CUDA-GDB

NVIDIA CUDA Debugger - 4.0 Release for Linux and Mac

DU-05227-001_V4.0 | April 25, 2011

User Manual

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

01 INTRODUCTION

This document introduces cuda-gdb, the NVIDIA® CUDA™ debugger, and describes what is new in version 4.0.

What is cuda-gdb?

CUDA-GDB is the NVIDIA tool for debugging CUDA applications running on Linux and Mac. CUDA-GDB is an extension to the i386/AMD64 port of GDB, the GNU Project debugger. The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables developers to debug applications without the potential variations introduced by simulation and emulation environments.

CUDA-GDB runs on Linux and Mac OS X, 32-bit and 64-bit.The Linux edition is based on GDB 6.6 whereas the Mac edition is based on GDB 6.3.5.

Supported features

CUDA-GDB is designed to present the user with a seamless debugging environment that allows simultaneous debugging of both GPU and CPU code within the same application. Just as programming in CUDA C is an extension to C programming, debugging with CUDA-GDB is a natural extension to debugging with GDB. The existing GDB debugging features are inherently present for debugging the host code, and additional features have been provided to support debugging CUDA device code.

CUDA-GDB supports C and C++ CUDA applications. All the C++ features supported by the NVCC compiler can be debugged by cuda-gdb.

CUDA-GDB allows the user to set breakpoints, to single-step CUDA applications, and also to inspect and modify the memory and variables of any given thread running on the hardware.

CUDA-GDB supports debugging all CUDA applications, whether they use the CUDA driver API, the CUDA runtime API, or both.

CUDA-GDB supports debugging kernels that have been compiled for specific CUDA architectures, such as sm_10 or sm_20, but also supports debugging kernels compiled at runtime, referred to as just-in-time compilation, or JIT compilation for short.

About this document

This document is the main documentation for CUDA-GDB and is organized more as a user manual than a reference manual. The rest of the document will describe how to install and use CUDA-GDB to debug CUDA kernels and how to use the new CUDA commands that have been added to GDB. Some walk-through examples are also provided. It is assumed that the user already knows the basic GDB commands used to debug host applications.

02 RELEASE NOTES

The following features have been added for the 4.0 release:

Three-Dimensional Grid Support

Starting with Release 270 of the CUDA driver, grids can be three-dimensional. Now, cuda-gdb supports the new Z dimension.

C++ Debugging Support

The debugger now supports the debugging of C++ applications, including templates, named namespaces (no alias), virtual functions, classes, and methods overloading. In particular, breakpoints set on lines within templatized functions will create multiple breakpoints, one per instance of the template.

Mac Debugging Support

CUDA-GDB now runs on 32-bit and 64-bit Mac OS X 10.6.5 systems and supports all the same features as its Linux counterpart.

Automatic breakpoint on kernel launches

A new option, '**`set cuda break_on_launch none|application|system|all`**', allows the user to decide if the debugger should stop at the entrance of every system or application kernel.

Textures

The debugger now supports the debugging of kernels using textures. Functionality has also been added to allow textures to be read.

Improved info cuda commands

Command names have been changed to be more consistent with other 'info' commands. The '**info cuda**' commands are now: **devices, sms, warps, lanes, kernels, blocks, threads**. The output format has been streamlined for readability.

Filters

The '**info cuda**' commands support focus filters where only the selected devices, SMs, warps, lanes, kernels, blocks, and threads are considered. It allows the user to filter the output data to the unit of interest.

MI support

The '**info cuda**' and '**cuda**' commands are now available as MI commands.

Fermi disassembly

Now, kernel code running on Fermi (sm_20) can be disassembled, using the 'x/i' command. Before this release, only Tesla code (sm_10) could be disassembled.

Conditional Breakpoints

The break command now supports conditional breakpoints on device code, as long as there is no function call in the conditional statement. Built-in variables such as **threadIdx** and **blockIdx** can also be used.

Deprecated Commands

The deprecated command thread <<<(x,y),(x,y,z)>>> used to switch CUDA thread focus has been retired. Instead, the user should use the **cuda thread** command.

OpenGL-Interop Applications

CUDA-GDB supports remote debugging of CUDA applications that use OpenGL-interoperability. For details, refer to the debugger usage scenarios in Chapter 03.

03 GETTING STARTED

Included in this chapter are instructions for installing cuda-gdb and for using NVCC, the NVIDIA CUDA compiler driver, to compile CUDA programs for debugging.

Installation Instructions

Follow these steps to install cuda-gdb.

1. Visit the NVIDIA CUDA Zone download page:

   `http://www.nvidia.com/object/cuda_get.html`.

2. Select the appropriate operating system–MacOS or Linux.

   (See "Host Platform Requirements" on page 26.)

3. Download and install the CUDA Driver.

4. Download and install the CUDA Toolkit.

Setting up the debugger environment

Linux

Set up the PATH and LD_LIBRARY_PATH environment variables:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:/usr/local/cuda/
lib:$LD_LIBRARY_PATH
```

Mac

Set up the PATH and DYLD_LIBRARY_PATH environment variables:

```
export PATH=/usr/local/cuda/bin:$PATH
export DYLD_LIBRARY_PATH=/ usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

Also, if you are unable to execute cuda-gdb or if you hit the “Unable to find Mach task port for processid” error, try resetting the correct permissions with the following commands:

```
sudo chgrp procmod /usr/local/cuda/bin/cuda-binary-gdb
sudo chmod 2755 /usr/local/cuda/bin/cuda-binary-gdb
sudo chmod 755 /usr/local/cuda/bin/cuda-gdb
```

Compiling the application

Debug Compilation

NVCC, the NVIDIA CUDA compiler driver, provides a mechanism for generating the debugging information necessary for cuda-gdb to work properly. The **-g -G** option pair must be passed to NVCC when an application is compiled in order to debug with cuda-gdb; for example,

```
nvcc -g -G foo.cu -o foo
```

Using this line to compile the CUDA application `foo.cu`

- forces **-O0** compilation, with the exception of very limited dead-code eliminations and register-spilling optimizations.
- makes the compiler include symbolic debugging information in the executable

Compiling for Fermi GPUs

For Fermi GPUs, add the following flags to target Fermi output when compiling the application:

```
-gencode arch=compute_20,code=sm_20
```

It will compile the kernels specifically for the Fermi architecture once and for all. If the flag is not specified, then the kernels must be recompiled at runtime every time.

