# CUDA-GDB
# CUDA DEBUGGER

DU-05227-042 _v7.0 | March 2015

**User Manual**

# TABLE OF CONTENTS

# LIST OF TABLES

# Chapter 1. INTRODUCTION

This document introduces CUDA-GDB, the NVIDIA® CUDA® debugger for Linux and Mac OS.

## 1.1. 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 x86-64 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. CUDA-GDB is based on GDB 7.6 on both Linux and Mac OS X.

## 1.2. 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 debugging C/C++ and Fortran CUDA applications. (Fortran debugging support is limited to 64-bit Linux operating system) 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_20** or **sm_30**, but also supports debugging kernels compiled at runtime, referred to as just-in-time compilation, or JIT compilation for short.

## 1.3. 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.

# Chapter 2. RELEASE NOTES

## 7.0 Release

### GPU core dump support

CUDA-GDB supports reading GPU and GPU+CPU core dump files.
New environment variables: **`CUDA_ENABLE_COREDUMP_ON_EXCEPTION`**, **`CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION`** and **`CUDA_COREDUMP_FILE`** can be used to enable and configure this feature.

## 6.5 Release

### CUDA Fortran Support

CUDA-GDB supports CUDA Fortran debugging on 64-bit Linux operating systems.

### GDB 7.6.2 Code Base

The code base for CUDA-GDB was upgraded to GDB 7.6.2.

## 6.0 Release

### Unified Memory Support

Managed variables can be read and written from either a host thread or a device thread. The debugger also annotates memory addresses that reside in managed memory with **`@managed`**. The list of statically allocated managed variables can be accessed through a new **`info cuda managed`** command.

### GDB 7.6 Code Base

The code base for CUDA-GDB was upgraded from GDB 7.2 to GDB 7.6.

### Android Support

CUDA-GDB can now be used to debug Android native applications either locally or remotely.

**Single-Stepping Optimizations**

CUDA-GDB can now use optimized methods to single-step the program, which accelerate single-stepping most of the time. This feature can be disabled by issuing **`set cuda single_stepping_optimizations off`**.

**Faster Remote Debugging**

A lot of effort has gone into making remote debugging considerably faster, up to 2 orders of magnitude. The effort also made local debugging faster.

**Kernel Entry Breakpoints**

The **`set cuda break_on_launch`** option will now break on kernels launched from the GPU. Also, enabling this option does not affect kernel launch notifications.

**Precise Error Attribution**

On Maxwell architecture (SM 5.0), the instruction that triggers an exception will be reported accurately. The application keeps making forward progress and the PC at which the debugger stops may not match that address but an extra output message identifies the origin of the exception.

**Live Range Optimizations**

To mitigate the issue of variables not being accessible at some code addresses, the debugger offers two new options. With **`set cuda value_extrapolation`**, the latest known value is displayed with **`(possibly)`** prefix. With **`set cuda ptx_cache`**, the latest known value of the PTX register associated with a source variable is displayed with the **`(cached)`** prefix.

**Event Notifications**

Kernel event notifications are not displayed by default any more.

New kernel events verbosity options have been added: **`set cuda kernel_events`**, **`set cuda kernel_events_depth`**. Also **`set cuda defer_kernel_launch_notifications`** has been deprecated and has no effect any more.

## 5.5 Release

**Kernel Launch Trace**

Two new commands, **`info cuda launch trace`** and **`info cuda launch children`**, are introduced to display the kernel launch trace and the children kernel of a given kernel when Dynamic Parallelism is used.

**Single-GPU Debugging (BETA)**

CUDA-GDB can now be used to debug a CUDA application on the same GPU that is rendering the desktop GUI. This feature also enables debugging of long-running or indefinite CUDA kernels that would otherwise encounter a launch timeout. In addition, multiple CUDA-GDB sessions can debug CUDA applications context-

switching on the same GPU. This feature is available on Linux with SM3.5 devices. For information on enabling this, please see Single-GPU Debugging with the Desktop Manager Running and Multiple Debuggers.

**Remote GPU Debugging**

CUDA-GDB in conjunction with CUDA-GDBSERVER can now be used to debug a CUDA application running on the remote host.

## 5.0 Release

**Dynamic Parallelism Support**

CUDA-GDB fully supports Dynamic Parallelism, a new feature introduced with the 5.0 toolkit. The debugger is able to track the kernels launched from another kernel and to inspect and modify variables like any other CPU-launched kernel.

**Attach/Detach**

It is now possible to attach to a CUDA application that is already running. It is also possible to detach from the application before letting it run to completion. When attached, all the usual features of the debugger are available to the user, as if the application had been launched from the debugger. This feature is also supported with applications using Dynamic Parallelism.

**Attach on exception**

Using the environment variable **`CUDA_DEVICE_WAITS_ON_EXCEPTION`**, the application will run normally until a device exception occurs. Then the application will wait for the debugger to attach itself to it for further debugging.

**API Error Reporting**

Checking the error code of all the 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 **`set cuda api_failures`** for more information.

**Inlined Subroutine Support**

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.

## 4.2 Release

**Kepler Support**

The primary change in Release 4.2 of CUDA-GDB is the addition of support for the new Kepler architecture. There are no other user-visible changes in this release.

## 4.1 Release

**Source Base Upgraded to GDB 7.2**

Until now, CUDA-GDB was based on GDB 6.6 on Linux, and GDB 6.3.5 on Darwin (the Apple branch). Now, both versions of CUDA-GDB are using the same 7.2 source base.

Now CUDA-GDB supports newer versions of GCC (tested up to GCC 4.5), has better support for DWARF3 debug information, and better C++ debugging support.

**Simultaneous Sessions Support**

With the 4.1 release, the single CUDA-GDB process restriction is lifted. Now, multiple CUDA-GDB sessions are allowed to co-exist as long as the GPUs are not shared between the applications being processed. For instance, one CUDA-GDB process can debug process foo using GPU 0 while another CUDA-GDB process debugs process bar using GPU 1. The exclusive of GPUs can be enforced with the CUDA_VISIBLE_DEVICES environment variable.

**New Autostep Command**

A new 'autostep' command was added. The command increases the precision of CUDA exceptions by automatically single-stepping through portions of code.

Under normal execution, the thread and instruction where an exception occurred may be imprecisely reported. However, the exact instruction that generates the exception can be determined if the program is being single-stepped when the exception occurs.

Manually single-stepping through a program is a slow and tedious process. Therefore 'autostep' aides the user by allowing them to specify sections of code where they suspect an exception could occur. These sections are automatically single-stepped through when the program is running, and any exception that occurs within these sections is precisely reported.

Type 'help autostep' from CUDA-GDB for the syntax and usage of the command.

**Multiple Context Support**

On GPUs with compute capability of SM20 or higher, debugging multiple contexts on the same GPU is now supported. It was a known limitation until now.

**Device Assertions Support**

The R285 driver released with the 4.1 version of the toolkit supports device assertions. CUDA_GDB supports the assertion call and stops the execution of the application when the assertion is hit. Then the variables and memory can be inspected as usual. The application can also be resumed past the assertion if needed. Use the 'set cuda hide_internal_frames' option to expose/hide the system call frames (hidden by default).

