CUDA-GDB

Release 12.6

NVIDIA Corporation

Jul 30, 2024

Contents

CUDA-GDB

The user manual for CUDA-GDB, the NVIDIA tool for debugging CUDA applications on Linux and QNX systems.

This document introduces CUDA-GDB, the NVIDIA® CUDA® debugger for Linux and QNX targets.

Chapter 1. What is CUDA-GDB?

CUDA-GDB is the NVIDIA tool for debugging CUDA applications running on Linux and QNX. CUDA-GDB is an extension to 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.

Chapter 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.

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_75` or `sm_80`, but also supports debugging kernels compiled at runtime, referred to as just-in-time compilation, or JIT compilation for short.

Chapter 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 4. Release Notes

12.6 Release

Features

- New `set/show cuda step_divergent_lanes` command to control automatic stepping of divergent threads when the focused CUDA thread is no longer active. On by default to match existing behavior.
- Improved performance of opening GPU core files containing many cubins.
- Allow users to ctrl-c and interrupt loading of GPU core files.
- Emit warning when starting to step divergent threads and `step_divergent_lanes` is on.
- Emit thread switch notification when switching from CUDA thread back to host thread focus.

Fixed Issues

- Fixed issue with CUDA thread selection when switching to nearest neighbor thread when current thread focus goes out of scope.
- Fixed issues with printing CUDA registers when debugging GPU core files.
- Enhanced error handling when encountering cubins with corrupted extended line info.
- Fixed errors when receiving out-of-order packets for QNX cuda-gdbserver.

12.5 Release

Updated GDB version

- Moved from GDB 13.1 to 13.2. See GDB 13.2 changes

Support removal notice

- Support for the macOS host client of CUDA-GDB has been removed.
- Support for Android has been removed.
- Support for Python 3.6 and 3.7 has been removed.

Features

- Multi build feature that supports native Python and TUI mode across all supported platforms. The cuda-gdb program is now a wrapper script that calls the appropriate cuda-gdb binary. If no supported Python or libncurses is detected, the wrapper will fallback to a cuda-gdb binary with Python and TUI support disabled.

- Added support for TUI mode.
- Added support for Python 3.10, 3.11, and 3.12.
- Added support for detecting and printing exceptions encountered in exited warps. This can occur when debugging an application with optimizations enabled.
- Added new gdb/mi command equivalents for info cuda managed and info cuda line.

Fixed Issues

- Fixed issue with printing reference parameter arguments to CUDA functions.
- Fixed issues resulting in crashes/errors when reading/writing from/to CUDA generic memory.
- Fixed issue where break_on_launch breakpoints were missed for back to back launches of the same kernel.
- Fixed issue with incorrectly reporting breakpoint hit events as SIGTRAP when breakpoint is hit in divergent thread.
- Fixed crash on QNX when cuda-gdbserver packets arrive out-of-order.
- Better error handling when encountering an error when reading CUDA disassembly.
- Better exit handling when resuming execution from a fatal CUDA exception.

12.4 Release

Updated GDB version

- Moved from GDB 12.1 to 13.1. See GDB 13.1 changes

Android deprecation notice

- Support for Android is deprecated. It will be dropped in an upcoming release.

Python 3.6 and 3.7 deprecation notice

- Support for end-of-life Python 3.6 and 3.7 versions is deprecated. It will be dropped in an upcoming release.

Features

- Performance enhancement which reduces the number of overall CUDA Debugger API calls.
- Performance enhancement when loading large cubins with device functions using a large number of GPU registers.
- Performance enhancement when single stepping over warp wide barriers.
- Added support for printing values contained within constant banks from GPU core dumps.

Fixed Issues

- Prevented shell expansion on cloned function names when disassembling.
- Fixed crash when setting a conditional breakpoint on an unknown symbol name.

- Fixed issue with setting a watchpoint on a global pointer.
- Fixed assertion in switch_to_thread_1 during inferior teardown.
- Fixed attach failures encountered with newer Intel processors.
- Refactored the libpython layer to avoid unnecessary gdb code changes.

12.3 Release

macOS host client deprecation notice

- Support for the macOS host client of CUDA-GDB is deprecated. It will be dropped in an upcoming release.

Features

- Added support for printing values contained within constant banks. New `$_cuda_const_bank(bank, offset)` convenience function to obtain address of offset in constant bank. See Const Banks.
- Performance enhancements added which reduce overhead when running applications with many CUDA threads.
- Added support for CUDA function pointers.

Fixed Issues

- Fixed issue when detaching from attached process that can result in a crash.
- Fixed thread ordering issues present with several `info cuda` commands.
- Added support for opening of GPU core dumps when no valid warps are present on the device.
- Added missing DWARF operators used by OptiX.
- Fixed issue with parsing CUDA Fortran pointer types.
- Fixed issue where CUDA Cluster coordinates were being displayed when no CUDA Cluster was present.

12.2 Release

Features

- Enabled printing of extended error messages when a CUDA Debugger API error is encountered.
- Enabled support for debugging with Confidential Compute mode with devtools mode. See *Confidential Computing Deployment Guide <https://docs.nvidia.com/confidential-computing-deployment-guide.pdf>* for more details on how to enable the mode.

Fixed Issues

- Fixed “??” appearing in backtrace in OptiX applications.
- Host shadow breakpoints are now handled correctly with CUDA Lazy Loading enabled.
- Fixed name mangling issue when debugging LLVM generated cubins.
- CUDA Cluster coordinates are now displayed correctly.

- Fixed issue with attaching to an application using CUDA Lazy Loading when debugging remotely with cuda-gdbserver.

12.1 Release

CUDA Driver API added for controlling core dump behavior

- CTK 12.1 and the r530 driver adds new APIs that allow developers to enable/configure core dump settings programmatically inside their application instead of using environment variables. See the CUDA Driver API manual for more information.

Features

- Performance improvements for applications using CUDA Lazy Loading.
- Added support for ELF cubins with a large number of sections (more than 32767).
- Added `break_on_launch` support for CUDA Graphs.

