CUDA-GDB (NVIDIA CUDA Debugger)

PG-00000-004_V3.0 | January 18, 2010

User Manual

TABLE OF CONTENTS

1 Introduction 1
CUDA-GDB: The NVIDIA CUDA Debugger 1
What's New in Version 3.0 2

2 CUDA-GDB Features and Extensions 3
Debugging CUDA applications on GPU hardware in real time 4
Extending the GDB debugging environment 4
Supporting an initialization file 4
Pausing CUDA execution at any function symbol or source file line number 4
Single-stepping individual warps 5
Displaying device memory in the device kernel 5
Variable Storage and Accessibility 5
Switching to any CUDA block or thread 7
Inspecting the coordinates 7
Changing the coordinates 7
cuda-gdb info commands 8
Breaking into running applications 10

3 Installation and Debug Compilation 11
Installation Instructions 11
Compiling for Debugging 12
Compiling/Debugging for Fermi and Tesla GPUs 12
Compiling for Fermi GPUs 12
Compiling for Fermi and Tesla GPUs 12

4 CUDA-GDB Walkthrough 13

Appendix A: Supported Platforms 17
Host Platform Requirements 17
GPU Requirements 17

Appendix B: Known Issues 18

01 INTRODUCTION

This document introduces CUDA-GDB, the NVIDIA® CUDA™ debugger, and describes what is new in version 3.0.

CUDA-GDB: The NVIDIA CUDA Debugger

CUDA-GDB is an extension to the standard i386/AMD64 port of GDB, the GNU Project debugger, version 6.6. It is designed to present the user with an all-in-one debugging environment capable of debugging native host code as well as CUDA code. Standard debugging features are inherently supported for host code, and additional features have been provided to support debugging CUDA code. CUDA-GDB is supported on 32-bit and 64-bit Linux.

> **Note: All information contained within this document is subject to change.**

What's New in Version 3.0

In this latest CUDA-GDB version the following improvements have been made:

- Improved Performance
  - Improved interactions with the debugger
  - Improved performance of applications being debugged
- The debugger now uses standard ELF format instead of cubins. As a consequence, most restrictions on debug compilations have been lifted:
  - Local variables are no longer forced to spill into local memory.
  - Variable-to-register mapping is now provided.
  - Added the ability to examine GPU memory spaces by casting an address.
- CUDA-GDB commands have changed. Many of the `info` commands now expose more low-level details.

  New commands:
  - `info cuda system`
  - `info cuda device`
  - `info cuda sm`
  - `info cuda warp`
  - `info cuda lane`

  Removed commands:
  - `info cuda state`
  - `info cuda threads`
- CUDA thread- and block-switching commands have been added:
  - `cuda device sm warp lane block thread`
- CUDA MemoryChecker feature is enabled which allows detection of global memory violations and mis-aligned global memory accesses. This feature is off by default and can be enabled using the the following variable in cuda-gdb before the application is run.
  - `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 `cuda-memcheck`.
- The debugger now supports applications using the CUDA driver APIs in addition to the support for runtime APIs.
- CUDA-GDB now supports JIT-path debugging.

02 CUDA-GDB FEATURES AND EXTENSIONS

Just as the CUDA programming model provides a seamless mechanism for programming host and GPU code, CUDA-GDB provides a model for seamlessly debugging both host and GPU code. CUDA-GDB provides a number of features to facilitate debugging CUDA applications:

- "Debugging CUDA applications on GPU hardware in real time" on page 4
- "Extending the GDB debugging environment" on page 4
- "Supporting an initialization file" on page 4
- "Pausing CUDA execution at any function symbol or source file line number" on page 4
- "Single-stepping individual warps" on page 5
- "Displaying device memory in the device kernel" on page 5
- "Switching to any CUDA block or thread" on page 7
- "cuda-gdb info commands" on page 8
- "Breaking into running applications" on page 10

Debugging CUDA applications on GPU hardware in real time

The goal of CUDA-GDB is to provide developers a mechanism for debugging a CUDA application on actual hardware in real time. This enables developers to verify program correctness without the potential variations introduced by simulation and emulation environments.