### Temporary Directory

By default, the debugger API will use /tmp as the directory to store temporary files. To select a different directory, the $TMPDIR environment variable and the API CUDBG_APICLIENT_PID variable must be set.

# Chapter 3. 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.

## 3.1. 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 X or Linux. (See Supported Platforms.)
3. Download and install the CUDA Driver.
4. Download and install the CUDA Toolkit.

## 3.2. Setting Up the Debugger Environment

### 3.2.1. Linux

Set up the PATH and LD_LIBRARY_PATH environment variables:

```
export PATH=/usr/local/cuda-7.0/bin:$PATH

export LD_LIBRARY_PATH=/usr/local/cuda-7.0/lib64:/usr/local/cuda-7.0/
lib:$LD_LIBRARY_PATH
```

### 3.2.2. Mac OS X

#### Set up environment variables

```
$ export PATH=/Developer/NVIDIA/CUDA-7.0/bin:$PATH
$ export DYLD_LIBRARY_PATH=/Developer/NVIDIA/CUDA-7.0/lib:$DYLD_LIBRARY_PATH
```

### Set permissions

The first time **cuda-gdb** is executed, a pop-up dialog window will appear to allow the debugger to take control of another process. The user must have Administrator priviledges to allow it. It is a required step.

Another solution used in the past is to add the **cuda-binary-gdb** to the **procmod** group and set the **taskgated** daemon to let such processes take control of other processes. It used to be the solution to fix the **Unable to find Mach task port for processid** error.

```
$ sudo chgrp procmod /Developer/NVIDIA/CUDA-7.0/bin/cuda-binary-gdb
$ sudo chmod 2755 /Developer/NVIDIA/CUDA-7.0/bin/cuda-binary-gdb
$ sudo chmod 755 /Developer/NVIDIA/CUDA-7.0/bin/cuda-gdb
```

To set the **taskgated** daemon to allow the processes in the **procmod** group to access Task Ports, **taskgated** must be launched with the -p option. To make it a permanent option, edit **/System/Library/LaunchDaemons/com.apple.taskgated.plist**. See **man taskgated** for more information. Here is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Inc.//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.taskgated</string>
        <key>MachServices</key>
        <dict>
                <key>com.apple.taskgated</key>
                <dict>
                        <key>TaskSpecialPort</key>
                        <integer>9</integer>
                </dict>
        </dict>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/libexec/taskgated</string>
                <string>-p</string>
                <string>-s</string>
        </array>
</dict>
</plist>
```

After editing the file, the system must be rebooted or the daemon stopped and relaunched for the change to take effect.

Using the **taskgated**, as every application in the procmod group will have higher priviledges, adding the -p option to the taskgated daemon is a possible security risk.

### Debugging in the console mode

While debugging the application in console mode, it is not uncommon to encounter kernel warnings about unnesting DYLD shared regions for a debugger or a debugged process that look as follows:

```
cuda-binary-gdb (map: 0xffffff8038644658) triggered DYLD shared region unnest
 for map: 0xffffff8038644bc8, region 0x7fff95e00000->0x7fff96000000. While not
 abnormal for debuggers, this increases system memory footprint until the target
 exits.
```

To prevent such messages from appearing, make sure that the **vm.shared_region_unnest_logging** kernel parameter is set to zero, for example, by using the following command:

```
$ sudo sysctl -w vm.shared_region_unnest_logging=0
```

### 3.2.3. Temporary Directory

By default, CUDA-GDB uses **/tmp** as the directory to store temporary files. To select a different directory, set the **$TMPDIR** environment variable.

The user must have write and execute permission to the temporary directory used by CUDA-GDB. Otherwise, the debugger will fail with an internal error.

Since **/tmp** folder does not exist on Android device, the **$TMPDIR** environment variable must be set and point to a user-writeable folder before launching cuda-gdb.

## 3.3. Compiling the Application

### 3.3.1. 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 debug information in the executable

To compile your CUDA Fortran code with debgging information necessary for CUDA-GDB to work properly, pgfortran, the PGI CUDA Fortran compiler, must be invoked with **-g** option. Also, for the ease of debugging and forward compatibility with the future GPU architectures, it is recommended to compile the code with **-Mcuda=nordc** option; for example,

```
pgfortran -g -Mcuda=nordc foo.cuf -o foo
```

For more information about the available compilation flags, please consult the PGI compiler documentation.

### 3.3.2. Compiling For Specific GPU architectures

By default, the compiler will only generate code for the compute_20 PTX and sm_20 cubins. For later GPUs, the kernels are recompiled at runtime from the PTX for the architecture of the target GPU(s). Compiling for a specific virtual architecture guarantees that the application will work for any GPU architecture after that, for a trade-off in performance. This is done for forward-compatibility.

It is highly recommended to compile the application once and for all for the GPU architectures targeted by the application, and to generate the PTX code for the latest virtual architecture for forward compatibility.

A GPU architecture is defined by its compute capability. The list of GPUs and their respective compute capability, see https://developer.nvidia.com/cuda-gpus. The same application can be compiled for multiple GPU architectures. Use the **`-gencode`** compilation option to dictacte which GPU architecture to compile for. The option can be specified multiple times.

For instance, to compile an application for a GPU with compute capability 3.0, add the following flag to the compilation command:

```
-gencode arch=compute_30,code=sm_30
```

To compile PTX code for any future architecture past the compute capability 3.5, add the following flag to the compilation command:

```
-gencode arch=compute_35,code=compute_35
```

For additional information, please consult the compiler documentation at http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#extended-notation

## 3.4. 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:

### 3.4.1. 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

On Linux you can stop the X11 server by stopping the **`lightdm`** service, or the equivalent for the target Linux distribution.

**On Mac OS X**

On Mac OS X you can log in with **>console** as the user name in the desktop UI login screen.

To enable console login option, open the System Prerences->Users & Group->Login Options tab, set automatic login option to Off, and set **Display login window as** to **Name and password**.

To launch/debug cuda applications in console mode on systems with an integrated GPU and a discrete GPU, also make sure that the **Automatic Graphics Switching** option in the System Settings->Energy Saver tab is unchecked.

## 3.4.2. Single-GPU Debugging with the Desktop Manager Running

CUDA-GDB can be used to debug CUDA applications on the same GPU that is running the desktop GUI.

This is a BETA feature available on Linux and supports devices with SM3.5 compute capability.

There are two ways to enable this functionality:

- Use the following command:

```
set cuda software_preemption on
```

- Export the following environment variable:

```
CUDA_DEBUGGER_SOFTWARE_PREEMPTION=1
```

Either of the options above will activate software preemption. These options must be set **prior** to running the application. When the GPU hits a breakpoint or any other event that would normally cause the GPU to freeze, CUDA-GDB releases the GPU for use by the desktop or other applications. This enables CUDA-GDB to debug a CUDA application on the same GPU that is running the desktop GUI, and also enables debugging of multiple CUDA applications context-switching on the same GPU.

The options listed above are ignored for GPUs with less than SM3.5 compute capability.

## 3.4.3. Multi-GPU Debugging

Multi-GPU debugging designates the scenario where the application is running on more than one CUDA-capable device. 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 a 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.

If the **`CUDA_VISIBLE_DEVICES`** environment is used, only the specified devices are suspended and resumed.

All CUDA-capable GPUs may run one or more kernels. To switch to an active kernel, use **`cuda kernel <n>`**, where **`n`** is the ID of the kernel retrieved from **`info cuda kernels`**.

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.

## 3.4.4. 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.

## 3.4.5. 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 might alter 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 the Aqua desktop manager. To determine which GPU should be used for CUDA, run the **`1_Utilities/deviceQuery`** CUDA sample. A truncated example output of **`deviceQuery`** is shown below.