Fixed Issues

- Removed unsupported set/show gpu_busy_check command.
- On QNX fixed an issue where `info threads` incorrectly reported dead host threads.
- Performance fixes for stepping/next over inline function calls.
- Performance fixes when using the `info cuda managed` command.
- Fixed issue when using `set follow-fork-mode child`.
- Fixed issue when parsing DWARF for self referential structures.

12.0 Release

Updated GDB version

- Moved from GDB 10.2 to 12.1. See GDB 12.1 changes

Texture and surface reference support removed

- CTK 12.0 removed support for the Texture and Surface Reference API. Support for printing texture and surface references has been removed.

CUDA Memory Checker integration removed

- cuda-memcheck has been deprecated in CUDA 11.x and replaced by Compute Sanitizer. The new memory checking workflow is to use Compute Sanitizer from the CLI. This will support coredumps when issues are detected which can then be opened and inspected with CUDA-GDB, similar to other coredumps. Support for cuda-memcheck has been removed with the CUDA 12.0 release.

Debugging of applications using CUDA Dynamic Parallelism

- Support for debugging applications using CUDA Dynamic Parallelism with the classic debugger backend or on Maxwell GPUs has been removed by default for applications compiled with the CTK 12.0 or newer. Debugging can be accomplished in these situations by recompiling the application while passing the -DCUDA_FORCE_CDP1_IF_SUPPORTED flag.

Features

- Moved from base gdb/10.2 to gdb/12.1.
- Added initial support for Thread Block Clusters.
- Changed the default behavior of `--cuda-use-lockfile` to `0`. Lockfiles are no longer created by default.

Fixed Issues

- Addressed a hang that could be encountered when stepping through device system calls.
- Fixed an overflow issue with displaying active warp masks in `info cuda` commands.
- Changed internal CUDA Dynamic Parallelsim detection breakpoint to be set only when `break_on_launch` is enabled.
- Removed unsupported `gpu_busy_check setting`.

11.8 Release

Features

- Uses the new Unified Debugger (UD) debugging backend by default.
- Added support for debugging applications using CUDA Lazy Loading.
- Debugger is now enabled on Windows Subsystem for Linux (WSL).
- Added basic type support for printing FP8 values (E4M3 and E5M2).

Notes

- By default, CUDA-GDB will use the new Unified Debugger (UD) backend. This change is transparent to most users using Pascal or newer cards. For Maxwell debugging, or to force the old classic debugging backend, set `CUDBG_USE_LEGACY_DEBUGGER` to 1 in your environment.
- WSL is not supported on GH100 platforms with this release.

11.7 Release

Features

- Major break_on_launch performance enhancements to use new KERNEL_READY notification mechanism instead of setting manual breakpoints.
- Refactored info cuda command output to be more condensed. Omitted printing of inactive messages.
- Added new `--disable-python` commandline option to disable Python interpreter dlopen.

Fixed Issues

- Fixed follow-fork child to avoid hanging behavior when both parent and child processes use CUDA.
- Added a missing dlsym of a libpython function that was causing errors with some versions of libpython.

11.6 Release

Updated GDB version

- Moved from GDB 10.1 to 10.2. See GDB 10.2 changes

Features

- Added errorpc instruction prefix to the disassembly view. If an error PC is set, prefix the instruction with *>.

Fixed Issues

- Fixed lineinfo frames to properly display the source filename.
- Fixed writing to gpu global memory that was allocated from the host.
- Fixed bug that was preventing reading host variables during certain situations.
- Fixed cuda-gdbserver init check that prevented QNX from starting.

11.5 Release

Python 3 support on Jetson and Drive Tegra devices

- Support for Python 2 has been removed. CUDA-GDB now supports Python 3 on Jetson and Drive Tegra devices.

Fixed Issues

- Added robust version checks when dynamic loading the libpython3 library. The loaded libpython3 will match the version of the python3 runtime in PATH.
- Added support for checking PEP-3149 flag names when loading libpython3 libraries.
- Added support for dynamic loading of Python 3.9.
- Fixed overriding PYTHONPATH on certain RHEL distributions.

11.4 Update 1 Release

Known Issues with Fedora 34

- CUDA-GDB has known issues with debugging on Fedora 34 and may not be reliable.

Fixed Issues

- Enabled python integration for ppc64le and aarch64 SBSA.
- Fixed a performance regression when debugging CUDA apps.
- Fixed an intermitent hang with remote debugging via cuda-gdbserver.
- Fixed bug with set cuda api_failures stop not triggering breakpoints on failure.
- Changed python behavior to dlopen libpython libraries that match the version of the python3 interpreter in PATH.
- OpenMP Fortran: Fixed a crash when setting breakpoints inside an OpenMP parallel region.
- OpenMP: Better support for printing local variables within a parallel region.
- Fortran: Added updated support for printing assumed shape arrays and array slices.

- Fixed selecting between host and device thread focus in cudacore debugging.
- Various fixes for QNX remote debugging.

11.4 Release

Updated GDB version

- Moved from GDB 8.3 to 10 (based on GDB 10.1). See GDB 10.1 changes

Python 3 support

- Support for Python 2 has been removed. CUDA-GDB now supports Python 3.

GDB TUI mode disabled

- Support for GDB TUI mode has been disabled. This avoids cross platform dependency mismatches for OSes that lack ncurses-5.5 support.

Kepler deprecation notice

- Support for Kepler devices (sm_35 and sm_37) is deprecated. Kepler support will be dropped in an upcoming release.

Coredump support

- Added support for writing coredumps to named pipe using `CUDA_COREDUMP_FILE`.

Fixed Issues

- Added support for displaying SIGTRAP exception in coredumps.
- Disabled ability to enable scheduler-locking when debugging CUDA targets.
- Fixed cuda_register_name and cuda_special_register_name to avoid returning old cached result on error.
- Fixed intermitent race condition when creating the CUDA temporary directory.
- Various fixes for QNX remote debugging.

