Release Notes for Windows, Linux, and Mac OS

REVISION HISTORY

- 10/2012 Version 5.0
- 08/2012 Version 5.0 RC
- 05/2012 Version 5.0 EA/Preview
- 04/2012 Version 4.2
- 01/2012 Version 4.1 Production
- 11/2011 Version 4.1 RC2
- 10/2011 Version 4.1 RC1
- 09/2011 Version 4.1 EA (Information in ReadMe.txt)
- 05/2011 Version 4.0
- 04/2011 Version 4.0 RC2 (Errata)
- 02/2011 Version 4.0 RC
- 11/2010 Version 3.2
- 10/2010 Version 3.2 RC2
- 09/2010 Version 3.2 RC

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1. RELEASE HIGHLIGHTS

- CUDA Dynamic Parallelism allows `__global__` and `__device__` functions running on the GPU to launch kernels using the familiar `<<< >>>` syntax and to directly call CUDA Runtime API routines (previously this ability was only available from `__host__` functions).
- All `__device__` functions can now be separately compiled and linked using `NVCC`. This allows creation of closed-source static libraries of `__device__` functions and the ability for these libraries to call user-defined `__device__` callback functions. The linker support is considered to be a BETA feature in this release.
- Nsight Eclipse Edition for Linux and Mac OS is an integrated development environment UI that allows developing, debugging, and optimizing CUDA code.
- A new command-line profiler, `nvprof`, provides summary information about where applications spend the most time, so that optimization efforts can be properly ocused.
- See also the New Features section of this document.
- This release contains the following:

  NVIDIA CUDA Toolkit documentation
  NVIDIA CUDA compiler (`nvcc`) and supporting tools
  NVIDIA CUDA runtime libraries
  NVIDIA CUDA-GDB debugger
  NVIDIA CUDA-MEMCHECK
  NVIDIA Visual Profiler, `nvprof`, and command-line profiler
  NVIDIA Nsight Eclipse Edition
  NVIDIA CUBLAS, CUFFT, CUSPARSE, CURAND, Thrust, and
  NVIDIA Performance Primitives (NPP) libraries

Chapter 2. DOCUMENTATION

- For a list of documents supplied with this release, please refer to the `doc` directory of your CUDA Toolkit installation. PDF documents are available in the `doc/pdf` folder. Several documents are now also available in HTML format and are found in the `doc/html` folder.
- The HTML documentation is now fully available from a single entry page available both locally in the CUDA Toolkit installation folder under `doc/html/index.html` and online at http://docs.nvidia.com/cuda/html/index.html.
- The license information for the toolkit portion of this release can be found at `doc/EULA.txt`.
- The CUDA Occupancy Calculator spreadsheet can be found at `tools/CUDA_Occupancy_Calculator.xls`.
- The CHM documentation has been removed.

Chapter 3. LIST OF IMPORTANT FILES

3.1 Core Files

```
bin/
    nvcc                        CUDA C/C++ compiler
    cuda-gdb                    CUDA Debugger
    cuda-memcheck               CUDA Memory Checker
    nsight                      Nsight Eclipse Edition (Linux and Mac OS)
    nvprof                      NVIDIA Command-Line Profiler
    nvvp                        NVIDIA Visual Profiler (Located in libnvvp/
                                                        on Windows)

include/
    cuda.h                      CUDA driver API header
    cudaGL.h                    CUDA OpenGL interop header for driver API
    cudaVDPAU.h                 CUDA VDPAU interop header for driver API
                                                                (Linux)
    cuda_gl_interop.h           CUDA OpenGL interop header for toolkit API
                                                                (Linux)
    cuda_vdpau_interop.h        CUDA VDPAU interop header for toolkit API
                                                                (Linux)
    cudaD3D9.h                  CUDA DirectX 9 interop header  (Windows)
    cudaD3D10.h                 CUDA DirectX 10 interop header (Windows)
    cudaD3D11.h                 CUDA DirectX 11 interop header (Windows)
    cufft.h                     CUFFT API header
    cublas_v2.h                 CUBLAS API header
    cublas.h                    CUBLAS Legacy API header
    cusparse_v2.h               CUSPARSE API header
    cusparse.h                  CUSPARSE Legacy API header
    curand.h                    CURAND API header
    curand_kernel.h             CURAND device API header
    thrust/*                    Thrust headers
    npp.h                       NPP API header
    nvToolsExt*.h               NVIDIA Tools Extension headers (Linux and Mac)
    nvcuvid.h                   CUDA Video Decoder header (Windows and Linux)
    cuviddec.h                  CUDA Video Decoder header (Windows and Linux)
    NVEncodeDataTypes.h         CUDA Video Encoder header (Windows; C-library
                                                          or DirectShow)
    NVEncoderAPI.h              CUDA Video Encoder header (Windows; C-library)
    INvTranscodeFilterGUIDs.h   CUDA Video Encoder header (Windows; DirectShow)
    INVVESetting.h              CUDA Video Encoder header (Windows; DirectShow)

extras/
    CUPTI                       CUDA Performance Tool Interface API
    Debugger                    CUDA Debugger API
```

```
src/
    *fortran*.{c,h}              FORTRAN interface files for CUBLAS and CUSPARSE
```