```
Detected 2 CUDA Capable device(s)

Device 0: "GeForce GT 330M"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    1.2
  Total amount of global memory:                 512 MBytes (536543232 bytes)
  ( 6) Multiprocessors x (  8) CUDA Cores/MP:    48 CUDA Cores
  [... truncated output ...]

Device 1: "Quadro K5000"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 4096 MBytes (4294508544 bytes)
  ( 8) Multiprocessors x (192) CUDA Cores/MP:    1536 CUDA Cores
  [... truncated output ...]

deviceQuery, CUDA Driver = CUDART, \
   CUDA Driver Version = 7.0, CUDA Runtime Version = 7.0, \
   NumDevs = 2, Device0 = GeForce GT 330M, Device1 = Quadro K5000
```

If Device 0 is rendering the desktop, then Device 1 must be selected for running and debugging the CUDA application. This exclusion of a device can be achieved by setting the **CUDA_VISIBLE_DEVICES** environment variable to the index of the device that will be used for CUDA. In this particular example, the value would be 1:

```
export CUDA_VISIBLE_DEVICES=1
```

As a safeguard mechanism, cuda-gdb will detect if a visible device is also used for display and return an error. To turn off the safeguard mechanism, the **set cuda gpu_busy_check** should be set to **off**.

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

## 3.4.6. Remote Debugging

There are multiple methods to remote debug an application with CUDA_GDB. In addition to using SSH or VNC from the host system to connect to the target system, it is also possible to use the **target remote** GDB feature. Using this option, the local **cuda-gdb** (client) connects to the **cuda-gdbserver** process (the server) running on the target system. This option is supported with a Linux or Mac OS X client and a Linux server. It is not possible to remotely debug a CUDA application running on Mac OS X.

Setting remote debugging that way is a 2-step process:

### Launch the cuda-gdbserver on the remote host

cuda-gdbserver can be launched on the remote host in different operation modes.

- Option 1: Launch a new application in debug mode.

  To launch a new application in debug mode, invoke cuda-gdb server as follows:

  ```
  $ cuda-gdbserver :1234 app_invocation
  ```

Where **1234** is the TCP port number that **cuda-gdbserver** will listen to for incoming connections from **cuda-gdb**, and **app-invocation** is the invocation command to launch the application, arguments included.

- Option 2: Attach **cuda-gdbserver** to the running process

  To attach cuda-gdbserver to an already running process, the **--attach** option followed by process identification number (PID) must be used:

  ```
  $ cuda-gdbserver :1234 --attach 5678
  ```

  Where **1234** is the TCP port number and **5678** is process identifier of the application cuda-gdbserver must be attached to.

When debugging a 32-bit application on a 64-bit server, **cuda-gdbserver** must also be 32-bit.

#### Launch cuda-gdb on the client

Configure **cuda-gdb** to connect to the remote target using either:

```
(cuda-gdb) target remote
```

or

```
(cuda-gdb) target extended-remote
```

It is recommended to use **set sysroot** command if libraries installed on the debug target might differ from the ones installed on the debug host. For example, cuda-gdb could be configured to connect to remote target as follows:

```
(cuda-gdb) set sysroot remote://
(cuda-gdb) target remote 192.168.0.2:1234
```

Where **192.168.0.2** is the IP address or domain name of the remote target, and **1234** is the TCP port previously previously opened by **cuda-gdbserver**.

## 3.4.7. Multiple Debuggers

In a multi-GPU environment, several debugging sessions may take place simultaneously as long as the CUDA devices are used exclusively. For instance, one instance of CUDA-GDB can debug a first application that uses the first GPU while another instance of CUDA-GDB debugs a second application that uses the second GPU. The exclusive use of a GPU is achieved by specifying which GPU is visible to the application by using the **CUDA_VISIBLE_DEVICES** environment variable.

```
$ CUDA_VISIBLE_DEVICES=1 cuda-gdb my_app
```

With software preemption enabled (**set cuda software_preemption on**), multiple CUDA-GDB instances can be used to debug CUDA applications context-switching on the same GPU. The **--cuda-use-lockfile=0** option must be used when starting each debug session, as mentioned in --cuda-use-lockfile.

```
$ cuda-gdb --cuda-use-lockfile=0 my_app
```

## 3.4.8. Attaching/Detaching

CUDA-GDB can attach to and detach from a CUDA application running on GPUs with compute capability 2.0 and beyond, using GDB's built-in commands for attaching to or detaching from a process.

Additionally, if the environment variable CUDA_DEVICE_WAITS_ON_EXCEPTION is set to 1 prior to running the CUDA application, the application will run normally until a device exception occurs. The application will then wait for CUDA-GDB to attach itself to it for further debugging.

By default on Ubuntu Linux debugger cannot attach to an already running processes. In order to enable the attach feature of CUDA debugger, either cuda-gdb should be laucnhed as root, or **/proc/sys/kernel/yama/ptrace_scope** should be set to zero, using the following command:

```
$ sudo sh -c "echo 0 >/proc/sys/kernel/yama/ptrace_scope"
```

To make the change permanent, please edit **/etc/sysctl.d/10-ptrace.conf**.

## 3.4.9. CUDA/OpenGL Interop Applications on Linux

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 OS X. However, 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 OpenGL device enumeration to access the render GPU. But if the X session is running in non-interactive mode while using the debugger, the render GPU will be enumerated correctly.

1. Launch your X session in non-interactive mode.
   a) Stop your X server.
   b) Edit **/etc/X11/xorg.conf** to contain the following line in the Device section corresponding to your display:

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

   c) Restart your X server.
2. Log in remotely (SSH, etc.) and launch your application under CUDA-GDB.
   This setup works properly for single-GPU and multi-GPU configurations.
3. Ensure your **DISPLAY** environment variable is set appropriately.
   For example:

   ```
   export DISPLAY=:0.0
   ```

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

# Chapter 4. CUDA-GDB EXTENSIONS

## 4.1. 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
```