11.3 Release

Python 2 deprecation notice

- Support for Python 2 is being deprecated. CUDA-GDB will move to build with Python 3 support in an upcoming release.

Fixed Issues

- Improvements to late attach for remote debugging.

11.2 Update 1 Release

GDB TUI deprecation notice

- Support for GDB TUI mode is being deprecated. This will avoid cross platform dependency mismatches for OSes that lack ncurses-5.5 support. GDB TUI mode will be disabled in an upcoming release.

Fixed Issues

- Fixed printing of strings in the global GPU memory while running CPU code.
- Fixed a bug with extended debug_line handling.
- Fixed truncation with builtin gdb variables such as `gridDim`.
- Fixed a segfault during startup for DWARF dies missing names.
- Fixed a segfault when a CUDA kernel calls `assert`.
- Fixed a bug that prevented debugging cubins > 2GB.
- Added minor usability enhancements for cubins compiled with `--lineinfo`.
- Fixed a segfault cause by a pretty printer when using CUDA-GDB within CLion.

11.1 Release

Updated GDB version

- Moved from GDB 8.2 to 8.3 (based on gdb 8.3.1). See gdb 8.3.1 changes

Support for SM 8.6

- CUDA-GDB now supports Devices with Compute Capability 8.6.

Updated DWARF parser

- Old binaries might need to be recompiled in order to ensure CUDA-specific DWARF info are up to date.

Fixed Issues

- Fixed an intermittent deadlock when attaching to a running CUDA process.
- Fixed a bug when inspecting the value of half registers.

11.0 Release

Updated GDB version

- CUDA-GDB has been upgraded from GDB/7.12 to GDB/8.2.

Support for SM8.0

- CUDA-GDB now supports Devices with Compute Capability 8.0.

Support for Bfloat16

- Support for Bfloat16 (__nv_bfloat16) types have been added.

MIG support

- CUDA-GDB supports MIG. There can be a separate debugger session on each MIG instance. Refer to Multiple Debuggers in case multiple debuggers are needed.

Mac support

- Debugging on macOS is no longer supported. However, macOS can still be used as the host system (where CUDA-GDB runs under macOS, using cuda-gdbserver to debug a remote target). The download for the macOS version of CUDA-GDB can be found at the following location: Download Here

10.1 Release

Enhanced debugging with only linenumber information

- Several enhancements were made to CUDA-GDB support for debugging programs compiled with `-lineinfo` but not with `-G`. This is intended primarily for debugging programs built with OptiX/RTCore. See also Compilation With Linenumber Information

10.0 Release

Turing Uniform Register Support

- Support added for examining and modifying uniform registers on Turing GPUs.

9.2 Release

User induced core dump support

- For the devices that support compute preemption, user induced core dump support is added. New environment variable: `CUDA_ENABLE_USER_TRIGGERED_COREDUMP` can be used to enable this feature.

9.1 Release

Volta-MPS core dump support

- GPU core dump generation is supported on Volta-MPS.

Lightweight GPU core dump support

- CUDA-GDB supports reading lightweight GPU core dump files. New environment variable: `CUDA_ENABLE_LIGHTWEIGHT_COREDUMP` can be used to enable this feature.

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 5. Getting Started

The CUDA toolkit can be installed by following instructions in the Quick Start Guide.

Further steps should be taken to set up the debugger environment, build the application, and run the debugger.

5.1. Setting Up the Debugger Environment

5.1.1. 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.

Note: 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.

Note: The value of `$TMPDIR` must be the same in the environment of the application and CUDA-GDB. If they do not match, CUDA-GDB will fail to attach onto the application process.

Note: 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.

5.1.2. Using the CUDA-GDB debugger on Jetson and Drive Tegra devices

By default, on Jetson and Drive Tegra devices, GPU debugging is supported only if `cuda-gdb` and `cuda-gdbserver` are launched by a user who is a member of the **debug** group.

To add the current user to the **debug** group run this command:

```
sudo usermod -a -G debug $USER
```

5.2. Compiling the Application

5.2.1. 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 for ease of debugging 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.

5.2.2. Compilation With Linenumber Information

Several enhancements were made to cuda-gdb's support for debugging programs compiled with `-lineinfo` but not with `-G`. This is intended primarily for debugging programs built with OptiX/RTCore.

Note that `-lineinfo` can be used when trying to debug optimized code. In this case, debugger stepping and breakpoint behavior may appear somewhat erratic.

- The PC may jump forward and backward unexpectedly while stepping.
- The user may step into code that has no linenumber information, leading to an inability to determine which source-file/linenumber the code at the PC belongs to.
- Breakpoints may break on a different line than they were originally set on.

When debugging OptiX/RTCore code, the following should be kept in mind:

- NVIDIA internal code cannot be debugged or examined by the user.
- OptiX/RTCode debugging is limited to `-lineinfo`, and building this code with full debug information (`-G`) is not supported.
- OptiX/RTCode code is highly optimized, and as such the notes above about debugging optimized code apply.

5.2.3. Compiling For Specific GPU architectures

By default, the compiler will only generate code for the compute_52 PTX and sm_52 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 dictate 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 7.0, add the following flag to the compilation command:

```
-gencode arch=compute_70,code=sm_70
```

To compile PTX code for any future architecture past the compute capability 7.0, add the following flag to the compilation command:

```
-gencode arch=compute_70,code=compute_70
```

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

5.3. Using the Debugger

CUDA-GDB can be used in the following system configurations:

5.3.1. Single-GPU Debugging with the Desktop Manager Running

For devices with compute capability 6.0 and higher CUDA-GDB can be used to debug CUDA applications on the same GPU that is running the desktop GUI.

Additionally for devices with compute capability less than 6.0 software preemption can be used to debug CUDA applications on the same GPU that is running the desktop GUI. There are two ways to enable this functionality:

Note: This is a BETA feature available on Linux and is only supported on Maxwell. The options listed below are ignored for GPUs with SM6.0 compute capability and higher.

- 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.

5.3.2. 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.