3.2 Windows lib Files

(Corresponding 32-bit or 64-bit DLLs are in `bin/`.)

```
lib/{Win32,x64}/
    cuda.lib                     CUDA driver library
    cudart.lib                   CUDA runtime library
    cudadevrt.lib                CUDA runtime device library
    cublas.lib                   CUDA BLAS library
    cublas_device.lib            CUDA BLAS device library
    cufft.lib                    CUDA FFT library
    cusparse.lib                 CUDA Sparse Matrix library
    curand.lib                   CUDA Random Number Generation library
    npp.lib                      NVIDIA Performance Primitives library
    nvcuvenc.lib                 CUDA Video Encoder library
    nvcuvid.lib                  CUDA High-level Video Decoder library

    OpenCL.lib                   OpenCL library
```

3.3 Linux lib Files

```
lib{64}/
    libcudart.so                 CUDA runtime library
    libcuinj.so                  CUDA internal library for profiling
    libcublas.so                 CUDA BLAS library
    libcublas_device.a           CUDA BLAS device library
    libcufft.so                  CUDA FFT library
    libcusparse.so               CUDA Sparse Matrix library
    libcurand.so                 CUDA Random Number Generation library
    libnpp.so                    NVIDIA Performance Primitives library
```

3.4 Mac OS X lib Files

```
lib/
    libcudart.dylib              CUDA runtime library
    libcuinj.dylib               CUDA internal library for profiling
    libcublas.dylib              CUDA BLAS library
    libcublas_device.a           CUDA BLAS device library
    libcufft.dylib               CUDA FFT library
    libcusparse.dylib            CUDA Sparse Matrix library
    libcurand.dylib              CUDA Random Number Generation library
    libnpp.dylib                 NVIDIA Performance Primitives library
    libtlshook.dylib             NVIDIA internal library
```

Chapter 4.
SUPPORTED NVIDIA HARDWARE

See http://www.nvidia.com/object/cuda_gpus.html.

Chapter 5. SUPPORTED OPERATING SYSTEMS

5.1 Windows

- Supported Windows Operating Systems (32-bit and 64-bit)

  Windows 8
  Windows 7
  Windows Vista
  Windows XP
  Windows Server 2012
  Windows Server 2008 R2

Table 1 Supported Windows Compilers (32-bit and 64-bit)

Compiler	IDE
Visual C++ 10.0	Visual Studio 2010
Visual C++ 9.0	Visual Studio 2008

5.2 Linux

- The CUDA development environment relies on tight integration with the host development environment, including the host compiler and C runtime libraries, and is therefore only supported on distribution versions that have been qualified for this CUDA Toolkit release.

Table 2 Distributions Currently Supported

Distribution	32	64	Kernel	GCC	GLIBC
Fedora 16	x	x	3.1.0-7.fc16	4.6.2	2.14.90
ICC Compiler 12.x		x			

Distribution	32	64	Kernel	GCC	GLIBC
OpenSUSE 12.1		x	3.1.0-1.2-desktop	4.6.2	2.14.1
Red Hat RHEL 6.x		x	2.6.32-131.0.15.el6	4.4.5	2.12
Red Hat RHEL 5.5+		x	2.6.18-238.el5	4.1.2	2.5
SUSE SLES 11 SP2		x	3.0.13-0.27-pae	4.3.4	2.11.3
SUSE SLES 11.1	x	x	2.6.32.12-0.7-pae	4.3.4	2.11.1
Ubuntu 11.10	x	x	3.0.0-19-generic-pae	4.6.1	2.13
Ubuntu 10.04	x	x	2.6.35-23-generic	4.4.5	2.12.1

Table 3 Distributions No Longer Supported

Distribution	32	64	Kernel	GCC	GLIBC
Fedora 14	x	x	2.6.35.6-45	4.5.1	2.12.90
ICC Compiler 11.1		x			
OpenSUSE 11.2	x	x	2.6.31.5-0.1	4.4.1	2.10.1
Red Hat RHEL 6.x	x		2.6.32-131.0.15.el6	4.4.5	2.12
Red Hat RHEL 5.5+	x		2.6.18-238.el5	4.1.2	2.5
Ubuntu 11.04	x	x	2.6.38-8-generic	4.5.2	2.13

5.3 Mac OS X

- Supported Mac Operating Systems

  Mac OS X 10.8.x
  Mac OS X 10.7.x

Chapter 6. INSTALLATION NOTES

6.1 Windows

For silent installation:

- To install, use `msiexec.exe` from the shell, passing these arguments:

```
msiexec.exe /i <cuda_toolkit_filename>.msi /qn
```

- To uninstall, use `/x` instead of `/i`.