Compiling for Fermi and Tesla GPUs

If you are targeting both Fermi and Tesla GPUs, include these two flags:

```
-gencode arch=compute_20,code=sm_20
-gencode arch=compute_10,code=sm_10
```

Using the debugger

Debugging a CUDA GPU involves pausing that GPU. When the graphics desktop manager is running on the same GPU, then debugging that GPU freezes the GUI and makes the desktop unusable. To avoid this, use cuda-gdb in the following system configurations:

Single GPU Debugging

In a single GPU system, cuda-gdb can be used to debug CUDA applications only if no X11 server (on Linux) or no Aqua desktop manager (on Mac OS X) is running on that system. On Linux you can stop the X11 server by stopping the gdm service. On Mac OS X you can log in with ">console" as the user name in the desktop UI login screen. This allows CUDA applications to be executed and debugged in a single GPU configuration.

Multi-GPU Debugging

Multi-GPU debugging is not much different than single-GPU debugging except for a few additional **`cuda-gdb`** commands that let you switch between the GPUs.

Any GPU hitting the breakpoint will pause all the GPUs running CUDA on that system. Once paused, you can use **info cuda kernels** to view all the active kernels and the GPUs they are running on. When any GPU is resumed, all the GPUs are resumed.

All CUDA-capable GPUs can run the same or different kernels. To switch to an active kernel you can use **cuda kernel <n>** or **cuda device <n>** to switch to the desired GPU where *n* is the id of the kernel or GPU retrieved from **info cuda kernels**. Once you are on an active kernel and a GPU, then the rest of the process is the same as single-GPU debugging.

Note: The same module, and therefore the same kernel, can be loaded and used by different contexts and devices at the same time. When a breakpoint is set in such a kernel, by either name or file name and line number, it will be resolved arbitrarily to only one instance of that kernel.
With the runtime API, the exact instance to which the breakpoint will be resolved cannot be controlled.
With the driver API, the user can control the instance to which the breakpoint will be resolved to by setting the breakpoint *right after* its module is loaded.

Multi-GPU Debugging in Console Mode

CUDA-GDB allows simultaneous debugging of applications running CUDA kernels on multiple GPUs. In console mode, cuda-gdb can be used to pause and debug every GPU in the system. You can enable console mode as described above for the single GPU console mode.

Multi-GPU Debugging with the Desktop Manager Running

This can be achieved by running the desktop GUI on one GPU and CUDA on the other GPU to avoid hanging the desktop GUI.

On Linux

The CUDA driver automatically excludes the GPU used by X11 from being visible to the application being debugged. This prevents the behavior of the application since, if there are *n* GPUs in the system, then only *n-1* GPUs will be visible to the application.

On Mac OS X

The CUDA driver exposes every CUDA-capable GPU in the system, including the one used by Aqua desktop manager. To determine which GPU should be used for CUDA, run the deviceQuery app from the CUDA SDK sample. The output of deviceQuery as shown in Figure 3.1 indicates all the GPUs in the system.

For example, if you have two GPUs you will see Device0: "GeForce xxxx" and Device1: "GeForce xxxx". Choose the Device<index> that is not rendering the desktop on your connected monitor. If Device0 is rendering the desktop, then choose Device1 for running and debugging the CUDA application. This exclusion of the desktop can be achieved by setting the CUDA_VISIBLE_DEVICES environment variable to 1:

```
export CUDA_VISIBLE_DEVICES=1
```