Note: 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`.

Note: 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.

5.3.3. 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 client and a Linux or QNX server.

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.

  Attaching to an already running process is not supported on QNX platforms.

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`.

5.3.4. Multiple Debuggers

For devices with compute capability 6.0 and higher several debugging sessions may take place simultaneously.

For devices with compute capability less than 6.0, 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
```

Additionally for devices with compute capability less than 6.0, 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.

5.3.5. 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. This feature is not supported on WSL.

Note: By default on some Linux distributions, the debugger cannot attach to an already running processes due to security settings. In order to enable the attach feature of the CUDA debugger, either cuda-gdb should be launched 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, edit `/etc/sysctl.d/10-ptrace.conf`.

Chapter 6. CUDA-GDB Extensions

6.1. 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
```

6.2. 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.

CUDA commands can also be queried using the `apropos` command.

6.3. 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.

6.4. 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.

6.5. GPU core dump support

There are two ways to configure the core dump options for CUDA applications. Environment variables set in the application environment or programmatically from the application with the CUDA Driver API.

Compilation for GPU core dump generation

GPU core dumps will be generated regardless of compilation flags used to generate the GPU application. For the best debugging experience, it is recommended to compile the application with the `-g -G` or the `-lineinfo` option with NVCC. See Compiling the Application for more information on passing compilation flags for debugging.

Enabling GPU core dump generation on exception with environment variables

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

Set the `CUDA_ENABLE_LIGHTWEIGHT_COREDUMP` environment variable to 1 in order to enable generating lightweight corefiles instead of full corefiles. When enabled, GPU core dumps will not contain the memory dumps (local, shared, global) of the application. This option is disabled by default.

Controlling behavior of GPU core dump generation

The CUDA_COREDUMP_GENERATION_FLAGS environment variable can be used when generating GPU core dumps to deviate from default generation behavior. Multiple flags can be provided to this environment variable and are delimited by `,`. These flags can be used to accomplish tasks such as reducing the size of the generated GPU core dump or other desired behaviors that deviate from the defaults. The table below lists each flag and the behavior when present.

Table 1: GPU core dump CUDA_COREDUMP_GENERATION_FLAGS

Environment Variable flag	Description
`skip_nonrelocated_elf_images`	Disables including copies of nonrelocated elf images in the GPU core dump. Only the relocated images will be present.
`skip_global_memory`	Disables dumping of GPU global and constbank memory segments.
`skip_shared_memory`	Disables dumping of GPU shared memory segments.
`skip_local_memory`	Disables dumping of GPU local memory segments.
`skip_abort`	Disables calling `abort()` at the end of the GPU core dump generation process.

Note: Setting the `CUDA_ENABLE_LIGHTWEIGHT_COREDUMP` environment variable to `1` is equivalent to `CUDA_COREDUMP_GENERATION_FLAGS="skip_nonrelocated_elf_images,skip_global_memory,skip_shared_memory,skip_local_memory"`.

Note: Setting the `CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION` environment variable to `0` is equivalent to `CUDA_COREDUMP_GENERATION_FLAGS="skip_abort"`.

Limitations and notes for core dump generation

The following limitations apply to core dump support:

- For Windows WDDM, GPU core dump is only supported on a GPU with compute capability 6.0 or higher. Windows TCC supports GPU core dump on all supported compute capabilities.
- GPU core dump is unsupported for the Windows Subsystem for Linux on GPUs running in SLI mode. Multi-GPU setups are supported, but SLI mode cannot be enabled in the Driver Control Panel.
- GPU core dump is supported for the Windows Subsystem for Linux only when the hardware scheduling mode is enabled.
- Generating a CPU core dump with `CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION` is currently unsupported on the QNX platform.
- GPU core dump is unsupported for the NVIDIA CMP product line.
- Per-context core dump can only be enabled on a GPU with compute capability 6.0 or higher. GPUs with compute capability less than 6.0 will return `CUDA_ERROR_NOT_SUPPORTED` when using the Coredump Attributes Control API.

- If an MPS client triggers a core dump, every other client running on the same MPS server will fault. The indirectly faulting clients will also generate a core dump if they have core dump generation enabled.
- GPU core dump is unsupported when other developer tools, including CUDA-GDB, are interacting with the application. Unless explicitly documented as a supported use case (e.g `generate-cuda-core-file` command).
- When generating a coredump on exception, if the kernel exits before the exception has been recognized it may result in failure to generate the corefile. See the note in GPU Error Reporting for strategies on how to work around this issue.

Note: The user should not send the application process a signal and ensure that the application process does not automatically terminate while the coredump generation is in process. Doing so may cause GPU coredump generation to abort.

Note: Starting from CUDA 11.6, the compute-sanitizer tool can generate a GPU core dump when an error is detected by using the `--generate-coredump yes` option. Once the core dump is generated, the target application will abort. See the compute-sanitizer documentation for more information: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html#coredump

Note: CPU core dumps will be located in a distribution specific location. Examining the `/proc/sys/kernel/core_pattern` file will typically hint at the name/location of the CPU core dump.

Note: NVIDIA vGPU platforms must explicitly enable debugging support to perform GPU core dump generation. Please reference the Virtual GPU Software User Guide for information on how to enable debugging on vGPU.

Note: NVIDIA Jetson and Drive Tegra devices must explicitly enable debugging support to perform GPU core dump generation. Refer to the *Using the CUDA-GDB debugger on Jetson and Drive Tegra devices* section.

Note: When generating core dumps on NVIDIA Drive Tegra devices running QNX, core dump generation may hang when generating CPU core dumps. If a hang is encountered, set `CUDA_ENABLE_CPU_COREDUMP_EXCEPTION` to 0.

Note: If core dumps are not generated when running programs built with OptiX/RTCore, try setting the environment variable `OPTIX_FORCE_DEPRECATED_LAUNCHER` to 1. Refer to the *Debugging OptiX/RTCore applications* section.