## 4.2. 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
```

Moreover, all the CUDA commands can be auto-completed by pressing the TAB key, as with any other GDB command.

## 4.3. 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.

## 4.4. GUI Integration

### Emacs

CUDA-GDB works with GUD in Emacs and XEmacs. No extra step is required other than 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**.

## 4.5. GPU core dump support

### Enabling core dump generation on exception

Set the **CUDA_ENABLE_COREDUMP_ON_EXCEPTION** environment variable to **1** in order to enable generating a GPU core dump when GPU exception is encountered. This option is disabled by default.

Set the **CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION** environment variable to **0** in order to disable generating a CPU core dump when GPU exception is encountered. This option is enabled by default when GPU core dump generation is enabled.

To change the default core dump file name, set the **CUDA_COREDUMP_FILE** environment variable to a specific file name. The default core dump file name is in the following format: **core.cuda.HOSTNAME.PID** where HOSTNAME is the host name of machine running the CUDA application and PID is the process identifier of the CUDA application.

### Inspecting GPU and GPU+CPU core dumps in cuda-gdb

Use the following command to load the GPU core dump into the debugger

- `(cuda-gdb) target cudacore core.cuda.localhost.1234`

This will open the core dump file and print the exception encountered during program execution. Then, issue standard cuda-gdb commands to further investigate application state on the device at the moment it was aborted.

Use the following command to load CPU and GPU core dumps into the debugger

- `(cuda-gdb) target core core.cpu core.cuda`

  This will open the core dump file and print the exception encountered during program execution. Then, issue standard cuda-gdb commands to further investigate application state on the host and the device at the moment it was aborted.

# Chapter 5. 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.

## 5.1. 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.

**Note: If software preemption is enabled (`set cuda software_preemption on`), hardware coordinates corresponding to a device thread are likely to change upon resuming execution on the device. However, software coordinates will remain intact and will not change for the lifetime of the device thread.**

## 5.2. 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
```

## 5.3. 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.
```

# Chapter 6. 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

## 6.1. 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 GPUs are 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 launched outside the debugger.

## 6.2. 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 active threads in the warp currently in focus. The divergent threads in the warp are not single-stepped.

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.

With Dynamic Parallelism on **sm_35**, several CUDA APIs can now be instantiated from the device. The following list defines single-step behavior when encountering these APIs:

- When encountering device side kernel launches (denoted by the **<<<>>>** launch syntax), the **step** and **next** commands will have the same behavior, and both will **step over** the launch call.
- When encountering **cudaDeviceSynchronize**, the launch synchronization routine, the **step** and **next** commands will have the same behavior, and both will **step over** the call. When stepping over the call, the **entire device** is resumed until the call has completed, at which point the device is suspended (without user intervention).
- When stepping a device grid launch to completion, focus will automatically switch back to the CPU. The **cuda kernel** focus switching command must be used to switch to another grid of interest (if one is still resident).

It is not possible to **step into** a device launch call (nor the routine launched by the call).

# Chapter 7. BREAKPOINTS & WATCHPOINTS

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.

When a breakpoint is hit by 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.

## 7.1. 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
```

## 7.2. 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.

## 7.3. 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.

## 7.4. 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
```

See set cuda break_on_launch for more information.

## 7.5. 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.

Conditional breakpoints can be set on code from CUDA modules that are not already loaded. The verification of the condition will then only take place when the ELF image of

that module is loaded. Therefore any error in the conditional expression will be deferred from the instantion of the conditional breakpoint to the moment the CUDA module is loaded. If unsure, first set an unconditional breakpoint at the desired location and add the conditional statement the first time the breakpoint is hit by using the **`cond`** command.

## 7.6. Watchpoints

Watchpoints on CUDA code are not supported.

Watchpoints on host code are supported. The user is invited to read the GDB documentation for a tutorial on how to set watchpoints on host code.

# Chapter 8. INSPECTING PROGRAM STATE

## 8.1. 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`**

## 8.2. 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</>
```

## 8.3. Inspecting Textures

**The debugger can always read/write the 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 a 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 *(@texture float *)tex
```

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

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

## 8.4. 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

**`launch trace`**
information about the parent kernels of the kernel in focus

**`launch children`**
information about the kernels launched by the kernels in focus

**`contexts`**
information about all the contexts

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]])**
- **breakpoint all** and **breakpoint n**

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.

The **breakpoint all** and **breakpoint n** filter are only effective for the **info cuda threads** command.

## 8.4.1. 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 PCI Bus/Dev ID                 Name Description SM Type SMs Warps/SM Lanes/
Warp Max Regs/Lane Active SMs Mask
    0        06:00.0 GeForce GTX TITAN Z      GK110B   sm_35  15       64
  32           256 0x00000000
    1        07:00.0 GeForce GTX TITAN Z      GK110B   sm_35  15       64
  32           256 0x00000000
```

## 8.4.2. 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
...
```

## 8.4.3. 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 command can be used to display which warp executes what block.

```
(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)
 ...
```

## 8.4.4. 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. The command can be used to display which lane executes what thread.

```
(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)
 ...
```

## 8.4.5. 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. The **`Parent`** column shows the kernel ID of the parent grid. This command supports filters and the default is **`kernel all`**.

```
(cuda-gdb) info cuda kernels
  Kernel Parent Dev Grid Status   SMs Mask   GridDim  BlockDim      Name Args
*      1      -   0    2 Active 0x00ffffff (240,1,1) (128,1,1) acos_main
 parms=...
```

This command will also show grids that have been launched on the GPU with Dynamic Parallelism. Kernels with a negative grid ID have been launched from the GPU, while kernels with a positive grid ID have been launched from the CPU.

With the **cudaDeviceSynchronize** routine, it is possible to see grid launches disappear from the device and then resume later after all child launches have completed.

## 8.4.6. 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
 ...
```

## 8.4.7. 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 displayed 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
 ...
```

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.

The command also supports **`breakpoint all`** and **`breakpoint breakpoint_number`** as filters. The former displays the threads that hit all CUDA breakpoints set by the user. The latter displays the threads that hit the CUDA breakpoint *breakpoint_number*.

```
(cuda-gdb) info cuda threads breakpoint all
  BlockIdx ThreadIdx         Virtual PC Dev SM Wp Ln        Filename  Line
Kernel 0
   (1,0,0)   (0,0,0) 0x0000000000948e58   0 11  0  0 infoCommands.cu    12
   (1,0,0)   (1,0,0) 0x0000000000948e58   0 11  0  1 infoCommands.cu    12
   (1,0,0)   (2,0,0) 0x0000000000948e58   0 11  0  2 infoCommands.cu    12
   (1,0,0)   (3,0,0) 0x0000000000948e58   0 11  0  3 infoCommands.cu    12
   (1,0,0)   (4,0,0) 0x0000000000948e58   0 11  0  4 infoCommands.cu    12
   (1,0,0)   (5,0,0) 0x0000000000948e58   0 11  0  5 infoCommands.cu    12

(cuda-gdb) info cuda threads breakpoint 2 lane 1
  BlockIdx ThreadIdx         Virtual PC Dev SM Wp Ln        Filename  Line
Kernel 0
   (1,0,0)   (1,0,0) 0x0000000000948e58   0 11  0  1 infoCommands.cu    12
```

## 8.4.8. info cuda launch trace

This command displays the kernel launch trace for the kernel in focus. The first element in the trace is the kernel in focus. The next element is the kernel that launched this

kernel. The trace continues until there is no parent kernel. In that case, the kernel is CPU-launched.

For each kernel in the trace, the command prints the level of the kernel in the trace, the kernel ID, the device ID, the grid Id, the status, the kernel dimensions, the kernel name, and the kernel arguments.