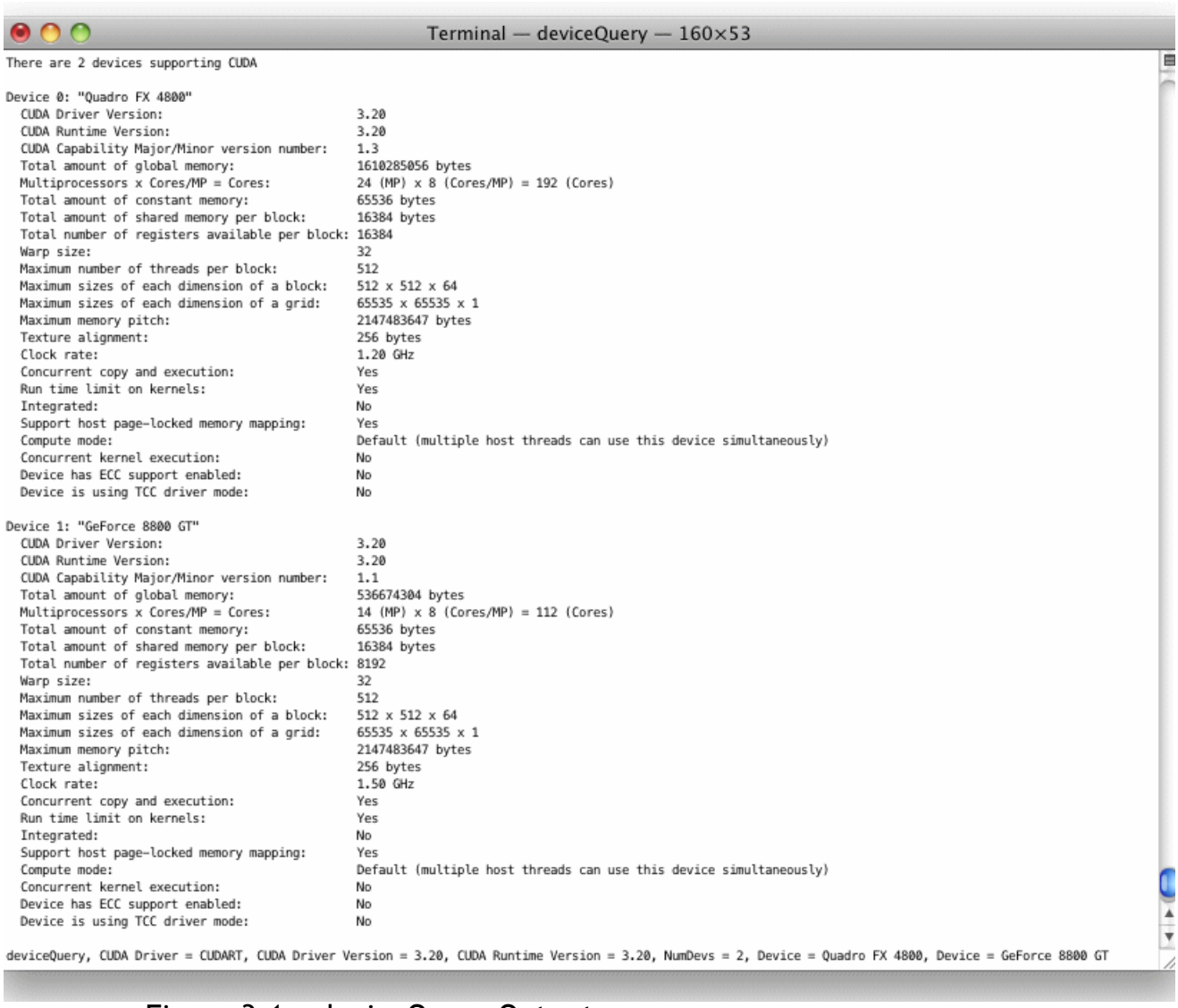

Figure 3.1 deviceQuery Output

Remote Debugging

To remotely debug an application, use SSH or VNC from the host system to connect to the target system. From there, cuda-gdb can be launched in console mode.

CUDA/OpenGL Interop Applications on Linux

Instructions

In order to use cuda-gdb to debug CUDA applications that interoperate with OpenGL, first launch X as a non-interactive session as follows:

1. Stop your X server.
2. Edit /etc/X11/xorg.conf to contain the following line in the Device section corresponding to your display:

   ```
   Option          "Interactive" "off"
   ```

3. Restart your X server.

Then, log in remotely (SSH, etc.) and launch your application under cuda-gdb. This setup works properly for single-GPU and multi-GPU configurations.

Ensure your DISPLAY environment variable is set appropriately (for example, export DISPLAY=:0.0).

Limitations

While X is in non-interactive mode, interacting with the X session can cause your debugging session to stall or terminate.

04 CUDA-GDB EXTENSIONS

Command Naming Convention

The existing GDB commands are unchanged. Every new CUDA command or option is prefixed with the CUDA keyword. As much as possible, cuda-gdb command names will be similar to the equivalent GDB commands used for debugging host code. For instance, the GDB command to display the host threads and switch to host thread 1 are, respectively:

```
(cuda-gdb) info threads
(cuda-gdb) thread 1
```

To display the CUDA threads and switch to cuda thread 1, the user only has to type:

```
(cuda-gdb) info cuda threads
(cuda-gdb) cuda thread 1
```

Getting Help

As with GDB commands, the built-in help for the CUDA commands is accessible from the cuda-gdb command line by using the help command:

```
(cuda-gdb) help cuda name_of_the_cuda_command
```

```
(cuda-gdb) help set cuda name_of_the_cuda_option
```

```
(cuda-gdb) help info cuda name_of_the_info_cuda_command
```

Initialization File

The initialization file for cuda-gdb is named `.cuda-gdbinit` and follows the same rules as the standard `.gdbinit` file used by GDB. The initialization file may contain any CUDA-

GDB command. Those commands will be processed in order when cuda-gdb is launched.

GUI Integration

Emacs

CUDA-GBD works with GUD in Emacs and XEmacs . No extra step is required besides pointing to the right binary.

To use cuda-gdb, the 'gud-gdb-command-name' variable must be set to "cuda-gdb --annotate=3". Use M-x customize-variable to set the variable.

Ensure that cuda-gdb is present in the Emacs/XEmacs $PATH.

DDD

CUDA-GDB works with DDD. To use DDD with cuda-gdb, launch DDD with the following command:

```
ddd --debugger cuda-gdb
```

cuda-gdb must be in your $PATH.

05 KERNEL FOCUS

A CUDA application may be running several host threads and many device threads. To simplify the visualization of information about the state of application, commands are applied to the entity in focus.

When the focus is set to a host thread, the commands will apply only to that host thread (unless the application is fully resumed, for instance). On the device side, the focus is always set to the lowest granularity level–the device thread.

Software Coordinates vs. Hardware Coordinates

A device thread belongs to a block, which in turn belongs to a kernel. Thread, block, and kernel are the software coordinates of the focus. A device thread runs on a lane. A lane belongs to a warp, which belongs to an SM, which in turn belongs to a device. Lane, warp, SM, and device are the hardware coordinates of the focus. Software and hardware coordinates can be used interchangeably and simultaneously as long as they remain coherent.

Another software coordinate is sometimes used: the grid. The difference between a grid and a kernel is the scope. The grid ID is unique per GPU whereas the kernel ID is unique across all GPUs. Therefore there is a 1:1 mapping between a kernel and a (grid,device) tuple.