Note: If core dumps are not generated when running programs on Windows Subsystem for Linux, ensure the debug interface is enabled via setting the registry key `>HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\GPUDebugger\EnableInterface` to `(DWORD) 1`. Refer to the *Debugging on*

Windows Subsystem for Linux section.

Note: GPU core dump is supported on GPUs running with Confidential Compute mode only with devtools mode. See *Confidential Computing Deployment Guide <https://docs.nvidia.com/confidential-computing-deployment-guide.pdf>* for more details on how to enable the mode.

Naming of GPU core dump files

By default, a GPU core dump is created in the current working directory. It is named `core_TIME_HOSTNAME_PID.nvcudmp` where `TIME` is the number of seconds since the Epoch, `HOSTNAME` is the host name of the machine running the CUDA application and `PID` is the process identifier of the CUDA application.

The `CUDA_COREDUMP_FILE` environment variable can be used to define a template that is used to change the name of a GPU core dump file. The template can either be an absolute path or a relative path to the current working directory. The template can contain % specifiers which are substituted by the following patterns when a GPU core dump is created:

Specifier	Description
%h	Host name of the machine running the CUDA application
%p	Process identifier of the CUDA application
%t	Time as the number of seconds since the Epoch, `1970-01-01 00:00:00 +0000 (UTC)`

As an example, setting `CUDA_COREDUMP_FILE` to:

```
export CUDA_COREDUMP_FILE=newName.%h.%p
```

Would result in GPU core dumps being written to `newName.myhost.1234` relative to the current working directory. Here `myhost` and `1234` are replaced with the real host name and pid respectively.

Setting `CUDA_COREDUMP_FILE` to:

```
export CUDA_COREDUMP_FILE="/home/$USER/newName.%h.%p"
```

Would result in GPU core dumps being written to the user's home directory with the same name logic as in the above example.

If `CUDA_COREDUMP_FILE` points to an existing file of FIFO type (e.g named pipe), the core dump will be streamed to it.

Coredumps may be piped to shell commands via `CUDA_COREDUMP_FILE` with the following format:

```
export CUDA_COREDUMP_FILE='| cmd > file'
```

For example, to pipe a coredump to `gzip` use:

```
export CUDA_COREDUMP_FILE='| gzip -9 > cuda-coredump.gz'
```

Note: When piping a coredump, the % specifiers will not be recognized.

Enabling user induced GPU core dump generation

For the devices that support compute preemption, the user can interrupt a running CUDA process to generate the GPU core dump.

Set the CUDA_ENABLE_USER_TRIGGERED_COREDUMP environment variable to 1 in order to enable generating a user induced GPU core dump. This option is disabled by default. Setting this environment variable will open a communication pipe for each subsequently running CUDA process. To induce the GPU core dump, the user simply writes to the pipe.

To change the default pipe file name, set the CUDA_COREDUMP_PIPE environment variable to a specific pipe name. The default pipe name is in the following format: `corepipe.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. This environment variable can take % specifiers as decribed in the above section.

Displaying core dump generation progress

By default, when an application crashes and generates a GPU core dump, the application may appear to be unresponsive or frozen until fully generated.

Set the CUDA_COREDUMP_SHOW_PROGRESS environment variable to 1 in order to print core dump generation progress messages to `stderr`. This can be used to determine how far along the coredump generation is:

```
coredump: SM 1/14 has finished state collection
coredump: SM 2/14 has finished state collection
coredump: SM 3/14 has finished state collection
coredump: SM 4/14 has finished state collection
coredump: SM 5/14 has finished state collection
coredump: SM 6/14 has finished state collection
coredump: SM 7/14 has finished state collection
coredump: SM 8/14 has finished state collection
coredump: SM 9/14 has finished state collection
coredump: SM 10/14 has finished state collection
coredump: SM 11/14 has finished state collection
coredump: SM 12/14 has finished state collection
coredump: SM 13/14 has finished state collection
coredump: SM 14/14 has finished state collection
coredump: Device 1/1 has finished state collection
coredump: Calculating ELF file layout
coredump: ELF file layout calculated
coredump: Writing ELF file to core_TIME_HOSTNAME_PID.nvcudmp
coredump: Writing out global memory (1073741824 bytes)
coredump: 5%...
coredump: 10%...
coredump: 15%...
coredump: 20%...
coredump: 25%...
coredump: 30%...
coredump: 35%...
coredump: 40%...
coredump: 45%...
coredump: 50%...
coredump: 55%...
coredump: 60%...
coredump: 65%...
coredump: 70%...
coredump: 75%...
coredump: 80%...
coredump: 85%...
```

(continues on next page)

(continued from previous page)

```
coredump: 90%...
coredump: 95%...
coredump: 100%...
coredump: Writing out device table
coredump: Finalizing
coredump: All done
```

Enabling GPU core dump generation with the CUDA Driver API

The Driver API has equivalent settings for all of the environment variables, with the added feature of being able to set different core dump settings per-context instead of globally. This API can be called directly inside your application. Use `cuCoredumpGetAttributeGlobal` and `cuCoredumpSetAttributeGlobal` to fetch or set the global attribute. Use `cuCoredumpGetAttribute` and `cuCoredumpSetAttribute` to fetch or set the per context attribute. See the Coredump Attributes Control API manual for more information.

The table below lists the environment variables and the equivalent `CUcoredumpSettings` flags that are available to manage core dump settings with the Coredump Attributes Control API.

Note: The `CU_COREDUMP_ENABLE_USER_TRIGGER` setting can only be set globally in the driver API and `CU_COREDUMP_PIPE` must be set (if desired) before user-triggered core dumps are enabled.

Table 2: GPU core dump configuration parameters

Environment Variable	**Description**

Environment Variable:
`CUDA_ENABLE_COREDUMP_ON_EXCEPTION`
CUcoredumpSettings Flag:
`CU_COREDUMP_ENABLE_ON_EXCEPTION`
	Enables GPU core dump generation for exceptions. Disabled by default.