Extending the GDB debugging environment

GPU memory is treated as an extension to host memory, and GPU threads and blocks are treated as extensions to host threads. Furthermore, there is no difference between CUDA-GDB and GDB when debugging host code.

Supporting an initialization file

CUDA-GDB supports an initialization file, which must reside in your home directory (`~/.cuda-gdbinit`). This file accepts any CUDA-GDB command or extension as input to be processed when the **`cuda-gdb`** command is executed. It is just like the `.gdbinit` file used by standard versions of GDB, only renamed.

Pausing CUDA execution at any function symbol or source file line number

CUDA-GDB supports setting breakpoints at any host or device function residing in a CUDA application by using the function symbol name or the source file line number. This can be accomplished in the same way for either host or device code.

For example, if the kernel's function name is **`mykernel_main`**, the break command is as follows:

```
(cuda-gdb) break mykernel_main
```

The above command sets a breakpoint at a particular device location (the address of **`mykernel_main`**) and forces all resident GPU threads to stop at this location. There is currently no method to stop only certain threads or warps at a given breakpoint.

Single-stepping individual warps

CUDA-GDB supports stepping GPU code at the finest granularity of a warp. This means that typing **`next`** or **`step`** from the CUDA-GDB command line (when in the focus of device code) advances all threads in the same *warp* as the current thread of focus. In order to advance the execution of more than one warp, a breakpoint must be set at the desired location.

A special case is the stepping of the thread barrier call **`__syncthreads()`**. In this case, an implicit breakpoint is set immediately after the barrier and *all threads* are continued to this point.

It is important to note that it is not currently possible to step over a device subroutine. Since all device subroutines are implicitly inlined, CUDA-GDB always steps into a device subroutine.

Displaying device memory in the device kernel

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

- allocations made 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 shared int array–can be directly accessed in order to see what the stored values are in the array.

```
(cuda-gdb) p &array

$1 = (@shared int (*)[0]) 0x20

(cuda-gdb) p 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) p *(@shared int*)0x20

$3 = 0

(cuda-gdb) p *(@shared int*)0x24

$4 = 128

(cuda-gdb) p *(@shared int*)0x28

$5 = 64
```

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

```
(cuda-gdb) p &data

$6 = (const @global void * const @parameter *) 0x10

(cuda-gdb) p *(@global void * const @parameter *) 0x10

$7 = (@global void * const @parameter) 0x110000
```

Switching to any CUDA block or thread

To support CUDA thread and block switching, new commands have been introduced to inspect or change the logical coordinates (grid, block, thread) and the physical coordinates (device, sm, warp, lane).

Inspecting the coordinates

To see the current selection, use the 'cuda' command followed by a space-separated list of parameters. For example:

```
(cuda-gdb) cuda device sm warp lane block thread

Current CUDA focus: device 0, sm 0, warp 0, lane 0, block (0,0), thread
(0,0,0).
```

Changing the coordinates

To change the focus, specify a value to the parameter you want to change. For example:

To change the physical coordinates

```
(cuda-gdb) cuda device 0 sm 1 warp 2 lane 3

New CUDA focus: device 0, sm 1, warp 2, lane 3, grid 1, block (10,0),
thread (67,0,0).
```

To change the logical coordinates (thread parameter)

```
(cuda-gdb) cuda thread (15,0,0)

New CUDA focus: device 0, sm 1, warp 0, lane 15, grid 1, block (10,0),
thread (15,0,0).
```

To change the logical coordinates (block and thread parameters)

```
(cuda-gdb) cuda block (1,0) thread (3,0,0)

New CUDA focus: device 0, sm 3, warp 0, lane 3, grid 1, block (1,0),
thread (3,0,0).
```

Note: If the specified set of coordinates is incorrect, cuda-gdb will try to find the lowest set of valid coordinates. If 'cuda thread_selection' is set to 'logical', the lowest set of valid logical coordinates will be selected. If 'cuda thread_selection' is set to 'physical', the lowest set of physical coordinates will be selected. Use 'set cuda thread_selection' to switch the value.

For more information, use the cuda-gdb help with the 'help cuda' and 'help set cuda' commands.

cuda-gdb info commands

info cuda system

This command displays system information that includes the number of GPUs in the system with device header information for each GPU. The device header includes the GPU type, compute capability of the GPU, number of SMs per GPU, number of warps per SM, number of threads(lanes) per warp, and the number of registers per thread. Example:

```
(cuda-gdb) info cuda system

Number of devices: 1
DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32
Regs/LN: 128
```

info cuda device

This command displays the device information with an SM header in addition to the device header per GPU. The SM header lists all the SMs that are actively running CUDA blocks with the valid warp mask in each SM. The example below shows eight valid warps running on one of the SMs.

```
(cuda-gdb) info cuda device

DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32
Regs/LN: 128
SM:  0/30   valid warps: 00000000000000ff
```

info cuda sm

This command displays the warp header in addition to the SM and the device headers for every active SM. The warp header lists all the warps with valid, active and divergent lane mask information for each warp. The warp header also includes the block index within the grid to which it belongs. The example below lists eight warps with 32 active threads each. There is no thread divergence on any of the valid active warps.

```
(cuda-gdb) info cuda sm

DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32
Regs/LN: 128
SM:  0/30   valid warps: 00000000000000ff
WP:  0/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  1/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  2/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
```

```
WP:  3/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  4/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  5/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  6/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
WP:  7/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
```

info cuda warp

This command takes the detailed information one level deeper by displaying lane information for all the threads in the warps. The lane header includes all the active threads per warp. It includes the program counter in addition to the thread index within the block to which it belongs. The example below lists the 32 active lanes on the first active warp index 0.

```
(cuda-gdb) info cuda warp

DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32
Regs/LN: 128
SM:  0/30   valid warps: 00000000000000ff
WP:  0/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
LN:  0/32   pc=0x00000000000002b8   thread: (0,0,0)
LN:  1/32   pc=0x00000000000002b8   thread: (1,0,0)
LN:  2/32   pc=0x00000000000002b8   thread: (2,0,0)
LN:  3/32   pc=0x00000000000002b8   thread: (3,0,0)
LN:  4/32   pc=0x00000000000002b8   thread: (4,0,0)
LN:  5/32   pc=0x00000000000002b8   thread: (5,0,0)
LN:  6/32   pc=0x00000000000002b8   thread: (6,0,0)
LN:  7/32   pc=0x00000000000002b8   thread: (7,0,0)
LN:  8/32   pc=0x00000000000002b8   thread: (8,0,0)
LN:  9/32   pc=0x00000000000002b8   thread: (9,0,0)
LN: 10/32   pc=0x00000000000002b8   thread: (10,0,0)
LN: 11/32   pc=0x00000000000002b8   thread: (11,0,0)
LN: 12/32   pc=0x00000000000002b8   thread: (12,0,0)
LN: 13/32   pc=0x00000000000002b8   thread: (13,0,0)
LN: 14/32   pc=0x00000000000002b8   thread: (14,0,0)
LN: 15/32   pc=0x00000000000002b8   thread: (15,0,0)
LN: 16/32   pc=0x00000000000002b8   thread: (16,0,0)
LN: 17/32   pc=0x00000000000002b8   thread: (17,0,0)
LN: 18/32   pc=0x00000000000002b8   thread: (18,0,0)
LN: 19/32   pc=0x00000000000002b8   thread: (19,0,0)
LN: 20/32   pc=0x00000000000002b8   thread: (20,0,0)
LN: 21/32   pc=0x00000000000002b8   thread: (21,0,0)
LN: 22/32   pc=0x00000000000002b8   thread: (22,0,0)
LN: 23/32   pc=0x00000000000002b8   thread: (23,0,0)
LN: 24/32   pc=0x00000000000002b8   thread: (24,0,0)
LN: 25/32   pc=0x00000000000002b8   thread: (25,0,0)
LN: 26/32   pc=0x00000000000002b8   thread: (26,0,0)
LN: 27/32   pc=0x00000000000002b8   thread: (27,0,0)
LN: 28/32   pc=0x00000000000002b8   thread: (28,0,0)
LN: 29/32   pc=0x00000000000002b8   thread: (29,0,0)
LN: 30/32   pc=0x00000000000002b8   thread: (30,0,0)
LN: 31/32   pc=0x00000000000002b8   thread: (31,0,0)
```