Current Focus

To inspect the current focus, use the cuda command followed by the coordinates of interest:

```
(cuda-gdb) cuda device sm warp lane block thread
block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0
(cuda-gdb) cuda kernel block thread
kernel 1, block (0,0,0), thread (0,0,0)
(cuda-gdb) cuda kernel
kernel 1
```

Switching Focus

To switch the current focus, use the cuda command followed by the coordinates to be changed:

```
(cuda-gdb) cuda device 0 sm 1 warp 2 lane 3
[Switching focus to CUDA kernel 1, grid 2, block (8,0,0), thread
(67,0,0), device 0, sm 1, warp 2, lane 3]
374 int totalThreads = gridDim.x * blockDim.x;
```

If the specified focus is not fully defined by the command, the debugger will assume that the omitted coordinates are set to the coordinates in the current focus, including the subcoordinates of the block and thread.

```
(cuda-gdb) cuda thread (15)
[Switching focus to CUDA kernel 1, grid 2, block (8,0,0), thread
(15,0,0), device 0, sm 1, warp 0, lane 15]
374 int totalThreads = gridDim.x * blockDim.x;
```

The parentheses for the block and thread arguments are optional.

```
(cuda-gdb) cuda block 1 thread 3
[Switching focus to CUDA kernel 1, grid 2, block (1,0,0), thread (3,0,0),
device 0, sm 3, warp 0, lane 3]
374 int totalThreads = gridDim.x * blockDim.
```

06 PROGRAM EXECUTION

Applications are launched the same way in cuda-gdb as they are with GDB by using the **`run`** command. This chapter describes how to interrupt and single-step CUDA applications.

Interrupting the Application

If the CUDA application appears to be hanging or stuck in an infinite loop, it is possible to manually interrupt the application by pressing **CTRL+C**. When the signal is received, the GPU is suspended and the cuda-gdb prompt will appear.

At that point, the program can be inspected, modified, single-stepped, resumed, or terminated at the user's discretion.

This feature is limited to applications running within the debugger. It is not possible to break into and debug applications that have been previously launched.

Single-Stepping

Single-stepping device code is supported. However, unlike host code single-stepping, device code single-stepping works at the warp level. This means that single-stepping a device kernel advances all the threads in the warp currently in focus.

In order to advance the execution of more than one warp, a breakpoint must be set at the desired location and then the application must be fully resumed.

A special case is single-stepping over a thread barrier call: __syncthreads(). In this case, an implicit temporary breakpoint is set immediately after the barrier and all threads are resumed until the temporary breakpoint is hit.

On GPUs with sm_type lower than sm_20 it is not possible to step over a subroutine in the device code. Instead, CUDA-GDB always steps into the device function. On GPUs with sm_type sm_20 and higher, you can step in, over, or out of the device functions as

long as they are not inlined. To force a function to not be inlined by the compiler, the **__ __noinline__ __** keyword must be added to the function declaration.

07 BREAKPOINTS

There are multiple ways to set a breakpoint on a CUDA application. Those methods are described below. The commands to set a breakpoint on the device code are the same as the commands used to set a breakpoint on the host code.

If the breakpoint is set on device code, the breakpoint will be marked pending until the ELF image of the kernel is loaded. At that point, the breakpoint will be resolved and its address will be updated.

When a breakpoint is set, it forces all resident GPU threads to stop at this location when it hits that corresponding PC. There is currently no method to stop only certain threads or warps at a given breakpoint.

When a breakpoint is hit in one thread, there is no guarantee that the other threads will hit the breakpoint at the same time. Therefore the same breakpoint may be hit several times, and the user must be careful with checking which thread(s) actually hit(s) the breakpoint.

Symbolic breakpoints

To set a breakpoint at the entry of a function, use the break command followed by the name of the function or method:

```
(cuda-gdb) break my_function
(cuda-gdb) break my_class::my_method
```

For templatized functions and methods, the full signature must be given:

```
(cuda-gdb) break int my_templatized_function<int>(int)
```

The mangled name of the function can also be used. To find the mangled name of a function, you can use the following command:

```
(cuda-gdb) set demangle-style none
(cuda-gdb) info function my_function_name
(cuda-gdb) set demangle-style auto
```

Line Breakpoints

To set a breakpoint on a specific line number, use the following syntax:

```
(cuda-gdb) break my_file.cu:185
```

If the specified line corresponds to an instruction within templatized code, multiple breakpoints will be created, one for each instance of the templatized code. Be aware that, at this point, those multiple breakpoints cannot be saved from one run to the next and will be deleted when the application is run again. The user must then manually set those breakpoints again.

Address Breakpoints

To set a breakpoint at a specific address, use the break command with the address as argument:

```
(cuda-gdb) break 0x1afe34d0
```

The address can be any address on the device or the host.

Kernel Entry Breakpoints

To break on the first instruction of every launched kernel, set the break_on_launch option to application:

```
(cuda-gdb) set cuda break_on_launch application
```

Possible options are:

- `application`: any kernel launched by the user application
- `system`: any kernel launched by the driver, such as memset.
- `all`: any kernel, application and kernel.
- `none`: no kernel, application or kernel.

Those automatic breakpoints are not displayed by the info breakpoints command and are managed separately from individual breakpoints. Turning off the option will not delete other individual breakpoints set to the same address and vice-versa.

Conditional Breakpoints

To make the breakpoint conditional, use the optional if keyword or the cond command.

```
(cuda-gdb) break foo.cu:23 if threadIdx.x == 1 && i < 5
(cuda-gdb) cond 3 threadIdx.x == 1 && i < 5
```

Conditional expressions may refer any variable, including built-in variables such as threadIdx and blockIdx. Function calls are not allowed in conditional expressions.

Note that conditional breakpoints are always hit and evaluated, but the debugger reports the breakpoint as being hit only if the conditional statement is evaluated to true. The process of hitting the breakpoint and evaluating the corresponding conditional statement is time-consuming. Therefore, running applications while using conditional breakpoints may slow down the debugging session. Moreover, if the conditional statement is always evaluated to false, the debugger may appear to be hanging or stuck, although it is not the case. You can interrupt the application with CTRL-C to verify that progress is being made.

08 INSPECTING PROGRAM STATE

Memory and Variables

The GDB print command has been extended to decipher the location of any program variable and can be used to display the contents of any CUDA program variable including:

- data allocated via **`cudaMalloc()`**
- data that resides in various GPU memory regions, such as shared, local, and global memory
- special CUDA runtime variables, such as **`threadIdx`**

Variable Storage and Accessibility

Depending on the variable type and usage, variables can be stored either in registers or in local, shared, const or global memory. You can print the address of any variable to find out where it is stored and directly access the associated memory.