Environment Variable:
`CUDA_ENABLE_CPU_COREDUMP_ON_EXCEPTION`
CUcoredumpSettings Flag:
`CU_COREDUMP_TRIGGER_HOST`
	Triggers host (CPU) core dump after GPU core dump is complete. Enabled by default.

Environment Variable:
`CUDA_ENABLE_LIGHTWEIGHT_COREDUMP`
CUcoredumpSettings Flag:
`CU_COREDUMP_LIGHTWEIGHT`
	When enabled, GPU core dumps will not contain the memory dumps (local, shared, global) of the application. Disabled by default.

Environment Variable:
`CUDA_ENABLE_USER_TRIGGERED_COREDUMP`
CUcoredumpSettings Flag:
`CU_COREDUMP_ENABLE_USER_TRIGGER`
	Enables user triggerable core dumps by writing to a pipe defined in the `COREDUMP_PIPE` setting. Disabled by default.

Environment Variable:
`CUDA_COREDUMP_FILE`
CUcoredumpSettings Flag:
`CU_COREDUMP_FILE`
	Filename template for the GPU core dump.

Environment Variable:
`CUDA_COREDUMP_PIPE`
CUcoredumpSettings Flag:
`CU_COREDUMP_PIPE`
	Filename template for the user pipe trigger.

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.

Note: Coredump inspection does not require that a GPU be installed on the system

Chapter 7. 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.

7.1. 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.

7.2. 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)
```

(continues on next page)

(continued from previous page)

```
(cuda-gdb) cuda kernel
kernel 1
```

7.3. 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 8. 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

8.1. 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.

8.2. 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. When the CUDA thread in focus becomes divergent, behavior depends on the value of `set cuda step_divergent_lanes`. When on (default), the warp in focus will be continuously single-stepped until the CUDA thread in focus becomes active. When off, the warp in focus will be stepped and the focused CUDA thread will be changed to the nearest active lane in the warp.

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 thread barrier calls like: `__syncthreads()` or cluster-wide barriers. In this case, an implicit temporary breakpoint is set immediately after the barrier and all threads are resumed until the temporary breakpoint is hit.

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.

Asynchronous SASS instructions executed on the device, such as the warpgroup instructions, at prior PCs are not guaranteed to be complete.

With Dynamic Parallelism, several CUDA APIs can be called directly from device code. 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.
- On devices prior to Hopper (SM 9.0), stepping into the deprecated `cudaDeviceSynchronize()` results in undefined behavior. Users shall step over this call instead.
- 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).

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

Chapter 9. Breakpoints and Watchpoints

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

If a 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 reaches the 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. The `disable` command can be used to prevent hitting the breakpoint by additional threads.

9.1. 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
```

9.2. 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.

9.3. 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.

9.4. 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.

9.5. 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 until the CUDA module is loaded. To double check the desired conditional expression, first set an unconditional breakpoint at the desired location and continue. When the breakpoint is hit, evaluate the desired conditional statement by using the `cond` command.

9.6. 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 10. Inspecting Program State

10.1. 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`

10.2. 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
```

(continues on next page)

(continued from previous page)

```
(cuda-gdb) print *(@global void * const @parameter *) 0x10
$7 = (@global void * const @parameter) 0x110000</>
```

10.3. 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 active SMs in the current device

`warps`
information about all the active warps in the current SM

`lanes`
information about all the active 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.

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

10.3.1. 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 active GPUs are found. A device is not considered active until the first kernel launch has been encountered.

```
(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
```

10.3.2. 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
...
```

10.3.3. 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)
 ...
```

10.3.4. 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)
 ...
```

10.3.5. 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.

10.3.6. 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
 ...
```

10.3.7. 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
 ...
```

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

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
```

10.3.8. 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`.

Note: With `set cuda software_preemption on`, no kernel will be reported as active.

10.3.9. 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`.

10.3.10. 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
```

10.3.11. 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}
```

10.4. 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]
```

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

In the disassembly view, the current pc is prefixed with `=>`. For Maxwell (SM 5.0) and newer architectures, if an instruction triggers an exception it will be prefixed with `*>`. If the pc and errorpc are the same instruction it will be prefixed with `*=>`.

For example, consider the following exception:

```
CUDA Exception: Warp Illegal Address
The exception was triggered at PC 0x555555c08620 (memexceptions_kernel.cu:17)

Thread 1 "memexceptions" received signal CUDA_EXCEPTION_14, Warp Illegal Address.
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0,
↪sm 0, warp 0, lane 0]
0x0000555555c08fb0 in exception_kernel<<<(1,1,1),(1,1,1)>>> (data=0x7fffccc00000,
↪exception=MMU_FAULT) at memexceptions_kernel.cu:50
50  }
(cuda-gdb)
```

The `disas` command can be used to view both the PC and the error PC that triggered the exception.

```
(cuda-gdb) disas $pc,+16
Dump of assembler code from 0x555555c08fb0 to 0x555555c08fc0:
=> 0x0000555555c08fb0 <_Z16exception_kernelPv11exception_t+3504>:  ERRBAR
End of assembler dump.
```

```
(cuda-gdb) disas $errorpc,+16
Dump of assembler code from 0x555555c08620 to 0x555555c08630:
```

(continues on next page)

(continued from previous page)

```
*> 0x0000555555c08620 <_Z16exception_kernelPv11exception_t+1056>:  ST.E.U8.STRONG.SYS
↪[R6.64], R5
End of assembler dump.
```

10.5. 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
```

10.6. Const banks

Memory allocated in the constant address space of GPU memory resides in two dimensional arrays called constant banks. Constant banks are noted `c[X][Y]` where `X` is the bank number and `Y` the offset. The memory address of a given bank/offset pair is obtained via the convenience function `$_cuda_const_bank(bank, offset)`.