```
(cuda-gdb) info cuda launch trace
  Lvl Kernel Dev Grid     Status   GridDim  BlockDim Invocation
*   0      3   0   -7     Active  (32,1,1)  (16,1,1) kernel3(c=5)
    1      2   0   -5 Terminated (240,1,1) (128,1,1) kernel2(b=3)
    2      1   0    2     Active (240,1,1) (128,1,1) kernel1(a=1)
```

A kernel that has been launched but that is not running on the GPU will have a **Pending** status. A kernel currently running on the GPU will be marked as **Active**. A kernel waiting to become active again will be displayed as **Sleeping**. When a kernel has terminated, it is marked as **Terminated**. For the few cases, when the debugger cannot determine if a kernel is pending or terminated, the status is set to **Undetermined**.

This command supports filters and the default is **kernel all**.

With **set cuda software_preemption on**, no kernel will be reported as active.

## 8.4.9. info cuda launch children

This command displays the list of non-terminated kernels launched by the kernel in focus. For each kernel, the kernel ID, the device ID, the grid Id, the kernel dimensions, the kernel name, and the kernel parameters are displayed.

```
(cuda-gdb) info cuda launch children
  Kernel Dev Grid GridDim BlockDim Invocation
*      3   0   -7 (1,1,1)  (1,1,1) kernel5(a=3)
      18   0   -8 (1,1,1) (32,1,1) kernel4(b=5)
```

This command supports filters and the default is **kernel all**.

## 8.4.10. info cuda contexts

This command enumerates all the CUDA contexts running on all GPUs. A ***** indicates the context currently in focus. This command shows whether a context is currently active on a device or not.

```
(cuda-gdb) info cuda contexts
     Context Dev    State
  0x080b9518   0 inactive
* 0x08067948   0   active
```

## 8.4.11. info cuda managed

This command shows all the static managed variables on the device or on the host depending on the focus.

```
(cuda-gdb) info cuda managed
Static managed variables on device 0 are:
managed_var = 3
managed_consts = {one = 1, e = 2.71000004, pi = 3.1400000000000001}
```

## 8.5. Disassembly

The device SASS code can be disassembled using the standard GDB disassembly instructions such as **x/i** and **display/i**.

```
(cuda-gdb) x/4i $pc-32
   0xa689a8 <acos_main(acosParams)+824>: MOV R0, c[0x0][0x34]
   0xa689b8 <acos_main(acosParams)+840>: MOV R3, c[0x0][0x28]
   0xa689c0 <acos_main(acosParams)+848>: IMUL R2, R0, R3
=> 0xa689c8 <acos_main(acosParams)+856>: MOV R0, c[0x0][0x28]
```

For disassembly instruction to work properly, **cuobjdump** must be installed and present in your **$PATH**.

## 8.6. Registers

The device registers code can be inspected/modified using the standard GDB commands such as **info registers**.

```
(cuda-gdb) info registers $R0 $R1 $R2 $R3
R0             0xf0 240
R1             0xfffc48 16776264
R2             0x7800 30720
R3             0x80 128
```

The registers are also accessible as **$R<regnum>** built-in variables, for example:

```
(cuda-gdb) printf "%d %d\n", $R0*$R3, $R2
30720 30720
```

Values of predicate and CC registers can be inspecting by printing system registers group or by using their respective pseudo-names: **$P0**..**$P6** and **$CC**.

```
(cuda-gdb) info registers system
P0             0x1 1
P1             0x1 1
P2             0x0 0
P3             0x0 0
P4             0x0 0
P5             0x0 0
P6             0x1 1
CC             0x0 0
```

# Chapter 9. EVENT NOTIFICATIONS

As the application is making forward progress, CUDA-GDB notifies the users about kernel events and context events. Within CUDA-GDB, *kernel* refers to the device code that executes on the GPU, while *context* refers to the virtual address space on the GPU for the kernel. You can enable output of CUDA context and kernel events to review the flow of the active contexts and kernels. By default, only context event messages are displayed.

## 9.1. Context Events

Any time a CUDA context is created, pushed, popped, or destroyed by the application, CUDA-GDB will display a notification message. The message includes the context id and the device id to which the context belongs.

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

The context event notification policy is controlled with the **`context_events`** option.

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

  CUDA-GDB does not display the context event notification messages.
- `(cuda-gdb) set cuda context_events on`

  CUDA-GDB displays the context event notification messages (default).

## 9.2. Kernel Events

Any time CUDA-GDB is made aware of the launch or the termination of a CUDA kernel, a notification message can be displayed. The message includes the kernel id, the kernel name, and the device to which the kernel belongs.

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

The kernel event notification policy is controlled with **kernel_events** and **kernel_events_depth** options.

- `(cuda-gdb) set cuda kernel_events none`

  Possible options are:

  **none**
  no kernel, application or system (default)

  **application**
  kernel launched by the user application

  **system**
  any kernel launched by the driver, such as memset

  **all**
  any kernel, application and system

- `(cuda-gdb) set cuda kernel_events_depth 0`

  Controls the maximum depth of the kernels after which no kernel event notifications will be displayed. A value of zero means that there is no maximum and that all the kernel notifications are displayed. A value of one means that the debugger will display kernel event notifications only for kernels launched from the CPU (default).

In addition to displaying kernel events, the underlying policy used to notify the debugger about kernel launches can be changed. By default, kernel launches cause events that CUDA-GDB will process. If the application launches a large number of kernels, it is preferable to defer sending kernel launch notifications until the time the debugger stops the application. At this time only the kernel launch notifications for kernels that are valid on the stopped devices will be displayed. In this mode, the debugging session will run a lot faster.

The deferral of such notifications can be controlled with the **defer_kernel_launch_notifications** option.

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

  CUDA_GDB receives events on kernel launches (default).

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

  CUDA-GDB defers receiving information about kernel launches

**set cuda defer_kernel_launch_notifications** option is deprecated and has no effect any more.

# Chapter 10. AUTOMATIC ERROR CHECKING

## 10.1. Checking API Errors

CUDA-GDB can automatically check the return code of any driver API or runtime API call. If the return code indicates an error, the debugger will stop or warn the user.

The behavior is controlled with the **`set cuda api_failures`** option. Three modes are supported:

- **`hide`** will not report any error of any kind
- **`ignore`** will emit a warning but continue the execution of the application (default)
- **`stop`** will emit an error and stop the application

The success return code and other non-error return codes are ignored. For the driver API, those are: **`CUDA_SUCCESS`** and **`CUDA_ERROR_NOT_READY`**. For the runtime API, they are **`cudaSuccess`** and **`cudaErrorNotReady`**.

## 10.2. 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.

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

Table 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. |
| `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. |

| Exception Code | Precision of the Error | Scope of the Error | Description |
|---|---|---|---|
| `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 'set cuda memcheck' option. |
| `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. |
| `CUDA_EXCEPTION_12 : "Warp Assert"` | Precise | Per warp | This occurs when any thread in the warp hits a device side assertion. |
| `CUDA_EXCEPTION_13: "Lane Syscall Error"` | Precise (Requires memcheck on) | Per lane/ thread error | This occurs when a thread corrupts the heap by invoking free with an invalid address (for example, trying to free the same memory region twice) |
| `CUDA_EXCEPTION_14 : "Warp Illegal Address"` | Not precise | Per warp | This occurs when a thread accesses an illegal(out of bounds) global/local/ shared address. For increased precision, use the 'set cuda memcheck' option. |
| `CUDA_EXCEPTION_15 : "Invalid Managed Memory Access"` | Precise | Per host thread | This occurs when a host thread attempts to access managed memory currently used by the GPU. |

# 10.3. set cuda memcheck

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 misaligned global memory accesses will be reported.

When CUDA memcheck is enabled, all the kernel launches are made blocking, as if the environment variable **CUDA_LAUNCH_BLOCKING** was set to 1. The host thread launching a kernel will therefore wait until the kernel has completed before proceeding. This may change the behavior of your application.

You can also run the 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.

By default, CUDA-GDB will report any memory error. See GPU Error Reporting for a list of the memory errors. To increase the number of memory errors being reported and to increase the precision of the memory errors, CUDA memcheck must be turned on.

## 10.4. Autostep

### Description

Autostep is a command to increase the precision of CUDA exceptions to the exact lane and instruction, when they would not have been otherwise.

Under normal execution, an exception may be reported several instructions after the exception occurred, or the exact thread where an exception occurred may not be known unless the exception is a lane error. However, the precise origin of the exception can be determined if the program is being single-stepped when the exception occurs. Single-stepping manually is a slow and tedious process; stepping takes much longer than normal execution and the user has to single-step each warp individually.

Autostep aides the user by allowing them to specify sections of code where they suspect an exception could occur, and these sections are automatically and transparently single-stepped the program is running. The rest of the program is executed normally to minimize the slow-down caused by single-stepping. The precise origin of an exception will be reported if the exception occurs within these sections. Thus the exact instruction and thread where an exception occurred can be found quickly and with much less effort by using autostep.

### Usage

```
autostep [LOCATION]
autostep [LOCATION] for LENGTH [lines|instructions]
```

- **LOCATION** may be anything that you use to specify the location of a breakpoint, such as a line number, function name, or an instruction address preceded by an asterisk. If no **LOCATION** is specified, then the current instruction address is used.

- **LENGTH** specifies the size of the autostep window in number of lines or instructions (*lines* and *instructions* can be shortened, e.g., *l* or *i*). If the length type is not specified, then *lines* is the default. If the **for** clause is omitted, then the default is 1 line.
- **astep** can be used as an alias for the **autostep** command.
- Calls to functions made during an autostep will be stepped over.
- In case of divergence, the length of the autostep window is determined by the number of lines or instructions the first active lane in each warp executes.

  Divergent lanes are also single stepped, but the instructions they execute do not count towards the length of the autostep window.
- If a breakpoint occurs while inside an autostep window, the warp where the breakpoint was hit will not continue autostepping when the program is resumed. However, other warps may continue autostepping.
- Overlapping autosteps are not supported.

If an autostep is encountered while another autostep is being executed, then the second autostep is ignored.

If an autostep is set before the location of a memory error and no memory error is hit, then it is possible that the chosen window is too small. This may be caused by the presence of function calls between the address of the autostep location and the instruction that triggers the memory error. In that situation, either increase the size of the window to make sure that the faulty instruction is included, or move to the autostep location to an instruction that will be executed closer in time to the faulty instruction.

**Autostep requires Fermi GPUs or above.**

### Related Commands

Autosteps and breakpoints share the same numbering so most commands that work with breakpoints will also work with autosteps.

**info autosteps** shows all breakpoints and autosteps. It is similar to **info breakpoints**.

```
(cuda-gdb) info autosteps
Num  Type      Disp Enb Address            What
1    autostep  keep y   0x0000000000401234 in merge at sort.cu:30 for 49
 instructions
3    autostep  keep y   0x0000000000489913 in bubble at sort.cu:94 for 11 lines
```

**disable autosteps** disables an autostep. It is equivalent to **disable breakpoints n**.

**delete autosteps n** deletes an autostep. It is quivalent to **delete breakpoints n**.

**ignore n i** tells the debugger to not single-step the next *i* times the debugger enters the window for autostep *n*. This command already exists for breakpoints.

# Chapter 11. WALK-THROUGH EXAMPLES

The chapter contains two CUDA-GDB walk-through examples:

- Example: bitreverse
- Example: autostep
- Example: MPI CUDA Application

## 11.1. Example: bitreverse

This section 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.

### 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 }
```

## 11.1.1. 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 Debug Compilation
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 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, let's continue until the device kernel is launched.

```
(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. The debugger 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) backtrace
#0   bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x110000) at bitreverse.cu:9
```

**7.** Corroborate this information by printing the block and thread indexes:

```
(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)
```

## 11.2. Example: autostep

This section shows how to use the autostep command and demonstrates how it helps increase the precision of memory error reporting.

### Source Code

```
1  #define NUM_BLOCKS 8
2  #define THREADS_PER_BLOCK 64
3
4  __global__ void example(int **data) {
5    int value1, value2, value3, value4, value5;
6    int idx1, idx2, idx3;
7
8    idx1 = blockIdx.x * blockDim.x;
9    idx2 = threadIdx.x;
10   idx3 = idx1 + idx2;
11   value1 = *(data[idx1]);
12   value2 = *(data[idx2]);
13   value3 = value1 + value2;
14   value4 = value1 * value2;
15   value5 = value3 + value4;
16   *(data[idx3]) = value5;
17   *(data[idx1]) = value3;
18   *(data[idx2]) = value4;
19   idx1 = idx2 = idx3 = 0;
20 }
21
22 int main(int argc, char *argv[]) {
23   int *host_data[NUM_BLOCKS * THREADS_PER_BLOCK];
24   int **dev_data;
25   const int zero = 0;
26
27   /* Allocate an integer for each thread in each block */
28   for (int block = 0; block &lt; NUM_BLOCKS; block++) {
29     for (int thread = 0; thread &lt; THREADS_PER_BLOCK; thread++) {
30       int idx = thread + block * THREADS_PER_BLOCK;
31       cudaMalloc(&amp;host_data[idx], sizeof(int));
32       cudaMemcpy(host_data[idx], &amp;zero, sizeof(int),
33                  cudaMemcpyHostToDevice);
34     }
35   }
36
37   /* This inserts an error into block 3, thread 39*/
38   host_data[3*THREADS_PER_BLOCK  + 39] = NULL;
39
40   /* Copy the array of pointers to the device */
41   cudaMalloc((void**)&amp;dev_data,  sizeof(host_data));
42   cudaMemcpy(dev_data, host_data, sizeof(host_data), cudaMemcpyHostToDevice);
43
44   /* Execute example */
45   example &lt;&lt;&lt; NUM_BLOCKS, THREADS_PER_BLOCK >>> (dev_data);
46   cudaThreadSynchronize();
47 }
```

In this small example, we have an array of pointers to integers, and we want to do some operations on the integers. Suppose, however, that one of the pointers is NULL as shown in line 38. This will cause **`CUDA_EXCEPTION_10 "Device Illegal Address"`** to be thrown when we try to access the integer that corresponds with block 3, thread 39. This exception should occur at line 16 when we try to write to that value.