The example below shows how the variable array, which is of type shared int *, can be directly accessed in order to see what the stored values are in the array.

```
(cuda-gdb) print &array
$1 = (@shared int (*)[0]) 0x20
(cuda-gdb) print array[0]@4
$2 = {0, 128, 64, 192}
```

You can also access the shared memory indexed into the starting offset to see what the stored values are:

```
(cuda-gdb) print *(@shared int*)0x20
$3 = 0
(cuda-gdb) print *(@shared int*)0x24
$4 = 128
(cuda-gdb) print *(@shared int*)0x28
$5 = 64
```

The example below shows how to access the starting address of the input parameter to the kernel.

```
(cuda-gdb) print &data
$6 = (const @global void * const @parameter *) 0x10
(cuda-gdb) print *(@global void * const @parameter *) 0x10
$7 = (@global void * const @parameter) 0x110000
```

Inspecting Textures

Note: The debugger can always read/write source variables when the PC is on the first assembly instruction of a source instruction. When doing assembly-level debugging, the value of source variables is not always accessible.

To inspect texture, use the print command while de-referencing the texture recast to the type of the array it is bound to. For instance, if texture tex is bound to array A of type float*, use:

```
(cuda-gdb) print *(float *)tex
```

All the array operators, such as [], can be applied to (float *)tex:

```
(cuda-gdb) print ((float *)tex)[2]
(cuda-gdb) print ((float *)tex)[2]@4
```

Info CUDA Commands

These are commands that display information about the GPU and the application's CUDA state. The available options are:

- **devices**: information about all the devices
- **sms**: information about all the SMs in the current device
- **warps**: information about all the warps in the current SM
- **lanes**: information about all the lanes in the current warp
- **kernels**: information about all the active kernels
- **blocks**: information about all the active blocks in the current kernel
- **threads**: information about all the active threads in the current kernel

A filter can be applied to every **'info cuda'** command. The filter restricts the scope of the command. A filter is composed of one or more restrictions. A restriction can be any of the following:

- **device *n***

- **sm *n***
- **warp *n***
- **lane *n***
- **kernel *n***
- **grid *n***
- **block *x*[,*y*] or block (*x*[,*y*])**
- **thread *x*[,*y*[,*z*]] or thread (*x*[,*y*[,*z*]])**

where **n, x, y, z** are integers, or one of the following special keywords: **'current'**, **'any'**, and **'all'**. **'current'** indicates that the corresponding value in the current focus should be used. **'any'** and **'all'** indicate that any value is acceptable.

info cuda devices

This command enumerates all the GPUs in the system sorted by device index. A '*' indicates the device currently in focus. This command supports filters. The default is "device all". This command prints "No CUDA Devices" if no GPUs are found.

```
(cuda-gdb) info cuda devices
 Dev/Description/SM Type/SMs Warps/SM Lanes/Warp Max Regs/Lane/Active SMs Mask
* 0   gt200     sm_13    24        32          32      128    0x00ffffff
```

info cuda sms

This command shows all the SMs for the device and the associated active warps on the SMs. This command supports filters and the default is "device current sm all". A '*' indicates the SM is focus. The results are grouped per device.

```
(cuda-gdb) info cuda sms
 SM   Active Warps Mask
 Device 0
* 0 0xffffffffffffffff
  1 0xffffffffffffffff
  2 0xffffffffffffffff
  3 0xffffffffffffffff
  4 0xffffffffffffffff
  5 0xffffffffffffffff
  6 0xffffffffffffffff
  7 0xffffffffffffffff
  8 0xffffffffffffffff
 ...
```

info cuda warps

This command takes you one level deeper and prints all the warps information for the SM in focus. This command supports filters and the default is “device current sm current warp all”. The GPU warps information can be used to map to the application CUDA blocks using the BlockIdx.

```
(cuda-gdb) info cuda warps
 Wp /Active Lanes Mask/  Divergent Lanes Mask/Active Physical PC/Kernel/BlockIdx
Device 0 SM 0
* 0   0xffffffff           0x00000000 0x000000000000001c    0   (0,0,0)
  1   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  2   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  3   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  4   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  5   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  6   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
  7   0xffffffff           0x00000000 0x0000000000000000    0   (0,0,0)
 ...
```

info cuda lanes

This command displays all the lanes (threads) for the warp in focus. This command supports filters and the default is “device current sm current warp current lane all”. In the example below you can see that all the lanes are at the same physical PC and these can be mapped to the application's CUDA threadIdx.

```
(cuda-gdb) info cuda lanes
  Ln     State   Physical PC          ThreadIdx
Device 0 SM 0 Warp 0
*  0    active 0x000000000000008c   (0,0,0)
   1    active 0x000000000000008c   (1,0,0)
   2    active 0x000000000000008c   (2,0,0)
   3    active 0x000000000000008c   (3,0,0)
   4    active 0x000000000000008c   (4,0,0)
   5    active 0x000000000000008c   (5,0,0)
   6    active 0x000000000000008c   (6,0,0)
   7    active 0x000000000000008c   (7,0,0)
   8    active 0x000000000000008c   (8,0,0)
   9    active 0x000000000000008c   (9,0,0)
  10    active 0x000000000000008c  (10,0,0)
  11    active 0x000000000000008c  (11,0,0)
  12    active 0x000000000000008c  (12,0,0)
  13    active 0x000000000000008c  (13,0,0)
  14    active 0x000000000000008c  (14,0,0)
  15    active 0x000000000000008c  (15,0,0)
  16    active 0x000000000000008c  (16,0,0)
 ...
```

info cuda kernels

This command displays on all the active kernels on the GPU in focus. It prints the SM mask, kernel ID and the grid ID for each kernel with the associated dimensions and arguments. The kernel ID is unique across all GPUs whereas the grid ID is unique per GPU. This command supports filters and the default is “kernel all”.

```
(cuda-gdb) info cuda kernels
 Kernel Dev Grid   SMs Mask   GridDim   BlockDim    Name        Args
    1   0   2  0x00ffffff (240,1,1) (128,1,1) acos_main  parms={arg =
0x110000, res = 0x110100, n = 5}
```

info cuda blocks

This command displays all the active or running blocks for the kernel in focus. The results are grouped per kernel. This command supports filters and the default is “kernel current block all”. The outputs are coalesced by default.

```
(cuda-gdb) info cuda blocks
   BlockIdx   To BlockIdx  Count   State
 Kernel 1
 *  (0,0,0)   (191,0,0)     192    running
```