6.2 Linux

- In order to run CUDA applications, the CUDA module must be loaded and the entries in `/dev` created. This may be achieved by initializing X Windows, or by creating a script to load the kernel module and create the entries. An example script (to be run at boot time) follows.

```
#!/bin/bash

/sbin/modprobe nvidia

if [ "$?" -eq 0 ]; then

    # Count the number of NVIDIA controllers found.
    N3D=`/sbin/lspci | grep -i NVIDIA | grep "3D controller" | wc -l`
    NVGA=`/sbin/lspci | grep -i NVIDIA | grep "VGA compatible controller" \
                      | wc -l`

    N=`expr $N3D + $NVGA - 1`
    for i in `seq 0 $N`; do
        mknod -m 666 /dev/nvidia$i c 195 $i;
    done

    mknod -m 666 /dev/nvidiactl c 195 255

else
    exit 1
fi
```

- On some Linux releases, due to a GRUB bug in the handling of upper memory and a default `vmalloc` too small on 32-bit systems, it may be necessary to pass this information to the bootloader:

  ```
  vmalloc=256MB, uppermem=524288
  ```

  Here is an example of GRUB conf:

  ```
  title Red Hat Desktop (2.6.9-42.ELsmp)
  root (hd0,0)
  uppermem 524288
  kernel /vmlinuz-2.6.9-42.ELsmp ro root=LABEL=/1 rhgb quiet vmalloc=256MB
  pci=nommconf
  initrd /initrd-2.6.9-42.ELsmp.img
  ```

Chapter 7. NEW FEATURES

7.1 General CUDA

- Support compatibility between CUDA driver and CUDA toolkit is as follows:
  - Any `nvcc` generated PTX code is forward compatible to newer GPU architectures. This means any CUDA binaries that include PTX code will continue to run on newer GPUs and newer CUDA drivers released from NVIDIA; as the PTX code gets JIT compiled at runtime to the newer GPU architecture.
  - CUDA drivers are backward compatible with CUDA toolkit. This means systems can be upgraded to newer drivers independent of upgrading to newer toolkit. Apps built using old toolkit will load and run with the newer drivers however if they require PTX JIT compilation to run on a newer GPU architecture (SM version) then such apps cannot be used with CUDA tools from old toolkit. Any JIT compiled code implies using the newer compiler and thus a new ABI which requires upgrading to the matching newer toolkit and associated tools.
  - Any separately compiled NVCC binaries (enabled in 5.0) require that all device objects follow the same ABI, and must target the same GPU architecture (SM version). Any CUDA tools usage on these binaries must match the associated toolkit version of the compiler.
- The CUDA 4.2 toolkit for sm_30 implicitly increased a -maxrregcount that was less than 32 to 32. The CUDA 5.0 toolkit does not implicitly increase the -maxrregcount unless it is less than 16 (because the ABI requires at least 16 registers). Note that 32 is the "best minimum" for sm_3x, and the libcublas_device library is compiled for 32 registers.
- Any PTX code generated by NVCC is forward compatible with newer GPU architectures. CUDA binaries that include PTX code will continue to run on newer GPUs with newer NVIDIA CUDA drivers because the PTX code is JIT compiled at runtime to the newer GPU architectures.
- CUDA drivers are backward compatible with the CUDA toolkit. This means systems can be upgraded to newer drivers independently of upgrading to a newer toolkit. Applications built using an older toolkit will load and run with the newer drivers; however, if the applications require PTX JIT compilation to run on a newer GPU

architecture (SM version) then they cannot be used with tools from an older CUDA toolkit. Any JIT-compiled code requires using the newer compiler and thus a new ABI, which, in turn, requires upgrading to the matching newer toolkit and associated tools.

- Any separately compiled NVCC binaries (enabled in 5.0) require that all device objects must follow the same ABI and must target the same GPU architecture (SM version). Any CUDA tool used with these binaries must match the associated toolkit version of the compiler.
- Using flag `cudaStreamNonBlocking` with `cudaStreamCreateWithFlags()` specifies that the created stream will run currently with stream 0 (the NULL stream) and will perform no synchronization with the NULL stream. This flag is functional in the CUDA 5.0 release.
- The `cudaStreamAddCallback()` routine introduces a mechanism to perform work on the CPU after work is finished on the GPU, without polling.
- The `cudaStreamCallbackNonblocking` option for `cudaStreamAddCallback()` and `cuStreamAddCallback()` has been removed from the CUDA 5.0 release. Option `cudaStreamCallbackBlocking` is supported and is the default behavior when no flags are specified.
- CUDA 5.0 introduces support for Dynamic Parallelism, which is a significant enhancement to the CUDA programming model. Dynamic Parallelism allows a kernel to launch and synchronize with new grids directly from the GPU using CUDA's standard <<< >>> syntax. A broad subset of the CUDA runtime API is now available on the device, allowing launch, synchronization, streams, events, and more. For complete information, please see the *CUDA Dynamic Parallelism Programming Guide* which is part of the CUDA 5.0 package. CUDA Dynamic Parallelism is available only on SM 3.5 architecture GPUs.
- The use of a character string to indicate a device symbol, which was possible with certain API functions, is no longer supported. Instead, the symbol should be used directly.

7.1.1 Linux

- Added the `cuIpc` functions, which are designed to allow efficient shared memory communication and synchronization between CUDA processes. Functions `cuIpcGetEventHandle()` and `cuIpcGetMemHandle()` get an opaque handle that can be freely copied and passed between processes on the same machine. The accompanying `cuIpcOpenEventHandle()` and `cuIpcOpenMemHandle()` functions allow processes to map handles to resources created in other processes.

7.2 CUDA Libraries

7.2.1 CUBLAS

- In addition to the usual CUBLAS Library host interface that supports all architectures, the CUDA toolkit now delivers a static CUBLAS library (`cublas_device.a`) that provides the same interface but is callable from the device

from within kernels. The device interface is only available on Kepler II because it uses the Dynamic Parallelism feature to launch kernels internally. More details can be found in the CUBLAS Documentation.

- The CUBLAS library now supports routines `cublas{S,D,C,Z}getrfBatched()`, for batched LU factorization with partial pivoting, and `cublas{S,D,C,Z}trsmBatched()` a batched triangular solver. Those two routines are restricted to matrices of dimension <= 32x32.
- The `cublasCsyr()`, `cublasZsyr()`, `cublasCsyr2()`, and `cublasZsyr2()` routines were added to the CUBLAS library to compute complex and double-complex symmetric rank 1 updates and complex and double-complex symmetric rank 2 updates respectively. Note, `cublasCher()`, `cublasZher()`, `cublasCher2()`, and `cublasZher2()` were already supported in the library and are used for Hermitian matrices.
- The `cublasCsymv()` and `cublasZsymv()` routines were added to the CUBLAS library to compute symmetric complex and double-complex matrix-vector multiplication. Note, `cublasChemv()` and `cublasZhemv()` were already supported in the library and are used for Hermitian matrices.
- A pair of utilities were added to the CUBLAS API for all data types. The `cublas{S,C,D,Z}geam()` routines compute the weighted sum of two optionally transposed matrices. The `cublas{S,C,D,Z}dgmm()` routines compute the multiplication of a matrix by a purely diagonal matrix (represented as a full matrix or with a packed vector).

7.2.2 CURAND

- The Poisson distribution has been added to CURAND, for all of the base generators. Poisson distributed results may be generated via a host function, `curandGeneratePoisson()`, or directly within a kernel via a device function, `curand_poisson()`. The internal algorithm used, and therefore the number of samples drawn per result and overall performance, varies depending on the generator, the value of the frequency parameter (lambda), and the API that is used.

7.2.3 CUSPARSE

- Routines to achieve addition and multiplication of two sparse matrices in CSR format have been added to the CUSPARSE Library.

  The combination of the routines `cusparse{S,D,C,Z}csrgemmNnz()` and `cusparse{S,C,D,Z}csrgemm()` computes the multiplication of two sparse matrices in CSR format. Although the transpose operations on the matrices are supported, only the multiplication of two non-transpose matrices has been optimized. For the other operations, an actual transpose of the corresponding matrices is done internally.

  The combination of the routines `cusparse{S,D,C,Z}csrgeamNnz()` and `cusparse{S,C,D,Z}csrgeam()` computes the weighted sum of two sparse matrices in CSR format.
- The location of the `csrVal` parameter in the `cusparse<t>csrilu0()` and `cusparse<t>csric0()` routines has changed. It now corresponds to the parameter

ordering used in other CUSPARSE routines, which represent the matrix in CSR-storage format (`csrVal`, `csrRowPtr`, `csrColInd`).

- The `cusparseXhyb2csr()` conversion routine was added to the CUSPARSE library. It allows the user to verify that the conversion to HYB format was done correctly.
- The CUSPARSE library has added support for two preconditioners that perform incomplete factorizations: incomplete LU factorization with no fill in (ILU0), and incomplete Cholesky factorization with no fill in (IC0). These are supported by the new functions `cusparse{S,C,D,Z}csrilu0()` and `cusparse{S,C,D,Z}csric0()`, respectively.
- The CUSPARSE library now supports a new sparse matrix storage format called Block Compressed Sparse Row (Block-CSR). In contrast to plain CSR which encodes all non-zero primitive elements, the Block-CSR format divides a matrix into a regular grid of small 2-dimensional sub-matrices, and fully encodes all sub-matrices that have any non-zero elements in them. The library supports conversion between the Block-CSR format and CSR via `cusparse{S,C,D,Z}csr2bsr()` and `cusparse{S,C,D,Z}bsr2csr()`, and matrix-vector multiplication of Block-CSR matrices via `cusparse{S,C,D,Z}bsrmv()`.

7.2.4 Math

- Single-precision `normcdff()` and double-precision `normcdf()` functions were added. They calculate the standard normal cumulative distribution function.

  Single-precision `normcdfinvf()` and double-precision `normcdfinv()` functions were also added. They calculate the inverse of the standard normal cumulative distribution function.
- The `sincospi(x)` and `sincospif(x)` functions have been added to the math library to calculate the double- and single-precision results, respectively, for both `sin(x * PI)` and `cos(x * PI)` simultaneously. Please see the *CUDA Toolkit Reference Manual* for the exact function prototypes and usage, and the *CUDA C Programmer's Guide* for accuracy information. The performance of `sincospi{f}(x)` should generally be faster than calling `sincos{f}(x * PI)` and should generally be faster than calling `sinpi{f}(x)` and `cospi{f}(x)` separately.
- Intrinsic `__frsqrt_rn(x)` has been added to compute the reciprocal square root of single-precision argument `x`, with the single-precision result rounded according to the IEEE-754 rounding mode `nearest` or `even`.

7.2.5 NPP

- The NPP library in the CUDA 5.0 release contains more than 1000 new basic image processing primitives, which include broad coverage for converting colors, copying and moving images, and calculating image statistics.
- Added support for a new filtering-mode for Rotate primitives:

  ```
  NPPI_INTER_CUBIC2P_CATMULLROM
  ```

  This filtering mode uses cubic Catumul-Rom splines to compute the weights for reconstruction. This and the other two CUBIC2P filtering modes are based on the 1988 SIGGRAPH paper: *Reconstruction Filters in Computer Graphics* by Don P. Mitchell

and Arun N. Netravali. At this point NPP only supports the Catmul-Rom filtering for Rotate.

7.3 CUDA Tools

7.3.1 CUDA Compiler

- The separate compilation `culib` format is not supported in the CUDA 5.0 release.
- From this release, the compiler checks the execution space compatibility among multiple declarations of the same function and generates warnings or errors based on the three rules described below.
  - Generates a warning if a function that was previously declared as `__host__` (either implicitly or explicitly) is redeclared with `__device__` or with `__host__ __device__`. After the redeclaration the function is treated as `__host__ __device__`.
  - Generates a warning if a function that was previously declared as `__device__` is redeclared with `__host__` (either implicitly or explicitly) or with `__host__ __device__`. After the redeclaration the function is treated as `__host__ __device__`.
  - Generates an error if a function that was previously declared as `__global__` is redeclared without `__global__`, or vice versa.
- With this release, `nvcc` allows more than one command-line switch that specifies a compilation phase, unless there is a conflict. Known conflicts are as follows:
  - lib cannot be used with `--link` or `--run`.
  - `--device-link` and `--generate-dependencies` cannot be used with other options that specify final compilation phases.

  When multiple compilation phases are specified, `nvcc` stops processing upon the completion of the compilation phase that is reached first. For example, `nvcc --compile --ptx` is equivalent to `nvcc --ptx`, and `nvcc --preprocess --fatbin` equivalent to `nvcc --preprocess`.
- Separate compilation and linking of device code is now supported. See the *Using Separate Compilation in CUDA* section of the `nvcc` documentation for details.

7.3.2 CUDA-GDB

- (Linux and Mac OS) CUDA-GDB fully supports Dynamic Parallelism, a new feature introduced with the 5.0 Toolkit. The debugger is able to track kernels launched from another kernel and to inspect and modify their variables like any CPU-launched kernel.
- When the environment variable `CUDA_DEVICE_WAITS_ON_EXCEPTION` is used, the application runs normally until a device exception occurs. The application then waits for the debugger to attach itself to it for further debugging.
- Inlined subroutines are now accessible from the debugger on SM 2.0 and above. The user can inspect the local variables of those subroutines and visit the call frame stack as if the routines were not inlined.

- Checking the error codes of all CUDA driver API and CUDA runtime API function calls is vital to ensure the correctness of a CUDA application. Now the debugger is able to report, and even stop, when any API call returns an error. See the CUDA-GDB documentation on `set cuda api_failures` for more information.
- It is now possible to attach the debugger to a CUDA application that is already running. It is also possible to detach it from the application before letting it run to completion. When attached, all the usual features of the debugger are available to the user, just as if the application had been launched from the debugger.

7.3.3 CUDA-MEMCHECK

- CUDA-MEMCHECK, when used from within the debugger, now displays the address space and the address of the faulty memory access.
- CUDA-MEMCHECK now displays the backtrace on the host and device when an error is discovered.
- CUDA-MEMCHECK now detects `double free()` and `invalid free()` on the device.
- The precision of the reported errors for `local`, `shared`, and `global` memory accesses has been improved.
- CUDA-MEMCHECK now reports leaks originating from the device heap.
- CUDA-MEMCHECK now reports error codes returned by the runtime API and the driver API in the user application.
- CUDA-MEMCHECK now supports reporting data access hazards in shared memory. Use the `--tool racecheck` command-line option to activate.

7.3.4 NVIDIA Nsight Eclipse Edition

- (Linux and Mac OS) Nsight Eclipse Edition is an all-in-one development environment that allows developing, debugging, and optimizing CUDA code in an integrated UI environment.

7.3.5 NVIDIA Visual Profiler, Command Line Profiler

- As mentioned in the Release Highlights, the tool, `nvprof`, is now available in release 5.0 for collecting profiling information from the command-line.

Chapter 8. PERFORMANCE IMPROVEMENTS

8.1 CUDA Libraries

8.1.1 CUBLAS

- On Kepler architectures, shared-memory access width can be configured for 4-byte banks (default) or 8-byte banks using the routine `cudaDeviceSetSharedMemConfig()`. The CUBLAS and CUSPARSE libraries do not affect the shared-memory configuration, although some routines might benefit from it. It is up to users to choose the best shared-memory configuration for their applications prior to calling the CUBLAS or CUSPARSE routines.
- In CUDA Toolkit 5.0, `cublas<S,D,C,Z>symv()` and `cublas<C/Z>chemv()` have an alternate, faster implementation that uses atomics. The regular implementation, which gives predictable results from one run to another, is run by default. The routine `cublasSetAtomicsMode()` can be used to choose the alternate, faster version.

8.1.2 CURAND

- In CUDA CURAND for 5.0, the Box-Muller formula, used to generate double-precision normally distributed results, has been optimized to use `sincospi()` instead of individual calls to `sin()` and `cos()` with multipliers to scale the parameters. This results in a 30% performance improvement on a Tesla C2050, for example, when generating double-precision normal results.

8.1.3 Math

- The performance of the double-precision `fmod()`, `remainder()`, and `remquo()` functions has been significantly improved for `sm_13`.
- The `sin()` and `cos()` family of functions [`sin()`, `sinpi()`, `cos()`, and `cospi()`] have new implementations in this release that are more accurate and faster. Specifically, all of these functions have a worst-case error bound of 1 ulp, compared to 2 ulps in previous releases. Furthermore, the performance of these functions has

improved by 25% or more, although the exact improvement observed can vary from kernel to kernel. Note that the `sincos()` and `sincospi()` functions also inherit any accuracy improvements from the component functions.

- Function `erfcinvf()` has been significantly optimized for both the Tesla and Fermi architectures, and the worst case error bound has improved from 7 ulps to 4 ulps.

Chapter 9. RESOLVED ISSUES

9.1 General CUDA

- When PTX JIT is used to execute `sm_1x`- or `sm_2x`-native code on Kepler, and when the maximum grid dimension is selected based on the grid-size limits reported by `cudaGetDeviceProperties()`, a conflict can occur between the grid size used and the size limit presumed by the JIT'd device code.

  The grid size limit on devices of compute capability 1.x and 2.x is 65535 blocks per grid dimension. If an application attempts to launch a grid with >= 65536 blocks in the x dimension on such devices, the launch fails outright, as expected. However, because Kepler increased the limit (for the x dimension) to 2^{31}-1 blocks per grid, previous CUDA Driver releases allowed such a grid to launch successfully; but this grid exceeds the number of blocks that can fit into the 16-bit grid size and 16-bit block index assumed by the compiled device code. Beginning in CUDA release 5.0, launches of kernels compiled native to earlier GPUs and JIT'd onto Kepler now return an error as they would have with the earlier GPUs, avoiding the silent errors that could otherwise result.

  This can still pose a problem for applications that select their grid launch dimensions based on the limits reported by `cudaGetDeviceProperties()`, since this function reports 2^{31}-1 for the grid size limit in the x dimension for Kepler GPUs. Applications that correctly limited their launches to 65535 blocks per grid in the x dimension on earlier GPUs may attempt bigger launches on Kepler--yet these launches will fail. To work around this issue for existing applications that were not built with Kepler-native code, a new environment variable has been added for backward compatibility with earlier GPUs: setting `CUDA_GRID_SIZE_COMPAT = 1` causes `cudaGetDeviceProperties()` to conservatively underreport 65535 as the maximum grid dimension on Kepler, allowing such applications to work as expected.
- Functions `cudaGetDeviceProperties()`, `cuDeviceGetProperties()`, and `cuDeviceGetAttribute()` may return the incorrect clock frequency for the SM clock on Kepler GPUs.

9.2 CUDA Libraries

9.2.1 CURAND

- In releases prior to CUDA 5.0, the CURAND pseudorandom generator MRG32k3a returned integer results in the range 1 through 4294967087 (the larger of two primes used in the generator). CUDA 5.0 results have been scaled to extend the range to 4294967295 (2^{32} - 1). This causes the generation of integer sequences that are somewhat different from previous releases. All other distributions (that is, uniform, normal, log-normal, and Poisson) were already correctly scaled and are not affected by this change.

9.2.2 CUSPARSE

- An extra parameter (`int * nnzTotalDevHostPtr`) was added to the parameters accepted by the functions `cusparseXcsrgeamNnz()` and `cusparseXcsrgemmNnz()`. The memory pointed to by `nnzTotalDevHostPtr` can be either on the device or host, depending on the selected `CUBLAS_POINTER_MODE`. On exit, `*nnzTotalDevHostPtr` holds the total number of non-zero elements in the resulting sparse matrix C.

9.2.3 NPP

- The `nppiLUT_Linear_8u_C1R` and all other LUT primitives that existed in NPP release 4.2 have undergone an API change. The pointers provided for the parameters `pValues` and `pLevels` have to be device pointers from version 5.0 onwards. In the past, those two values were expected to be host pointers, which was in violation of the general NPP API guideline that all pointers to NPP functions are device pointers (unless explicitly noted otherwise).
- The implementation of the `nppiWarpAffine*()` routines in the NPP library have been completely replaced in this release. This fixes several outstanding bugs related to these routines.
- Added these two primitives, which were temporarily removed from release 4.2:

```
nppiAbsDiff_8u_C3R
nppiAbsDiff_8u_C4R
```

9.2.4 Thrust

- The version of Thrust included with the current CUDA toolkit was upgraded to version 1.5.3 in order to address several minor issues.

9.3 CUDA Tools

- (Windows) The file `fatbinary.h` has been released with the CUDA 5.0 Toolkit. The file, which replaces `__cudaFatFormat.h`, describes the format used for all fat binaries since CUDA 4.0.

9.3.1 CUDA Compiler

- The CUDA compiler driver, `nvcc`, predefines the macro `__NVCC__`. This macro can be used in C/C++/CUDA source files to test whether they are currently being compiled by `nvcc`. In addition, `nvcc` predefines the macro `__CUDACC__`, which can be used in source files to test whether they are being treated as CUDA source files. The `__CUDACC__` macro can be particularly useful when writing header files.
- It is to be noted that the previous releases of `nvcc` also predefined the `__CUDACC__` macro; however, the description in the document *The CUDA Compiler Driver NVCC* was incorrect. The document has been corrected in the CUDA 5.0 release.

9.3.2 CUDA Occupancy Calculator

- There was an issue in the CUDA Occupancy Calculator that caused it to be overly conservative in reporting the theoretical occupancy on Fermi and Kepler when the number of warps per block was not a multiple of 2 or 4.

Chapter 10. KNOWN ISSUES

10.1 General CUDA

- The CUDA reference manual incorrectly describes the type of `CUdeviceptr` as an unsigned `int` on all platforms. On 64-bit platforms, a `CUdeviceptr` is an unsigned `long long`, not an unsigned `int`.
- Individual GPU program launches are limited to a run time of less than 5 seconds on a GPU with a display attached. Exceeding this time limit usually causes a launch failure reported through the CUDA driver or the CUDA runtime. GPUs without a display attached are not subject to the 5 second runtime restriction. For this reason it is recommended that CUDA be run on a GPU that is NOT attached to a display and does not have the Windows desktop extended onto it. In this case, the system must contain at least one NVIDIA GPU that serves as the primary graphics adapter.

10.1.1 Linux, Mac OS

- Device code linking does not support object files that are in Mac OS fat-file format. As a result, the device libraries included in the toolkit (libcudadevrt.a and libcublas_device.a) do not use the fat file format and only contain code for a 64-bit architecture. In contrast, the other libraries in the toolkit on the Mac OS platform do use the fat file format and support both 32-bit and 64-bit architectures.
- At the time of this release, there are no Mac OS configurations available that support GPUs that implement the sm_35 architecture. Code that targets this architecture can be built, but cannot be run or tested on a Mac OS platform with the CUDA 5.0 toolkit.
- The Linux kernel provides a mode where it allows user processes to overcommit system memory. (Refer to kernel documentation for `/proc/sys/vm/` for details). If this mode is enabled (the default on many distros) the kernel may have to kill processes in order to free up pages for allocation requests. The CUDA driver process, especially for CUDA applications that allocate lots of zero-copy memory with `cuMemHostAlloc()` or `cudaMallocHost()`, is particularly vulnerable to being killed in this way. Since there is no way for the CUDA SW stack to report an OOM error to the user before the process disappears, users, especially on 31-bit Linux, are encouraged to disable memory overcommit in their kernel to avoid

this problem. Please refer to documentation on `vm.overcommit_memory` and `vm.overcommit_ratio` for more information.

- When compiling with GCC, special care must be taken for structs that contain 64-bit integers. This is because GCC aligns long longs to a 4-byte boundary by default, while `nvcc` aligns long longs to an 8-byte boundary by default. Thus, when using GCC to compile a file that has a struct/union, users must give the `-malign-double` option to GCC. When using `nvcc`, this option is automatically passed to GCC.
- (Mac OS) When CUDA applications are run on 2012 MacBook Pro models, allowing or forcing the system to go to sleep causes a system crash (kernel panic). To prevent the computer from automatically going to sleep, set the **Computer Sleep** option slider to **Never** in the **Energy Saver** pane of the **System Preferences**.
- (Mac OS) To save power, some Apple products automatically power down the CUDA- capable GPU in the system. If the operating system has powered down the CUDA-capable GPU, CUDA fails to run and the system returns an error that no device was found. In order to ensure that your CUDA-capable GPU is not powered down by the operating system do the following:

  1. Go to **System Preferences**.
  2. Open the **Energy Saver** section.
  3. Uncheck the **Automatic graphics switching** box in the upper left.

10.1.2 Windows

- Individual kernels are limited to a 2-second runtime by Windows Vista. Kernels that run for longer than 2 seconds will trigger the Timeout Detection and Recovery (TDR) mechanism. For more information, see http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx.
- The maximum size of a single memory allocation created by `cudaMalloc()` or `cuMemAlloc()` on WDDM devices is limited to

  `MIN( (System Memory Size in MB - 512 MB) / 2, PAGING_BUFFER_SEGMENT_SIZE ).`

  For Vista, `PAGING_BUFFER_SEGMENT_SIZE` is approximately 2 GB.

10.2 CUDA Libraries

10.2.1 NPP

- The NPP `ColorTwist_32f_8u_P3R` primitive does not work properly for line strides that are not 64-byte aligned. This issue can be worked around by using the image memory allocators provided by the NPP library.

10.3 CUDA Tools

With separate compiled binaries the values of the local variables may be incorrect in the debugger, please use fully compiled binaries while debugging.