```
(cuda-gdb) disass $pc,+16
Dump of assembler code from 0x7fffd5043d40 to 0x7fffd5043d50:
=> 0x00007fffd5043d40 <_Z9acos_main10acosParams+1856>:  MOV R0, c[0x0][0xc]
End of assembler dump.
(cuda-gdb) p *$_cuda_const_bank(0x0,0xc)
$1 = 8
```

Chapter 11. 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.

11.1. Context Events

Any time a CUDA context is created, pushed, popped, or destroyed by the application, CUDA-GDB can optionally 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]
```

By default, context event notification is disabled. 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 (default).

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

  CUDA-GDB displays the context event notification messages.

11.2. 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).

Chapter 12. Automatic Error Checking

12.1. 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` CUDA API call failures are not reported
- `ignore` Warning message is printed for every fatal CUDA API call failure (default)
- `stop` The application is stopped when a CUDA API call returns a fatal error
- `ignore_all` Warning message is printed for every CUDA API call failure
- `stop_all` The application is stopped when a CUDA API call returns any error

Note: 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`.

12.2. 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.

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

Note: Warp errors may result in instructions to continue executing before the exception is recognized and reported. The reported `$errorpc` shall contain the precise address of the instruction that caused the exception. If the warp exits after the instruction causing exception has executed, but before the exception has been recognized and reported, it may result in the exception not being reported. CUDA-GDB relies on an active warp present on the device in order to report exceptions. To help avoid this scenario of unreported exceptions:

- For Volta+ architectures, compile the application with `-G`. See Compiling the Application for more information.
- Add `while(1);` before kernel exit. This shall ensure the exception is recognized and reported.
- Rely on the compute-sanitizer `memcheck` tool to catch accesses that can lead to an exception.

Table 1: CUDA Exception Codes

Exception Code	**Precision of the Error**	**Scope of the Error**	**Description**
CUDA_EXCEPTION_0 : "Device Unknown Exception"	Unknown	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 : "Deprecated"	Deprecated	Deprecated	This exception is deprecated and should be treated as `CUDA_EXCEPTION_0`.
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"	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"	Precise	Warp error	This occurs when any thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5 : "Warp Out-of-range Address"	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"	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.

continues on next page

Table 1 – continued from previous page

Exception Code	Precision of the Error	Scope of the Error	Description
CUDA_EXCEPTION_7 : "Warp Invalid Address Space"	Precise	Warp error	This occurs when any thread within a warp executes an instruction that accesses a memory space not permitted for that instruction.
CUDA_EXCEPTION_8 : "Warp Invalid PC"	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"	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"	Precise	Global error	This occurs when a thread accesses an illegal(out of bounds) global address.
CUDA_EXCEPTION_11 : "Deprecated"	Deprecated	Deprecated	This exception is deprecated and should be treated as `CUDA_EXCEPTION_0`.
CUDA_EXCEPTION_12 : "Warp Assert"	Precise	Per warp	This occurs when any thread in the warp hits a device side assertion.
CUDA_EXCEPTION_13 : "Deprecated"	Deprecated	Deprecated	This exception is deprecated and should be treated as `CUDA_EXCEPTION_0`.
CUDA_EXCEPTION_14 : "Warp Illegal Address"	Precise	Per warp	This occurs when a thread accesses an illegal(out of bounds) global/local/shared address.
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.
CUDA_EXCEPTION_13 : "Deprecated"	Deprecated	Deprecated	This exception is deprecated and should be treated as `CUDA_EXCEPTION_0`.
CUDA_EXCEPTION_17 : "Cluster Out-of-range Address"	Not precise	Per Cuda Cluster	This occurs when any thread within a block accesses an address that is outside the valid range of shared memory regions belonging to the cluster.

continues on next page

Table 1 – continued from previous page

Exception Code	**Precision of the Error**	**Scope of the Error**	**Description**
CUDA_EXCEPTION_18 : "Cluster Target Block Not Present"	Not precise	Per Cuda Cluster	This occurs when any thread within a block accesses another block that is outside the valid range of blocks belonging to the cluster.

12.3. Autostep

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.

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.

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 13. Walk-Through Examples

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

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

13.1. 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

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

// Simple 8-bit bit reversal Compute test

#define N 256

__global__ void bitreverse(void *data) {
   unsigned int *idata = (unsigned int*)data;
   extern __shared__ int array[];

   array[threadIdx.x] = idata[threadIdx.x];

   array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
                       ((0x0f0f0f0f & array[threadIdx.x]) << 4);
   array[threadIdx.x] = ((0xcccccccc & array[threadIdx.x]) >> 2) |
                       ((0x33333333 & array[threadIdx.x]) << 2);
   array[threadIdx.x] = ((0xaaaaaaaa & array[threadIdx.x]) >> 1) |
                        ((0x55555555 & array[threadIdx.x]) << 1);

   idata[threadIdx.x] = array[threadIdx.x];
}

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

```

(continues on next page)

(continued from previous page)

```
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 }
```

13.1.1. 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)
```

13.2. 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

```
#define NUM_BLOCKS 8
#define THREADS_PER_BLOCK 64

__global__ void example(int **data) {
  int value1, value2, value3, value4, value5;
```

(continues on next page)

(continued from previous page)

```
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 < NUM_BLOCKS; block++) {
29     for (int thread = 0; thread < THREADS_PER_BLOCK; thread++) {
30       int idx = thread + block * THREADS_PER_BLOCK;
31       cudaMalloc(&host_data[idx], sizeof(int));
32       cudaMemcpy(host_data[idx], &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**)&dev_data,  sizeof(host_data));
42   cudaMemcpy(dev_data, host_data, sizeof(host_data), cudaMemcpyHostToDevice);
43
44   /* Execute example */
45   example <<< 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.

13.2.1. 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.

13.3. Example: MPI CUDA Application

For large scale 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 be used.

If the cluster nodes have xterm support, launch CUDA-GDB in the same way you would launch gdb with your job launcher. For example:

```
$ 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:

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

Job launchers have different ways of exporting environment variables to the cluster nodes. Consult your job launcher documentation for more details.