## 11.2.1. Debugging with Autosteps

1. Compile the example and start CUDA-GDB as normal. We begin by running the program:

```
(cuda-gdb) run
Starting program: /home/jitud/cudagdb_test/autostep_ex/example
[Thread debugging using libthread_db enabled] [New Thread 0x7ffff5688700 (LWP
 9083)]
[Context Create of context 0x617270 on Device 0]
[Launch of CUDA Kernel 0 (example<<<(8,1,1),(64,1,1)>>>) on Device 0]

Program received signal CUDA_EXCEPTION_10, Device Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (0,0,0),
 device 0, sm 1, warp 0, lane 0]
0x0000000000796f60 in example (data=0x200300000) at example.cu:17
17        *(data[idx1]) = value3;
```

As expected, we received a **CUDA_EXCEPTION_10**. However, the reported thread is block 1, thread 0 and the line is 17. Since **CUDA_EXCEPTION_10** is a Global error, there is no thread information that is reported, so we would manually have to inspect all 512 threads.

2. Set **autosteps**. To get more accurate information, we reason that since **CUDA_EXCEPTION_10** is a memory access error, it must occur on code that accesses memory. This happens on lines 11, 12, 16, 17, and 18, so we set two autostep windows for those areas:

```
(cuda-gdb) autostep 11 for 2 lines
Breakpoint 1 at 0x796d18: file example.cu, line 11.
Created autostep of length 2 lines
(cuda-gdb) autostep 16 for 3 lines
Breakpoint 2 at 0x796e90: file example.cu, line 16.
Created autostep of length 3 lines
```

3. Finally, we run the program again with these autosteps:

```
(cuda-gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
[Termination of CUDA Kernel 0 (example<<<(8,1,1),(64,1,1)>>>) on Device 0]
Starting program: /home/jitud/cudagdb_test/autostep_ex/example
[Thread debugging using libthread_db enabled]
[New Thread 0x7ffff5688700 (LWP 9089)]
[Context Create of context 0x617270 on Device 0]
[Launch of CUDA Kernel 1 (example<<<(8,1,1),(64,1,1)>>>) on Device 0]
[Switching focus to CUDA kernel 1, grid 1, block (0,0,0), thread (0,0,0),
device 0, sm 0, warp 0, lane 0]

Program received signal CUDA_EXCEPTION_10, Device Illegal Address.
[Current focus set to CUDA kernel 1, grid 1, block (3,0,0), thread
(32,0,0), device 0, sm 1, warp 3, lane 0]
Autostep precisely caught exception at example.cu:16 (0x796e90)
```

This time we correctly caught the exception at line 16. Even though **CUDA_EXCEPTION_10** is a global error, we have now narrowed it down to a warp error, so we now know that the thread that threw the exception must have been in the same warp as block 3, thread 32.

In this example, we have narrowed down the scope of the error from 512 threads down to 32 threads just by setting two **autosteps** and re-running the program.

## 11.3. Example: MPI CUDA Application

For doing large MPI CUDA application debugging, NVIDIA recommends using parallel debuggers supplied by our partners Allinea and Totalview. Both make excellent parallel debuggers with extended support for CUDA. However, for debugging smaller applications, or for debugging just a few processes in a large application, CUDA-GDB can easily be used.

If the cluster nodes have xterm support, then it is quite easy to use CUDA-GDB. Just launch CUDA-GDB in the same way you would have launched gdb.

```
$ mpirun -np 4 -host nv1,nv2 xterm -e cuda-gdb a.out
```

You may have to export the DISPLAY variable to make sure that the xterm finds its way back to your display. For example, with Open MPI you would do something like this.

```
$ mpirun -np 4 -host nv1,nv2 -x DISPLAY=host.nvidia.com:0 xterm -e cuda-gdb
 a.out
```

Different MPI implementations have different ways of exporting environment variables to the cluster nodes so check your documentation.

In the case where you cannot get xterm support, you can insert a spin loop inside your program. This works in just the same way as when using gdb on a host only program. Somewhere near the start of your program, add a code snippet like the following.

```
{
    int i = 0;
    char host[256];
    printf("PID %d on node %s is ready for attach\n",
            getpid(), host);
    fflush(stdout);
    while (0 == i) {
        sleep(5);
    }
}
```

Then recompile and run the program. After it starts, ssh to the nodes of interest and attach to the process. Set the variable i to 1 to break out of the loop.

```
$ mpirun -np 2 -host nv1,nv2 a.out
PID 20060 on node nv1 is ready for attach
PID 5488 on node nv2 is ready for attach
```

```
[nv1]$ cuda-gdb --pid 5488
```

```
[nv2]$ cuda-gdb --pid 20060
```

For larger applications in the case where you may just want to attach to a few of the processes, you can conditionalize the spin loop based on the rank. Most MPIs set an environment variable that is the rank of the process. For Open MPI it is

OMPI_COMM_WORLD_RANK and for MVAPICH it is MV2_COMM_WORLD_RANK. Assuming you want to attach to rank 42, you could add a spin loop like this.

```
{
    char *stoprank;
    stoprank = getenv("OMPI_COMM_WORLD_RANK");
    if (42 == atoi(stoprank) {
        int i = 0;
        char hostname[256];
        printf("PID %d on %s ready for attach\n",
               getpid(), hostname);
        fflush(stdout);
        while (0 == i) {
            sleep(5);
        }
    }
}
```

Note that by default CUDA-GDB allows debugging a single process per node. The workaround described in Multiple Debuggers does not work with MPI applications. If CUDA_VISIBLE_DEVICES is set, it may cause problems with the GPU selection logic in the MPI application. It may also prevent CUDA IPC working between GPUs on a node. In order to start multiple CUDA-GDB sessions to debug individual MPI processes on the same node, use the **`--cuda-use-lockfile=0`** option when starting CUDA-GDB, as described in --cuda-use-lockfile. Each MPI process must guarantee it targets a unique GPU for this to work properly.

# Chapter 12. ADVANCED SETTINGS

## 12.1. --cuda-use-lockfile

When debugging an application, CUDA-GDB will suspend all the visible CUDA-capable devices. To avoid any resource conflict, only one CUDA-GDB session is allowed at a time. To enforce this restriction, CUDA-GDB uses a locking mechanism, implemented with a lock file. That lock file prevents 2 CUDA-GDB processes from running simultaneously.

However, if the user desires to debug two applications simultaneously through two separate CUDA-GDB sessions, the following solutions exist:

- Use the **CUDA_VISIBLE_DEVICES** environment variable to target unique GPUs for each CUDA-GDB session. This is described in more detail in Multiple Debuggers.
- Lift the lockfile restriction by using the **--cuda-use-lockfile** command-line option.

  ```
  $ cuda-gdb --cuda-use-lockfile=0 my_app
  ```

  This option is the recommended solution when debugging multiple ranks of an MPI application that uses separate GPUs for each rank. It is also required when using software preemption (**set cuda software_preemption on**) to debug multiple CUDA applications context-switching on the same GPU.

## 12.2. set cuda break_on_launch

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:

**none**
no kernel, application or system (default)
**application**
kernel launched by the user application
**system**
any kernel launched by the driver, such as memset
**all**
any kernel, application and system

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.

# 12.3. set cuda gpu_busy_check

As a safeguard mechanism, cuda-gdb will detect if a visible device is also used for display and return an error. A device used for display cannot be used for compute while debugging. To hide the device, use the **CUDA_VISIBLE_DEVICES** environment variable. This option is only valid on Mac OS X.

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

  The safeguard mechanism is turned off and the user is responsible for guaranteeing the device can safely be used.
- `(cuda-gdb) set cuda gpu_busy_check on`

  The debugger will return an error if at least one visible device is already in use for display. It is the default setting.

# 12.4. set cuda launch_blocking

When enabled, the kernel launches are synchronous as if the environment variable **CUDA_LAUNCH_BLOCKING** had been set to 1. Once blocking, the launches are effectively serialized and may be easier to debug.

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

  The kernel launches are launched synchronously or asynchronously as dictacted by the application. This is the default.
- `(cuda-gdb) set cuda launch_blocking on`

  The kernel launches are synchronous. If the application has already started, the change will only take affect after the current session has terminated.

# 12.5. set cuda notify

Any time a CUDA event occurs, the debugger needs to be notified. The notification takes place in the form of a signal being sent to a host thread. The host thread to receive that special signal is determined with the **`set cuda notify`** option.

- `(cuda-gdb) set cuda notify youngest`

  The host thread with the smallest thread id will receive the notification signal (default).
- `(cuda-gdb) set cuda notify random`

  An arbitrary host thread will receive the notification signal.

# 12.6. set cuda ptx_cache

Before accessing the value of a variable, the debugger checks whether the variable is live or not at the current PC. On CUDA devices, the variables may not be live all the time and will be reported as "Optimized Out".

CUDA-GDB offers an option to circumvent this limitation by caching the value of the variable at the PTX register level. Each source variable is compiled into a PTX register, which is later mapped to one or more hardware registers. Using the debug information emitted by the compiler, the debugger may be able cache the value of a PTX register based on the latest hardware register it was mapped to at an earlier time.

This optimization is always correct. When enabled, the cached value will be displayed as the normal value read from an actual hardware register and indicated with the **`(cached)`** prefix. The optimization will only kick in while single-stepping the code.

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

  The debugger only read the value of live variables.
- `(cuda-gdb) set cuda ptx_cache on`

  The debugger will use the cached value when possible. This setting is the default and is always safe.

# 12.7. set cuda single_stepping_optimizations

Single-stepping can take a lot of time. When enabled, this option tells the debugger to use safe tricks to accelerate single-stepping.

- `(cuda-gdb) set cuda single-stepping-optimizations off`

  The debugger will not try to accelerate single-stepping. This is the unique and default behavior in the 5.5 release and earlier.

- `(cuda-gdb) set cuda single-stepping-optimizations on`

  The debugger will use safe techniques to accelerate single-stepping. This is the default starting with the 6.0 release.

## 12.8. set cuda thread_selection

When the debugger must choose an active thread to focus on, the decision is guided by a heuristics. The **`set cuda thread_selection`** guides those heuristics.

- `(cuda-gdb) set cuda thread_selection logical`

  The thread with the lowest blockIdx/threadIdx coordinates is selected.
- `(cuda-gdb) set cuda thread_selection physical`

  The thread with the lowest dev/sm/warp/lane coordinates is selected.

## 12.9. set cuda value_extrapolation

Before accessing the value of a variable, the debugger checks whether the variable is live or not at the current PC. On CUDA devices, the variables may not be live all the time and will be reported as "Optimized Out".

CUDA-GDB offers an option to opportunistically circumvent this limitation by extrapolating the value of a variable when the debugger would otherwise mark it as optimized out. The extrapolation is not guaranteed to be accurate and must be used carefully. If the register that was used to store the value of a variable has been reused since the last time the variable was seen as live, then the reported value will be wrong. Therefore, any value printed using the option will be marked as **`"(possibly)"`**.

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

  The debugger only read the value of live variables. This setting is the default and is always safe.
- `(cuda-gdb) set cuda value_extrapolation on`

  The debugger will attempt to extrapolate the value of variables beyound their respecitve live ranges. This setting may report erroneous values.

# Appendix A. SUPPORTED PLATFORMS

### Host Platform Requirements

CUDA-GDB is supported on all the platforms supported by the CUDA toolkit with which it is shipped. See the CUDA Toolkit release notes for more information.

### GPU Requirements

Debugging is supported on all CUDA-capable GPUs with a compute capability of 1.1 or later.

# Appendix B. KNOWN ISSUES

The following are known issues with the current release.

- Setting the **`cuda memcheck`** option ON will make all the launches blocking.
- On GPUs with **`sm_type`** lower than **`sm_20`** it is not possible to step over a subroutine in the device code.
- Requesting to read or write GPU memory may be unsuccessful if the size is larger than 100MB on Tesla GPUs and larger than 32MB on Fermi GPUs.
- 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.
- To debug CUDA application that uses OpenGL, X server may need to be launched in non-interactive mode. See CUDA/OpenGL Interop Applications on Linux for details.
- Pretty-printing is not supported.
- When remotely debugging 32-bit applications on a 64-bit server, gdbserver must be 32-bit.
- Attaching to a CUDA application with Software Preemption enabled in cuda-gdb is not supported.
- Attaching to CUDA application running in MPS client mode is not supported.

- Attaching to the MPS server process (nvidia-cuda-mps-server) using cuda-gdb, or starting the MPS server with cuda-gdb is not supported.
- If a CUDA application is started in the MPS client mode with cuda-gdb, the MPS client will wait until all other MPS clients have terminated, and will then run as non-MPS application.
- On Android and on other systems-on-chip with compute-capable GPU, debugger will always report managed memory as resident on the device.
- Attaching to CUDA application on Android is not supported.
- Debugging APK binaries is not supported.
- Significant performance degradation when debugger steps over inlined routines.

  Because inlined code blocks may have multiple exit points, under the hood, the debugger steps every single instruction until an exit point is reached, which incurs considerable cost for large routines. The following actions are recommended to avoid this problem:

  - Avoid using **`__forceinline__`** when declaring a function. (For code is compiled with debug information, only routines declared with the **`__forceinline__`** keyword are actually inlined)
  - Use the **`until <line#>`** command to step over inlined subroutines.

## Notice



## Trademarks



## Copyright