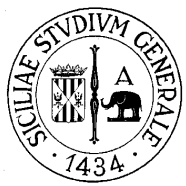
General-purpose programming on GPU

gdb and cuda-gdb

Eugenio Rustico
rustico@dmi.unict.it

D.M.I. - Università di Catania

Updated: May 29, 2011



Overview

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger runs an instance of the program in a controlled environment; it actually executes the program, but enable the programmer to:

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger runs an instance of the program in a controlled environment; it actually executes the program, but enable the programmer to:

- Run step-by-step at **language programming level**

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger runs an instance of the program in a controlled environment; it actually executes the program, but enable the programmer to:

- Run step-by-step at **language programming level**
- Display the value of variableas and expressions at runtime

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger runs an instance of the program in a controlled environment; it actually executes the program, but enable the programmer to:

- Run step-by-step at **language programming level**
- Display the value of variableas and expressions at runtime
- **Change** the execution flow

A debugger is a tool aimed to assist the programmer in finding *semantic bugs*. It does not debug itself.

A debugger runs an instance of the program in a controlled environment; it actually executes the program, but enable the programmer to:

- Run step-by-step at **language programming level**
- Display the value of variableas and expressions at runtime
- **Change** the execution flow
- New feature (gdb 7.0 and few commercial debuggers): reverse debugging, aka go back in time and undo also destructive operations by saving a series of *states*

The GNU Debugger (gdb) runs on Windows and Unix-like OSes. It has no native GUI, but several softwares provide a user friendly interface (e.g. ddd and most IDEs).

The GNU Debugger (gdb) runs on Windows and Unix-like OSes. It has no native GUI, but several softwares provide a user friendly interface (e.g. ddd and most IDEs).

It is a *source-level* or *symbolic* debugger, i.e. it is capable of analyzing a program at the programming language level (not just at assembly level). Symbolic debuggers are language-specific and require some extra information (*debugging symbols*) to map assembly instructions to source code.

Debugging symbols are produced at compile time (e.g. `-g` option in `gcc`) and integrated into the executable (in which case, executables are **much bigger**) or put apart (e.g. `...-dbg` apt/yum packages). They consist of information about

Debugging symbols are produced at compile time (e.g. `-g` option in `gcc`) and integrated into the executable (in which case, executables are **much bigger**) or put apart (e.g. `...-dbg` apt/yum packages). They consist of information about

- Which source code lines produced which assembly instructions

Debugging symbols are produced at compile time (e.g. `-g` option in `gcc`) and integrated into the executable (in which case, executables are **much bigger**) or put apart (e.g. `...-dbg` apt/yum packages). They consist of information about

- Which source code lines produced which assembly instructions
- Name of variables

Debugging symbols are produced at compile time (e.g. `-g` option in `gcc`) and integrated into the executable (in which case, executables are **much bigger**) or put apart (e.g. `...-dbg` apt/yum packages). They consist of information about

- Which source code lines produced which assembly instructions
- Name of variables

Together with `-g` it is usually defined the macro `_DEBUG` or `_ DEBUG _`; the source files may include special code (e.g. lots of `printf`) only if this is enabled:

```
#ifdef _DEBUG_
printf("Line %d, var is %.4f\n", _LINE_, myvar);
#endif
```

Let's dive in:

Let's dive in:

```
user@localhost:~$ gdb my_program
```

Let's dive in:

```
user@localhost:~$ gdb my_program
```

gdb has lots of options. If program needs its own arguments:

```
user@localhost:~$ gdb [gdb args] --args my_program arg1 arg2
```

```
  user@localhost:~$ gdb --args my_program input.xgm
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/user/my_program...done.
(gdb)_
```

gdb can **attach** to a running process (say, PID=123):

```
user@localhost:~$ gdb my_program 123
```

(can trace only the call stack if no debugging symbols are provided)

gdb can **attach** to a running process (say, PID=123):

```
user@localhost:~$ gdb my_program 123
```

(can trace only the call stack if no debugging symbols are provided)

It is possible to activate a Text User Interface (price: sacrifice some shortcuts...)

```
user@localhost:~$ gdb -tui my_program
```

(switch from/to pure console with "CTRL+x", then "a")

gdb can **attach** to a running process (say, PID=123):

```
user@localhost:~$ gdb my_program 123
```

(can trace only the call stack if no debugging symbols are provided)

It is possible to activate a Text User Interface (price: sacrifice some shortcuts...)

```
user@localhost:~$ gdb -tui my_program
```

(switch from/to pure console with "CTRL+x", then "a")

Finally, it is possible to execute scripts to automate sets of commands. See man gdb or type "help" in interactive mode for more useful options

