NVIDIA CUDA DEBUGGER

CUDA-GDB

User Manual

PG-00000-004_V2.3
June, 2009

Published by
NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

Table of Contents

CHAPTER

1

Introduction

CUDA-GDB, the NVIDIA® CUDA™ debugger, is introduced, and what is new in the 2.3 Beta version is described.

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. Starting with the 2.2 Beta release, 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 the 2.3 Beta Version

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

- The number of supported Linux platforms has been increased. See "Host Platform Requirements" on page 15 for the current list.
- Scope shadowing is supported. If a variable is introduced in an inner scope and has the same name as a variable in an outer scope, the inner scope variable's value can now be seen.
- CUDA-GDB is now integrated into the toolkit installer.

CHAPTER

2

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

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`**

Displaying CUDA state information

CUDA-GDB provides a command, **`info cuda state`**, that displays information such as the current GPU being used and memory that has been allocated with **`cudaMalloc()`**.

Displaying CUDA blocks and threads

The CUDA-GDB command, **`info cuda threads`**, displays a summary of all CUDA threads that are currently resident on the GPU. CUDA threads are specified using the syntax described in "Switching to any CUDA block or thread" on page 7 and are summarized by grouping all contiguous threads that are stopped at the same program location. A sample display is shown below:

```
<<<(0,0),(0,0,0)>>> … <<<(0,0),(31,0,0)>>>
   GPUBlackScholesCallPut() at blackscholes.cu:73
<<<(0,0),(32,0,0)>>> … <<<(119,0),(0,0,0)>>>
   GPUBlackScholesCallPut() at blackscholes.cu:72
```

The above example shows that 32 threads (a warp) have been advanced to line `73` of `blackscholes.cu` and that the remainder of the resident threads stopped at line `72`.

Since this summary only shows thread coordinates for the start and end range, it may be unclear how many threads or blocks are actually within the displayed range. This can be checked by printing the dimension values **`gridDim`** and **`blockDim`**.

CUDA-GDB also has the ability to display a full list of each individual thread that is currently resident on the GPU by using the command **`info cuda threads all`**.

Switching to any CUDA block or thread

To support CUDA thread and block switching, CUDA-GDB provides an extension to the GDB **thread** command that uses the CUDA syntax as follows:

```
thread <<<(BX,BY),(TX,TY,TZ)>>>
```

This extension supports multiple variations.

- Providing fewer coordinates for the CUDA thread or block than are indicated sets the specified coordinates and clears all others to 0:

  The command **thread <<<(0),(1)>>>** switches to the CUDA block with X coordinate 0 and Y coordinate 0 and to the CUDA thread with X coordinate 1 and Y and Z coordinates 0. This is the same as the command **thread <<<(0,0),(1,0,0)>>>**.

- Providing only the CUDA thread coordinates maintains the current block of focus while switching to the specified CUDA thread:

  The command **thread <<<(10)>>>** maintains the current CUDA block and switches to the CUDA thread with X coordinate 10 and Y and Z coordinates 0. It is a shorthand version of the previous command, **thread <<<(0),(1)>>>**, and only works for specifying threads within a current block.

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.

CHAPTER

3

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`.
2. Select the appropriate Linux operating system. (See "Host Platform Requirements" on page 15.)
3. Download and install the 2.3 Beta CUDA Driver.
4. Download and install the 2.3 Beta 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`.
5. Download and install the 2.3 Beta 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, which spills all variables to local memory
- 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.

CHAPTER

4

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.

```
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(unsigned int *data)
9  {
10     unsigned int *idata = data;
11
12     unsigned int x = idata[threadIdx.x];
13
14     x = ((0xf0f0f0f0 & x) >> 4) | ((0x0f0f0f0f & x) << 4);
15     x = ((0xcccccccc & x) >> 2) | ((0x33333333 & x) << 2);
16     x = ((0xaaaaaaaa & x) >> 1) | ((0x55555555 & x) << 1);
17
18     idata[threadIdx.x] = x;
```

```
19 }
20
21 int main(void)
22 {
23     unsigned int *d = NULL; int i;
24     unsigned int idata[N], odata[N];
25
26     for (i = 0; i < N; i++)
27         idata[i] = (unsigned int)i;
28
29     cudaMalloc((void**)&d, sizeof(int)*N);
30     cudaMemcpy(d, idata, sizeof(int)*N,
31                cudaMemcpyHostToDevice);
32
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 9.

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.*x*
- Red Hat Enterprise Linux 4.*x*
- Fedora 10
- Novell SLED 11
- Novell SLED 10 SP2
- openSUSE 11.1
- Ubuntu 9.04
- Ubuntu 8.10

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://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 to be 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: Starting with CUDA 2.2 Beta, 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, and the CUDA driver supports this by letting only one GPU remain visible to an application that is being debugged.

Note: Because the CUDA driver, starting with CUDA 2.3 Beta, automatically excludes all but one GPU when an application is being debugged, the application's behavior could be affected.

- 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 currently possible to step over a subroutine in device code.
- Debugging using the device driver API is not supported.