info cuda lane

This command displays information per thread level if you are not interested in the warp level information for every thread.

```
(cuda-gdb) info cuda lane

DEV:  0/1    Device Type: gt200   SM Type: sm_13   SM/WP/LN: 30/32/32
Regs/LN: 128
SM:  0/30   valid warps: 00000000000000ff
WP:  0/32   valid/active/divergent lanes: 0xffffffff/0xffffffff/
0x00000000   block: (0,0)
LN:  0/32   pc=0x00000000000001b8   thread: (0,0,0)
```

Breaking into running applications

CUDA-GDB provides support for debugging kernels that appear to be hanging or looping indefinitely. The CTRL+C signal freezes the GPU and reports back the source code location. At this point, the program can be modified and then either resumed or terminated at the developer'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.

03 INSTALLATION AND DEBUG COMPILATION

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 NVIDIA CUDA-GDB.

1. Visit the NVIDIA CUDA Zone download page:

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

4 Select the appropriate Linux operating system.

   (See "Host Platform Requirements" on page 17.)

5 Download and install the 3.0 CUDA Driver.

6 Download and install the 3.0 CUDA Toolkit.

   This installation should point the environment variable `LD_LIBRARY_PATH` to `/usr/local/cuda/lib` and should also include `/usr/local/cuda/bin` in the environment variable `PATH`.

7 Download and install the 3.0 CUDA Debugger.

Compiling for Debugging

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** (mostly unoptimized) compilation
- makes the compiler include symbolic debugging information in the executable

Note: It is currently not possible to generate debugging information when compiling with the **-cubin** option.

Compiling/Debugging for Fermi and Tesla GPUs

Compiling for Fermi GPUs

If you are using the latest Fermi board, add the following flags to target Fermi output when compiling the application:

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

Compiling for Fermi and Tesla GPUs

If you are targetting both Fermi and Telsa GPUs, include these two flags:

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

04 CUDA-GDB WALKTHROUGH

This chapter presents a CUDA-GDB walk-through of twelve steps based on the following source code, `bitreverse.cu`, which performs a simple 8-bit bit reversal on a data set.

```
#include <stdio.h>
#include <stdlib.h>

// Simple 8-bit bit reversal Compute test

#define N 256

__global__ void bitreverse(unsigned int *data)
{
    unsigned int *idata = data;

    unsigned int x = idata[threadIdx.x];

    x = ((0xf0f0f0f0 & x) >> 4) | ((0x0f0f0f0f & x) << 4);
    x = ((0xcccccccc & x) >> 2) | ((0x33333333 & x) << 2);
    x = ((0xaaaaaaaa & x) >> 1) | ((0x55555555 & x) << 1);

    idata[threadIdx.x] = x;
}

int main(void)
{
    unsigned int *d = NULL; int i;
    unsigned int idata[N], odata[N];

    for (i = 0; i < N; i++)
        idata[i] = (unsigned int)i;

    cudaMalloc((void**)&d, sizeof(int)*N);
    cudaMemcpy(d, idata, sizeof(int)*N,
                   cudaMemcpyHostToDevice);

```

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

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 the source file name to be `bitreverse.cu` and that no additional compiler flags are required for compilation. See also "Compiling for Debugging" on page 12.

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 0x8051e8c: file bitreverse.cu, line 23.
(cuda-gdb) break bitreverse
Breakpoint 2 at 0x805b4f6: file bitreverse.cu, line 10.
(cuda-gdb) break bitreverse.cu:18
Breakpoint 3 at 0x805b4fb: file bitreverse.cu, line 18.
```

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

```
(cuda-gdb) run
Breakpoint 1, main() at bitreverse.cu:23
          unsigned int *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.
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 2, bitreverse() at bitreverse.cu:10
          unsigned int *idata = 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 **thread** command:

```
(cuda-gdb) thread
[Current Thread 2 (Thread 1584864 (LWP 9146))]
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]
```

The above output indicates that the host thread of focus has LWP ID `9146` and the current CUDA thread has block coordinates (`0,0`) and thread coordinates (`0,0,0`).

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

```
(cuda-gdb) thread <<<170>>>
Switching to <<<(0,0),(170,0,0)>>> bitreverse () at
bitreverse.cu:10
          unsigned int *idata = data;
```

10 Advance the execution to verify the data value that thread (`170,0,0`) should be working on:

```
(cuda-gdb) next
[Current CUDA Thread <<<(0,0),(170,0,0)>>>]
bitreverse () at bitreverse.cu:12
      unsigned int x = idata[threadIdx.x];
(cuda-gdb) next
[Current CUDA Thread <<<(0,0),(170,0,0)>>>]
bitreverse () at bitreverse.cu:14
      x = ((0xf0f0f0f0 & x) >> 4) | ((0x0f0f0f0f &x) << 4);
(cuda-gdb) print x
$5 = 170
(cuda-gdb) print/x x
$6 = 0xaa
```

This verifies thread (`170,0,0`) is working on the correct data (`170`).

11 Use the last breakpoint (set at `bitreverse.cu:18`) to verify that the logic is correct to reverse the original data:

```
(cuda-gdb) continue
Continuing.
[Current CUDA Thread <<<(0,0),(170,0,0)>>>]

Breakpoint 3, bitreverse() at bitreverse.cu:18
         idata[threadIdx.x] = x;
(cuda-gdb) print x
$7 = 85
(cuda-gdb) print/x x
$8 = 0x55
```

12 Delete the breakpoints and continue the program to completion:

```
(cuda-gdb) delete b
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

NVIDIA supports CUDA-GDB on the 32-bit and 64-bit Linux distributions listed below:

- Red Hat Enterprise Linux 5.*3*
- Red Hat Enterprise Linux 4.*8*
- Fedora 10
- Novell SLED 11
- openSUSE 11.1
- Ubuntu 9.04

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://www.nvidia.com/object/cuda_home.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.

- X11 cannot be running on the GPU that is used for debugging because the debugger effectively makes the GPU look hung to the X server, resulting in a deadlock or crash. Two possible debugging setups exist:
  - remotely accessing a single GPU (using VNC, ssh, etc.)
  - using two GPUs, where X11 is running on only one

> **Note:** The CUDA driver automatically excludes the device used by X11 from being picked by the application being debugged. This can change the behavior of the application.

- Multi-GPU applications are *not* supported.

  CUDA-GDB can debug only CUDA applications that use one GPU.

  In multi-GPU applications, the CUDA driver exposes only one GPU to the application that is being debugged. This can alter the multi-GPU application's behavior under the debugger.
- The debugger enforces blocking kernel launches.
- Device memory allocated via **`cudaMalloc()`** is not visible outside of the kernel function.
- Host memory allocated with **`cudaMallocHost()`** is not visible in CUDA-GDB.
- Not all illegal program behavior can be caught in the debugger; examples include out-of-bounds memory accesses or divide-by-zero situations.
- It is not possible to step over a subroutine in the device code.
- Multi-threaded applications may not work.
- Device allocations larger than 100 MB on Tesla GPUs, and larger than 32 MB on Fermi GPUs, may not be accessible in the debugger.
- Breakpoints in divergent code may not behave as expected.

- Debugging applications using textures is not supported.

  CUDA-GDB may output the following error message when setting breakpoints in kernels using textures:

  “Cannot access memory at address 0x0”.

Notice

Trademarks

Copyright

www.nvidia.com