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

User hits CTRL+C: SIGINT is trapped by gdb. To send it to the program, use `signal`

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

User hits CTRL+C: SIGINT is trapped by gdb. To send it to the program, use `signal`

The program runs an error: if we hit `run`, the operative system usually terminate sthe program

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

User hits CTRL+C: SIGINT is trapped by gdb. To send it to the program, use `signal`

The program runs an error: if we hit `run`, the operative system usually terminate sthe program

A breakpoint is reached (see next slide)

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

User hits CTRL+C: SIGINT is trapped by gdb. To send it to the program, use `signal`

The program runs an error: if we hit `run`, the operative system usually terminate sthe program

A breakpoint is reached (see next slide)

A watch is verified (see next slide)

When gdb starts, it breaks before any instruction is executed. Type `run` to run the program. It stops when:

User hits CTRL+C: SIGINT is trapped by gdb. To send it to the program, use `signal`

The program runs an error: if we hit `run`, the operative system usually terminate sthe program

A breakpoint is reached (see next slide)

A watch is verified (see next slide)

when stopped, it shows the **next line** yet to be executed

Breakpoint: *location* in source files where gdb stops; may be conditional, i.e. stop only if one or more conditions are verified.

Breakpoint: *location* in source files where `gdb` stops; may be conditional, i.e. stop only if one or more conditions are verified.

Watch: *condition* checked at every step of the execution; `gdb` stops when verified, in **any location** of the program.

Basic execution control:

run/r - executes, possibly restart program

continue/c - continue execution after a break or interruption

next/n - executes next instrucion (may be a function call) and stops again

step/s - enter next function step-by-step

finish/f - continue execution until current function (*frame*) finishes

Basic execution control:

run/r - executes, possibly restart program

continue/c - continue execution after a break or interruption

next/n - executes next instrucion (may be a function call) and stops again

step/s - enter next function step-by-step

finish/f - continue execution until current function (*frame*) finishes

We skip reverse-debugging features

Basic information:

print/p	- print value of given var/expression (must be available in current scope, of globally indicated)
display/d	- print value of given var/expression at every interruption
list/l	- show next 10 lines of code (takes parameters)
print/p	- print value of given var/expression (must be available in current scope, of globally indicated)
info frame	- show information about current execution frame (*scope*)
backtrace/bt	- show call stack

Adding breaks:

```
(gdb) break filename:row [if expr] [thread n]
(gdb) break filename:function [if expr] [thread n]
(gdb) break function [if expr] [thread n]
(gdb) break namespace::function [if expr] [thread n]
(gdb) watch expr
```

Managing breaks:

info break - show current breakpoints

info watch - show current watches

delete - delete given breakpoint/watch

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable
- Jumping to a specific point of source code

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable
- Jumping to a specific point of source code
- Changing at runtime the argument list

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable
- Jumping to a specific point of source code
- Changing at runtime the argument list
- Calling arbitrary functions with arbitrary parameters in any moment

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable
- Jumping to a specific point of source code
- Changing at runtime the argument list
- Calling arbitrary functions with arbitrary parameters in any moment
- Reverse-debugging (external operations will be done twice, e.g. `printf` or kernel launches)
- ...

We can actually **change the execution flow** of a program at runtime by

- Changing the content of a variable
- Jumping to a specific point of source code
- Changing at runtime the argument list
- Calling arbitrary functions with arbitrary parameters in any moment
- Reverse-debugging (external operations will be done twice, e.g. `printf` or kernel launches)
- ...

`help set` shows other flow control options

```
(gdb) jump myfunction()
(gdb) jump main.cc:32
(gdb) set args other_input_file.xpm
(gdb) call fopen(myfile)
(gdb) call MyClass::myMethod("string")
(gdb) reverse-next 5
```

```
(gdb) jump myfunction()
(gdb) jump main.cc:32
(gdb) set args other_input_file.xpm
(gdb) call fopen(myfile)
(gdb) call MyClass::myMethod("string")
(gdb) reverse-next 5
```

In any moment, hit TAB for **autocompletion**

Overview

How to deal with **multi-thread applications**?

How to deal with **multi-thread applications**?

- Any interruption stops *at least* a thread

How to deal with **multi-thread applications**?

- Any interruption stops *at least* a thread
- Breaks are valid in all threads, unless "thread" option is specified

How to deal with **multi-thread applications**?

- Any interruption stops *at least* a thread
- Breaks are valid in all threads, unless "thread" option is specified
- It is possible to make all threads hold until we explicitly switch to them

How to deal with **multi-thread applications**?

- Any interruption stops *at least* a thread
- Breaks are valid in all threads, unless "thread" option is specified
- It is possible to make all threads hold until we explicitly switch to them
- Reverse-debugging is not available

info threads - list threads

thread/t - switch to given thread

info threads - list threads

thread/t - switch to given thread

How do threads behave while one is interrupted? We decide with

```
(gdb) set scheduler-locking [off|on|step]
```

info threads - list threads

thread/t - switch to given thread

How do threads behave while one is interrupted? We decide with

```
(gdb) set scheduler-locking [off|on|step]
```

off - default; all threads but the interrupted one may execute, up to the s.o. scheduler

step - every thread is stopped if current is stopped; when current is running or stepping, other threads may be executing

on - every thread is stopped until we explicitly switch to it and run/continue/step (useful to debug obscure race conditions)

CUDA-based programs are normal programs plus the fact that they "dialogue" with the GPU.

CUDA-based programs are normal programs plus the fact that they "dialogue" with the GPU.

A kernel launch as we saw it the high-level APIs actually is a set of normal function calls to the CUDA runtime.

CUDA-based programs are normal programs plus the fact that they "dialogue" with the GPU.

A kernel launch as we saw it the high-level APIs actually is a set of normal function calls to the CUDA runtime.

With `call` command one can easily call CUDA runtime libraries (e.g. additional memcpys to check intermediate values)

However, it is not possible with `gdb` to step *inside* a kernel launch; `cuda-gdb` allows it.

However, it is not possible with `gdb` to step *inside* a kernel launch; `cuda-gdb` allows it.

`cuda-gdb` is a fork of `gdb` (at the moment, `cuda-gdb` is based on gdb 6.6 - no reverse debugging!) capable of stepping into kernels; up to CUDA 3.2, it is not available for Mac.

Stepping into kernels requires locking the GPU; this is only possible on multi-GPU systems or when integrated chipsets are used for visualization instead of NVIDIA cards.

Stepping into kernels requires locking the GPU; this is only possible on multi-GPU systems or when integrated chipsets are used for visualization instead of NVIDIA cards.

It is possible to step into a kernel **with the granularity of a warp** (recall: SIMD architecture!)

Stepping into kernels requires locking the GPU; this is only possible on multi-GPU systems or when integrated chipsets are used for visualization instead of NVIDIA cards.

It is possible to step into a kernel **with the granularity of a warp** (recall: SIMD architecture!)

Let's do an overview of the extensions `cuda-gdb` provides to `gdb`.

First, we need to compile with debugging symbols for both host and device (`-g -G` to `nvcc`).

First, we need to compile with debugging symbols for both host and device (`-g -G` to `nvcc`).

From nvcc help:

```
--debug (-g)
        Generate debug information for host code.

--device-debug <level>                          (-G)
        Generate debug information for device code,
        plus also specify the  optimization level
        for the device code in order to control its
        debuggability.
```

```
user@host:~$ nvcc -g -G -o my_program my_program.cu
```

Break into applications:

```
user@host:~$ cuda-gdb --args my_program my_input.png
[...]
(cuda-gdb) break mykernel_name
(cuda-gdb) run
```

Break into applications:

```
user@host:~$ cuda-gdb --args my_program my_input.png
[...]
(cuda-gdb) break mykernel_name
(cuda-gdb) run
```

Alternatively, attach to running applications (even those which seem freezed or in infinite loop), then CTRL+C and step into

Get **general information**:

```
(cuda-gdb) info cuda system
[...]
(cuda-gdb) info cuda device(s)
[...]
(cuda-gdb) info cuda lane
[...]
```

The latter also gives the number of divergent threads

Print **content or type of array**:

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

The latter also tells if `array` is in global or shared memory

Print **content or type of array**:

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

The latter also tells if `array` is in global or shared memory

Print content of an **arbitrary shared memory address**:

```
(cuda-gdb) p *(@shared int*)0x20
$3 = 0
```

Enable CUDA MemoryChecker to detect global memory violations and misaligned global memory accesses

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

Enable CUDA MemoryChecker to detect global memory violations and misaligned global memory accesses

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

Inspect current kernel coordinates:

```
(cuda-gdb) cuda kernel
[Current CUDA kernel 0 (device 0, sm 0, warp 0, lane 0, grid 1,
block (0,0), thread (0,0,0))]
```

Switch kernel coordinates:

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

Switch kernel coordinates:

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

In case of multiple GPUs, **inspect/switch device**:

```
(cuda-gdb) cuda device
[...]
(cuda-gdb) cuda device 2
[...]
```

See `cuda-gdb.pdf` in CUDA downloads for more usage information and limitations

See `cuda-gdb.pdf` in CUDA downloads for more usage information and limitations

Practice time!