10.3.1 CUDA Compiler

- (Windows) Because Microsoft changed the declaration of the `hypot()` function between MSVC v9 and MSVC v10, users of Microsoft Visual Studio 2010 who link with the new `cublas_device.lib` and `cudadevrt.lib` device-code libraries may encounter an error. Specifically, performing device- and host-linking in a single pass using `NVCC` on a system with Visual Studio 2010 gives the error `unresolved external symbol hypot`. Users who encounter this error can avoid it by linking in two stages: first device-link with `nvcc -dlink` and then host-link using `cl`. This error should not arise from the VS2010 IDE when using the CUDA plug-in, as that plug-in already links in two stages.
- A CUDA program may not compile correctly if a `type` or `typedef T` is private to a class or a structure, and at least one of the following is satisfied:
  - `T` is a parameter type for a `__global__` function.
  - `T` is an argument type for a template instantiation of a `__global__` function.

  This restriction will be fixed in a future release.
- (Linux) The `__float128` data type is not supported for the `gcc` host compiler.
- (Mac OS) The documentation surrounding the use of the flag `-malign-double` suggests it be used to make the struct size the same between host and device code. We know now that this flag causes problems with other host libraries. The CUDA documentation will be updated to reflect this.

  The work around for this issue is to manually add padding so that the structs between the host compiler and CUDA are consistent.
- (Windows) When the `PATH` environment variable contains double quotes (`"`), `nvcc` may fail to set up the environment for Microsoft Visual Studio 2010, generating an error. This is because `nvcc` runs `vcvars32.bat` or `vcvars64.bat` to set up the environment for Microsoft Visual Studio 2010 and these batch files are not always able to process `PATH` if it contains double quotes.

  One workaround for this issue is as follows:

  1. Make sure that `PATH` does not contain any double quotes.
  2. Run `vcvars32.bat` or `vcvars64.bat`, depending on the system.
  3. Add the directories that need to be added to `PATH` with double quotes.
  4. Run `NVCC` with the `--use-local-env` switch.

10.3.2 NVIDIA Visual Profiler, Command Line Profiler

- On Mac OS X systems with NVIDIA drivers earlier than version 295.10.05, the Visual Profiler may fail to import session files containing profile information collected from GPUs with compute capability 3.0 or later.
- If required, a Java installation is triggered the first time the Visual Profiler is launched. If this occurs, the Visual Profiler must be exited and restarted.
- Visual Profiler fails to generate events or counter information. Here are a couple of reasons why Visual Profiler may fail to gather counter information.

  More than one tool is trying to access the GPU. To fix this issue please make sure only one tool is using the GPU at any given point. Tools include the CUDA command line profiler, Parallel NSight Analysis Tools and Graphics Tools, and applications that use either CUPTI or PerfKit API (NVPM) to read counter values.

  More than one application is using the GPU at the same time Visual Profiler is profiling a CUDA application. To fix this issue please close all applications and just run the one with Visual Profiler. Interacting with the active desktop should be avoided while the application is generating counter information. Please note that for some types of counters Visual Profiler gathers counters for only one context if the application is using multiple contexts within the same application.
- Enabling certain counters can cause GPU kernels to run longer than the driver's watchdog time-out limit. In these cases the driver will terminate the GPU kernel resulting in an application error and profiling data will not be available. Please disable the driver watchdog time out before profiling such long running CUDA kernels.
  - On Linux, setting the X Config option `Interactive` to `false` is recommended.
  - For Windows, detailed information on disabling the Windows TDR is available at

    http://msdn.microsoft.com/en-us/windows/hardware/gg487368.aspx#E2
- Enabling counters on GPUs with compute capability (SM type) 1.x can result in occasional hangs. Please disable counters on such runs.
- The `warp serialize` counter for GPUs with compute capability 1.x is known to give incorrect and high values for some cases.
- To ensure that all profile data is collected and flushed to a file, `cudaDeviceSynchronize()` followed by either `cudaDeviceReset()` or `cudaProfilerStop()` should be called before the application exits.
- Counters `gld_incoherent` and `gst_incoherent` always return zero on GPUs with compute capability (SM type) 1.3. A value of zero doesn't mean that all load/stores are 100% coalesced.
- Use Visual Profiler version 4.1 onwards with NVIDIA driver version 285 (or later). Due to compatibility issues with profile counters, Visual Profiler 4.0 (or earlier) must not be used with NVIDIA driver version 285 (or later).

Chapter 11. SOURCE CODE FOR OPEN64 AND CUDA-GDB

- The Open64 and CUDA-GDB source files are controlled under terms of the GPL license. Current and previously released versions are located here:

  ftp://download.nvidia.com/CUDAOpen64.
- Linux users:
  - Please refer to the *Release Notes* and *Known Issues* sections in the *CUDA-GDB User Manual* (`CUDA_GDB.pdf`).
  - Please refer to `CUDA_Memcheck.pdf` for notes on supported error detection and known issues.

Chapter 12. MORE INFORMATION

- For more information and help with CUDA, please visit http://www.nvidia.com/cuda.
- Please refer to the *LLVM Release License* text in `EULA.txt` for details on LLVM licensing.