Coalescing can be turned off as follows in which case more information on the Device and the SM get displayed:

```
(cuda-gdb) set cuda coalescing off
```

The following is the output of the same command when coalescing is turned off.

```
(cuda-gdb) info cuda blocks
  BlockIdx   State    Dev SM
Kernel 1
*   (0,0,0)   running   0   0
    (1,0,0)   running   0   3
    (2,0,0)   running   0   6
    (3,0,0)   running   0   9
    (4,0,0)   running   0  12
    (5,0,0)   running   0  15
    (6,0,0)   running   0  18
    (7,0,0)   running   0  21
    (8,0,0)   running   0   1
 ...
```

info cuda threads

This command displays the application's active CUDA blocks and threads with the total count of threads in those blocks. Also displayed are the virtual PC and the associated source file and the line number information. The results are grouped per kernel. The command supports filters with default being “kernel current block all thread all”. The outputs are coalesced by default as follows:

```
(cuda-gdb) info cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count  Virtual PC    Filename  Line
Device 0 SM 0
* (0,0,0   (0,0,0)   (0,0,0)  (31,0,0)    32  0x000000000088f88c  acos.cu  376
  (0,0,0) (32,0,0) (191,0,0) (127,0,0) 24544  0x000000000088f800  acos.cu  374
 ...
```

Coalescing can be turned off as follows in which case more information is presented with the output.

```
(cuda-gdb) info cuda threads
    BlockIdx   ThreadIdx  Virtual PC           Dev SM Wp Ln   Filename  Line
 Kernel 1
*    (0,0,0)    (0,0,0) 0x000000000088f88c   0  0  0  0   acos.cu   376
     (0,0,0)    (1,0,0) 0x000000000088f88c   0  0  0  1   acos.cu   376
     (0,0,0)    (2,0,0) 0x000000000088f88c   0  0  0  2   acos.cu   376
     (0,0,0)    (3,0,0) 0x000000000088f88c   0  0  0  3   acos.cu   376
     (0,0,0)    (4,0,0) 0x000000000088f88c   0  0  0  4   acos.cu   376
     (0,0,0)    (5,0,0) 0x000000000088f88c   0  0  0  5   acos.cu   376
     (0,0,0)    (6,0,0) 0x000000000088f88c   0  0  0  6   acos.cu   376
     (0,0,0)    (7,0,0) 0x000000000088f88c   0  0  0  7   acos.cu   376
     (0,0,0)    (8,0,0) 0x000000000088f88c   0  0  0  8   acos.cu   376
     (0,0,0)    (9,0,0) 0x000000000088f88c   0  0  0  9   acos.cu   376
 ...
```

Note: In coalesced form, threads must be contiguous in order to be coalesced. If some threads are not currently running on the hardware, they will create "holes" in the thread ranges. For instance, if a kernel consist of 2 blocks of 16 threads, and only the 8 lowest threads are active, then 2 coalesced ranges will be printed: one range for block 0 thread 0 to 7, and one range for block 1 thread 0 to 7. Because threads 8-15 in block 0 are not running, the 2 ranges cannot be coalesced.

09 CONTEXT AND KERNEL EVENTS

Within cuda-gdb,"kernel" refers to your device code that executes on the GPU, while "context" refers to the virtual address space on the GPU for your kernel.

You can turn ON or OFF the display of CUDA context and kernel events to review the flow of the active contexts and kernels.

Display CUDA context events

- `(cuda-gdb)` **`set cuda context_events 1`**

  Display CUDA context events.

- `(cuda-gdb)` **`set cuda context_events 0`**

  Do not display CUDA context events.

Display CUDA kernel events

- (cuda-gdb) **`set cuda kernel_events 1`**

  Display CUDA kernel events.

- (cuda-gdb) **`set cuda kernel_events 0`**

  Do not display CUDA kernel events.

Examples of displayed events

The following are examples of context events displayed:

```
[Context Create of context 0xad2fe60 on Device 0]
```

```
[Context Pop of context 0xad2fe60 on Device 0]
```

```
[Context Destroy of context 0xad2fe60 on Device 0]
```

The following are examples of kernel events displayed:

```
[Launch of CUDA Kernel 1 (kernel3) on Device 0]
```

```
[Termination of CUDA Kernel 1 (kernel3) on Device 0]
```

010 CHECKING MEMORY ERRORS

Checking Memory Errors

The CUDA memcheck feature detects global memory violations and mis-aligned global memory accesses. This feature is off by default and can be enabled using the following variable in cuda-gdb before the application is run.

```
(cuda-gdb) set cuda memcheck on
```

Once CUDA memcheck is enabled, any detection of global memory violations and mis-aligned global memory accesses will be detected only in the run or continue mode and not while single-stepping through the code.

You can also run CUDA memory checker as a standalone tool, named `cuda-memcheck`. This tool is also part of the toolkit. Please read the related documentation for more information.

GPU Error Reporting

With improved GPU error reporting in cuda-gdb, application bugs are now easier to identify and easy to fix. The following table shows the new errors that are reported on GPUs with compute capability sm_20 and higher.

Note: Continuing the execution of your application after these errors are found can lead to application termination or indeterminate results.

Table 10.1 CUDA Exception Codes

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_0 : "Device Unknown Exception"	Not precise	Global error on the GPU	This is a global GPU error caused by the application which does not match any of the listed error codes below. This should be a rare occurrence. Potentially, this may be due to Device Hardware Stack overflows or a kernel generating an exception very close to its termination.
CUDA_EXCEPTION_1 : "Lane Illegal Address"	Precise (Requires memcheck on)	Per lane/thread error	This occurs when a thread accesses an illegal(out of bounds) global address.
CUDA_EXCEPTION_2 : "Lane User Stack Overflow"	Precise	Per lane/thread error	This occurs when a thread exceeds its stack memory limit.
CUDA_EXCEPTION_3 : "Device Hardware Stack Overflow"	Not precise	Global error on the GPU	This occurs when the application triggers a global hardware stack overflow. The main cause of this error is large amounts of divergence in the presence of function calls.

Table 10.1 CUDA Exception Codes (continued)

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_4 : "Warp Illegal Instruction"	Not precise	Warp error	This occurs when any thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5 : "Warp Out-of-range Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address that is outside the valid range of local or shared memory regions.
CUDA_EXCEPTION_6 : "Warp Misaligned Address"	Not precise	Warp error	This occurs when any thread within a warp accesses an address in the local or shared memory segments that is not correctly aligned.
CUDA_EXCEPTION_7 : "Warp Invalid Address Space"	Not precise	Warp error	This occurs when any thread within a warp executes an instruction that accesses a memory space not permitted for that instruction.
CUDA_EXCEPTION_8 : "Warp Invalid PC"	Not precise	Warp error	This occurs when any thread within a warp advances its PC beyond the 40-bit address space.
CUDA_EXCEPTION_9 : "Warp Hardware Stack Overflow"	Not precise	Warp error	This occurs when any thread in a warp triggers a hardware stack overflow. This should be a rare occurrence.
CUDA_EXCEPTION_10 : "Device Illegal Address"	Not precise	Global error	This occurs when a thread accesses an illegal(out of bounds) global address. For increased precision, use the cuda memcheck feature.

Table 10.1 CUDA Exception Codes (continued)

Exception code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_11 : "Lane Misaligned Address"	Precise (Requires memcheck on)	Per lane/thread error	This occurs when a thread accesses a global address that is not correctly aligned.

011 WALK-THROUGH EXAMPLE

This chapter presents a walk-through of cuda-gdb by debugging a sample application–called bitreverse–that performs a simple 8 bit reversal on a data set.

bitreverse.cu Source Code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // Simple 8-bit bit reversal Compute test
5
6  #define N 256
7
8  __global__ void bitreverse(void *data) {
9      unsigned int *idata = (unsigned int*)data;
10     extern __shared__ int array[];
11
12     array[threadIdx.x] = idata[threadIdx.x];
13
14     array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
15                          ((0x0f0f0f0f & array[threadIdx.x]) << 4);
16     array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
17                          ((0x33333333 & array[threadIdx.x]) << 2);
18     array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
19                           ((0x55555555 & array[threadIdx.x]) << 1);
20
21     idata[threadIdx.x] = array[threadIdx.x];
22 }
23
24 int main(void) {
25     void *d = NULL; int i;
26     unsigned int idata[N], odata[N];
27
28     for (i = 0; i < N; i++)
29         idata[i] = (unsigned int)i;
```

```
30
31      cudaMalloc((void**)&d, sizeof(int)*N);
32      cudaMemcpy(d, idata, sizeof(int)*N,
33                 cudaMemcpyHostToDevice);
34
35      bitreverse<<<1, N, N*sizeof(int)>>>(d);
36
37      cudaMemcpy(odata, d, sizeof(int)*N,
38                 cudaMemcpyDeviceToHost);
39
40      for (i = 0; i < N; i++)
41         printf("%u -> %u\n", idata[i], odata[i]);
42
43      cudaFree((void*)d);
44      return 0;
45  }
```

Walking Through the Code

1. Begin by compiling the `bitreverse.cu` CUDA application for debugging by entering the following command at a shell prompt:

   ```
   $ nvcc -g -G bitreverse.cu -o bitreverse
   ```

   This command assumes that the source file name is `bitreverse.cu` and that no additional compiler flags are required for compilation. See also "Compiling for Debugging" on page 20.

2. Start the CUDA debugger by entering the following command at a shell prompt:

   ```
   $ cuda-gdb bitreverse
   ```

3. Set breakpoints. Set both the host (`main`) and GPU (`bitreverse`) breakpoints here. Also, set a breakpoint at a particular line in the device function (`bitreverse.cu:18`).

   ```
   (cuda-gdb) break main
   Breakpoint 1 at 0x18e1: file bitreverse.cu, line 25.
   (cuda-gdb) break bitreverse
   Breakpoint 2 at 0x18a1: file bitreverse.cu, line 8.
   (cuda-gdb) break 21
   Breakpoint 3 at 0x18ac: file bitreverse.cu, line 21.
   ```

4 Run the CUDA application, and it executes until it reaches the first breakpoint (`main`) set in step 3.

```
(cuda-gdb) run
Starting program: /Users/CUDA_User1/docs/bitreverse
Reading symbols for shared libraries
..++.................................................... done

Breakpoint 1, main () at bitreverse.cu:25
25              void *d = NULL; int i;
```

5 At this point, commands can be entered to advance execution or to print the program state. For this walkthrough, continue to the device kernel.

```
(cuda-gdb) continue
Continuing.
Reading symbols for shared libraries .. done
Reading symbols for shared libraries .. done
[Context Create of context 0x80f200 on Device 0]
[Launch of CUDA Kernel 0 (bitreverse<<<(1,1,1),(256,1,1)>>>) on Device 0]
Breakpoint 3 at 0x8667b8: file bitreverse.cu, line 21.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0),
device 0, sm 0, warp 0, lane 0]

Breakpoint 2, bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x110000) at
bitreverse.cu:9
9               unsigned int *idata = (unsigned int*)data;
```

cuda-gdb has detected that a CUDA device kernel has been reached, so it prints the current CUDA thread of focus.

6 Verify the CUDA thread of focus with the "**`info cuda threads`**" command and switch between host thread and the CUDA threads:

```
(cuda-gdb) info cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC
Filename  Line
Kernel 0
*  (0,0,0)   (0,0,0)     (0,0,0) (255,0,0)   256 0x0000000000866400
bitreverse.cu     9
(cuda-gdb) thread
[Current thread is 1 (process 16738)]
(cuda-gdb) thread 1
[Switching to thread 1 (process 16738)]
#0  0x000019d5 in main () at bitreverse.cu:34
34    bitreverse<<<1, N, N*sizeof(int)>>>(d);
(cuda-gdb) backtrace
#0  0x000019d5 in main () at bitreverse.cu:34
(cuda-gdb) info cuda kernels
  Kernel Dev Grid   SMs Mask GridDim  BlockDim       Name Args
       0   0    1 0x00000001 (1,1,1) (256,1,1) bitreverse data=0x110000
```

```
(cuda-gdb) cuda kernel 0
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0),
device 0, sm 0, warp 0, lane 0]
9     unsigned int *idata = (unsigned int*)data;
(cuda-gdb) bt
#0  bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x110000) at bitreverse.cu:9
```

7 Corroborate this information by printing the block and thread indices:

```
(cuda-gdb) print blockIdx
$1 = {x = 0, y = 0}
(cuda-gdb) print threadIdx
$2 = {x = 0, y = 0, z = 0)
```

8 The grid and block dimensions can also be printed:

```
(cuda-gdb) print gridDim
$3 = {x = 1, y = 1}
(cuda-gdb) print blockDim
$4 = {x = 256, y = 1, z = 1)
```

9 Advance kernel execution and verify some data:

```
(cuda-gdb) next
12       array[threadIdx.x] = idata[threadIdx.x];
(cuda-gdb) next
14       array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
(cuda-gdb) next
16       array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
(cuda-gdb) next
18       array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
(cuda-gdb) next


Breakpoint 3, bitreverse <<<(1,1),(256,1,1)>>> (data=0x100000) at
bitreverse.cu:21
21             idata[threadIdx.x] = array[threadIdx.x];
(cuda-gdb) print array[0]@12
$7 = {0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208}
(cuda-gdb) print/x array[0]@12
$8 = {0x0, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0, 0x10, 0x90, 0x50,
0xd0}


(cuda-gdb) print &data
$9 = (@global void * @parameter *) 0x10
(cuda-gdb) print *(@global void * @parameter *) 0x10
$10 = (@global void * @parameter) 0x100000
```

The resulting output depends on the current content of the memory location.

10 Since thread (0,0,0) reverses the value of 0, switch to a different thread to show more interesting data:

```
cuda-gdb) cuda thread 170
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread
(170,0,0), device 0, sm 0, warp 5, lane 10]
```

11 Delete the breakpoints and continue the program to completion:

```
(cuda-gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(cuda-gdb) continue
Continuing.

Program exited normally.
(cuda-gdb)
```

This concludes the cuda-gdb walkthrough.

APPENDIX A SUPPORTED PLATFORMS

The general platform and GPU requirements for running NVIDIA cuda-gdb are described in this section.

Host Platform Requirements

Mac OS

CUDA-GDB is supported on 32-bit and 64-bit MacOS X 10.6.5.

Linux

CUDA-GDB is supported on 32-bit and 64-bit editions of the following Linux distributions:

- Red Hat Enterprise Linux 4.8, 5.5 , and 6.0
- Fedora 13
- Novell SLED 11SP1
- OpenSUSE 11.2
- Ubuntu 10.10

GPU Requirements

Debugging is supported on all CUDA-capable GPUs with a compute capability of 1.1 or later. *Compute capability* is a device attribute that a CUDA application can query about; for more information, see the latest *NVIDIA CUDA Programming Guide* on the NVIDIA CUDA Zone Web site: http://developer.nvidia.com/object/gpucomputing.html.

These GPUs have a compute capability of 1.0 and are *not supported*:

GeForce 8800 GTS	Quadro FX 4600
GeForce 8800 GTX	Quadro FX 5600
GeForce 8800 Ultra	Tesla C870
Quadro Plex 1000 Model IV	Tesla D870
Quadro Plex 2100 Model S4	Tesla S870

APPENDIX B KNOWN ISSUES

The following are known issues with the current release.

- Device memory allocated via **`cudaMalloc()`** is not visible outside of the kernel function.
- On GPUs with sm_type lower than sm_20 it is not possible to step over a subroutine in the device code.
- Device allocations larger than 100 MB on Tesla GPUs, and larger than 32 MB on Fermi GPUs, may not be accessible in the debugger.
- Debugging applications with multiple CUDA contexts running on the same GPU is not supported on any GPU.
- On GPUs with sm_20, if you are debugging code in device functions that get called by multiple kernels, then setting a breakpoint in the device function will insert the breakpoint in only one of the kernels.
- In a multi-GPU debugging environment on Mac OS X with Aqua running, you may experience some visible delay while single-stepping the application.
- Setting a breakpoint on a line within a __device__ or __global__ function before its module is loaded may result in the breakpoint being temporarily set on the first line of a function below in the source code. As soon as the module for the targeted function is loaded, the breakpoint will be reset properly. In the meantime, the breakpoint may be hit, depending on the application. In those situations, the breakpoint can be safely ignored, and the application can be resumed.
- The 'scheduler-locking' option cannot be set to 'on'.
- Stepping again after stepping out of a kernel results in undetermined behavior. It is recommended to use the 'continue' command instead.
- Kernels containing printf function calls where some arguments are missing cannot be debugged.
- Any CUDA application that uses OpenGL interoperability requires an active windows server–such applications will fail to run under console mode debugging on both Linux and MAC. If the X server is running on Linux, the render GPU will not be enumerated when debugging so the application could still fail unless the application uses the OGL device enumeration to access the render GPU.

Notice

Trademarks

Copyright