When xterm is not supported by your cluster environment, you can insert a spin loop inside your program, ssh to the compute node(s), and attach onto the MPI processes. Somewhere near the start of your program, add a code snippet similar to 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);
    }
}
```

Recompile and launch the application. After it starts, ssh to the node(s) of interest and attach to the process using CUDA-GDB. 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
```

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

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

For larger applications, you can conditionalize the spin loop based on the MPI rank using the MPI_Comm_rank function.

For devices with compute capability below 6.0, the software preemption workaround described in Multiple Debuggers does not work with MPI applications. For those GPUs, ensure each MPI rank targets a unique GPU.

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.

Chapter 14. Tips and Tricks

This section serves as reference to advanced settings and various tips and tricks users of CUDA-GDB can utilize which are not documented elsewhere.

14.1. 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.

14.2. 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.

14.3. 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.

14.4. 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.

14.5. 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.

14.6. 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.

14.7. 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.

14.8. Debugging Docker Containers

When debugging an application within a Docker container, the PTRACE capability needs to be enabled. The user needs to also ensure that the root file system has both read/write permissions set.

To enable the PTRACE capability, add the following to your Docker run command:

```
--cap-add=SYS_PTRACE
```

14.9. Switching to Classic Debugger Backend

A new debugger backend named the Unified Debugger (UD) has been introduced on Linux platforms with the CTK 11.8 release. UD allows for a unified debugger backend shared with debugging tools such as cuda-gdb and NVIDIA® Nsight™ VSE. UD is supported across multiple platforms including both Windows and Linux. The end user experience with UD is transparent to existing tool use.

The previous debugger backend, known as the classic debugger backend, can still be used by setting `CUDBG_USE_LEGACY_DEBUGGER` to 1 in the environment before starting CUDA-GDB.

UD is not supported on Maxwell GPUs. Users must switch to the classic debugger backend to debug their applications on Maxwell GPUs.

14.10. Thread Block Clusters

CUDA applications that make use of Thread Block Clusters will see the cluster index displayed in the CUDA focus. Both cluster index and cluster dimension can be queried by printing the convenience variables `clusterIdx` and `clusterDim`.

14.11. Debugging OptiX/RTCore applications

When debugging programs built with OptiX/RTCore, it may be necessary to set the environment variable `OPTIX_FORCE_DEPRECATED_LAUNCHER` to 1. If breakpoints are unable to be hit, try setting this environment variable before starting your application.

14.12. Debugging on Windows Subsystem for Linux

If you are unable to use the debugger on Windows Subsystem for Linux, make sure the debug interface is enabled by setting the registry key `>HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\GPUDebugger\EnableInterface` to `(DWORD) 1`

Chapter 15. 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 supported by the current CUDA release.

GDB Python integration

GDB Python integration is supported in cuda-gdb with a multiple builds mechanism in order to support multiple python3 interpreters across different platforms. The cuda-gdb program is a shell script that selects the associated supported cuda-gdb binary based on the version of python available on the system. Support exists for the following Python versions: `Python 3.8`, `Python 3.9`, `Python 3.10`, `Python 3.11`, and `Python 3.12`

Windows Subsystem for Linux (WSL)

- cuda-gdb supports debugging CUDA application on WSL2.
- Make sure this capability is enabled via the registry key `>HKEY_LOCAL_MACHINE\SOFTWARE\NVIDIA Corporation\GPUDebugger\EnableInterface` set to `(DWORD) 1`.
- Debugging compute-intensive apps may require to increase or disable TDR.

Chapter 16. Common Issues on Supported Operating Systems

The following are known issues with the current release on supported operating systems and how to fix them.

Python not initialized

This happens due to a missing Python 3.x library on the machine, installing it fixes the issue. This can also be caused by having a mismatched major.minor version of libpython installed with the default python3 interpreter in PATH. A libpython version matching the default python3 interpreter in PATH must be available. The libpython version can be determined with the `python3 --version` command. For example, the following command would tell us that a libpython3.8.so* needs to be installed in a default library search path:

```
$ python3 --version
Python 3.8.10
```

Specific commands to install the proper libpython are below.

RHEL 8/9

```
$ sudo yum -y install python3-libs
```

Debian 10/11/12

```
$ sudo apt-get -y install libpython3-stdlib
```

Fedora 39

```
$ sudo yum -y install python3-libs
```

OpenSUSE 15

```
$ sudo zypper install -y libpython3
```

Ubuntu 20.04/22.04

```
$ sudo apt-get -y install python3.8 $ sudo apt-get -y install libpython3.8
```

Chapter 17. Known Issues

The following are known issues with the current release.

- Debugging on Hopper architecture with MCDM enabled is not supported on Windows or WSL.
- 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.
- When remotely debugging 32-bit applications on a 64-bit server, the cuda-gdbserver binary used must be 32-bit.
- Attaching to a CUDA application with Software Preemption enabled in cuda-gdb is not supported.
- Attaching to a CUDA application on QNX is not supported.
- Attaching to a 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.
- Significant performance degradation will occur when the 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.
- On Jetson, calls to the cuda API might result in the debugger jumping to _dl_catch_exception(). A workaround is to continue.
- On Jetson and Drive devices GPU debugging works correctly only if the debugger is run with the root permissions. Changes to devfs node permissions are required for the debugger to work without running as root.
- Debugger can miss reporting an induced trap(`__trap()`) in case it is the next instruction executed after the device resumes from a breakpoint.

- Debugger can miss reporting breakpoints or exceptions during resume in case new warps are launched on a previously empty SM.
- Debugger uses the libpython installed on the system. Use of Python scripting functionality will expose cuda-gdb to the same vulnerabilities as those in the system libpython version. It is recommended to always keep the system libpython library up-to-date.
- Debugger doesn't support accesses to shared memory allocations that are imported from other processes using the CUDA IPC APIs. Attempts to access these shared memory allocations by the debugger will result in an error stating access to memory allocations shared via IPC is not supported.
- `break_on_launch` will not function with OptiX/RTCore programs unless `OPTIX_FORCE_DEPRECATED_LAUNCHER` is set to 1.

Chapter 18. Notices

18.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

18.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

18.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright