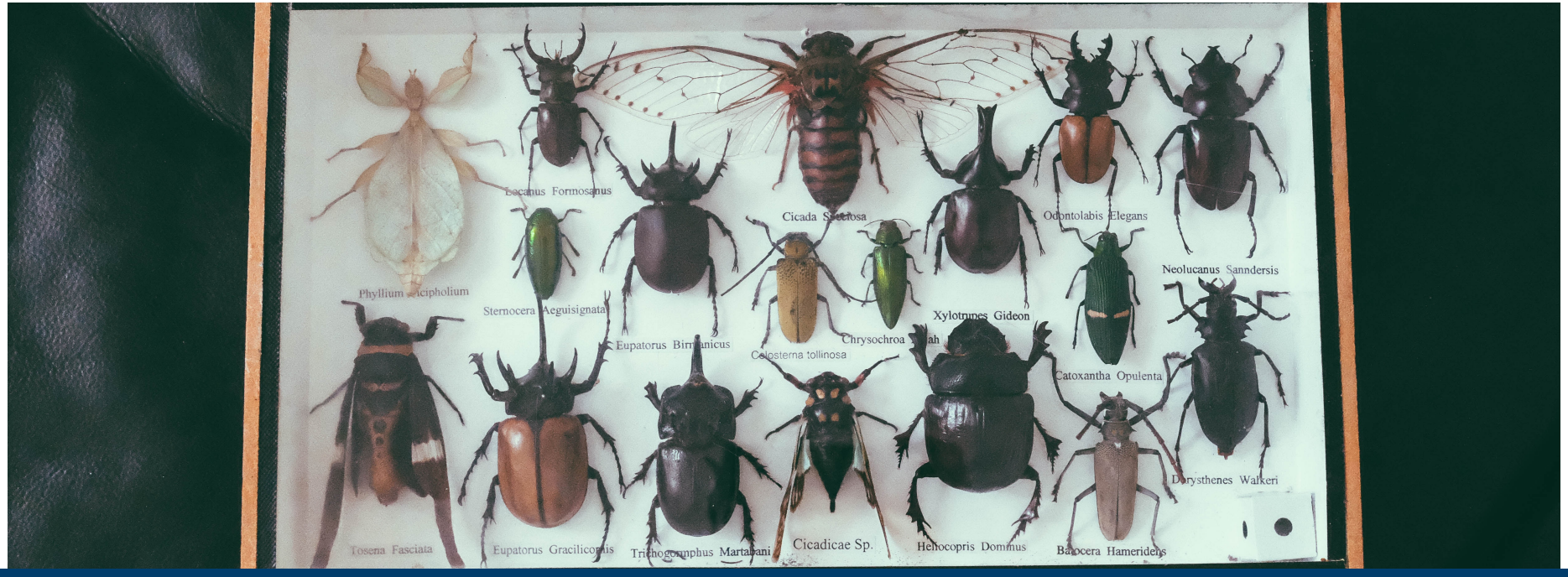

Tools and Methods for Debugging
OpenACC Course 2024

Oct 2024 | T. Hater | Forschungszentrum Jülich

Motivation

“It was on one of my journeys between the EDSAC room and the punching equipment that ‘hesitating at the angles of stairs’ the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.”

Sir Maurice Wilkes

Motivation

> *"It was on one of my journeys between the EDSAC room and the punching equipment that 'hesitating at the angles of stairs' the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs."*

Sir Maurice Wilkes

```
$> # Fast forward 70 years
$> ./spmv
call to cuStreamSynchronize returned error 700:
   Illegal address during kernel execution
```

General notes

Getting Ready

- Keep a notepad ready. At every step, write down…
  - …what you *do*.
  - …what you *see*.
  - …what you *think*.
- Work with version control
  - Check in intermediate steps.
  - Write down the current commit (hash).
  - Learn to use `git bisect`.
- Formulate hypotheses and design experiments to test them.
- Tasks will have some guiding questions!

General notes

Building for debug

When using CUDA (nvcc)

`-g`	CPU-side debug info
`-G`	GPU-side debug info (expensive, forces `-O0`)
`-lineinfo`	GPU-side light-weight info for profilers, but still helpful

General notes

Building for debug

When using CUDA (nvcc)

`-g`	CPU-side debug info
`-G`	GPU-side debug info (expensive, forces `-O0`)
`-lineinfo`	GPU-side light-weight info for profilers, but still helpful

When using OpenACC (nvc)

`-g`	CPU debug information; almost no overhead
`-gpu=debug`	GPU debug information; significant overhead
`-O0`	Disable optimisation. **NB.** Slow, may actually *hide* bugs.
`-gpu=lineinfo`	correlate GPU assembly to source code; lightweight
	Check compiler output: `-Minfo=accel`

Lightweight Tracing

Unravel the flow of kernels, might expose infinite loops and gives a coarse approximation of the call stack.
In particular: What was the last kernel launch?

`NV_ACC_TIME`	Lightweight profiler for time of data movement and kernels
`NV_ACC_NOTIFY`	Print information for GPU-related events.

Lightweight Tracing

Unravel the flow of kernels, might expose infinite loops and gives a coarse approximation of the call stack.
In particular: What was the last kernel launch?

`NV_ACC_TIME` Lightweight profiler for time of data movement and kernels

`NV_ACC_NOTIFY` Print information for GPU-related events.

- `=1` … kernel launches only
- `=2` … data transfers only
- `=3` … kernel launches and data transfers
- `=4` … region entry/exits only
- `=5` … region entry/exits and kernel launches
- `=8` … wait operations, synchronizations
- `=16` … (de)allocation of device memory

Task 0

TASK
C F90

- Compile example with debug info.
- Run the executable with tracing enabled.
- Use different levels of tracing, but 3 seems most useful here.
- Which kernels are suspicious?

```
$> cd Debugging/tasks/{C,Fortran}/spmv/ # Choose language
$> make                                 # Build
$> eval $JSC_SUBMIT_CMD --pty bash -i   # Start shell
$> NV_ACC_NOTIFY=3 ./spmv.bin           # Run program
$> exit                                 # Leave shell
```

Example

```
$> NV_ACC_NOTIFY=3 ./spmv.bin
upload CUDA data  file=.../spmv.c
function=main line=36 device=0 threadid=1 variable=row_ptr bytes=15705192
upload CUDA data  file=.../spmv.c
function=main line=36 device=0 threadid=1 variable=row_ptr bytes=16777216
[...]
launch CUDA kernel  file=.../spmv.c
function=main line=42 device=0 threadid=1 num_gangs=65535 [...]
block=128 shared memory=1024
launch CUDA kernel  file=.../spmv.c
function=main line=42 device=0 threadid=1 num_gangs=65535 [...]
block=128 shared memory=1024
[...]
download CUDA data  file=.../spmv.c
function=main line=59 device=0 threadid=1 variable=y bytes=14633352
download CUDA data  file=.../spmv.c
function=main line=59 device=0 threadid=1 variable=y bytes=16777152
[...]
Runtime 0.172498 s.
```

compute-sanitizer

Command-line memory access analyzer

A majority of bugs (and vulnerabilities) are related to memory access.

- Memory error detector; similar to Valgrind's `memcheck`
- One of most helpful tools for error-finding.
  - Out-of-bounds accesses.
  - Kernels/API execution failures.
  - Memory leaks.
  - Use-after-free.
- Has more sub-tools, via `compute-sanitizer --tool` *NAME*:

→ http://docs.nvidia.com/cuda/compute-sanitizer/

Task 1

TASK
C F90

- Get an interactive shell as before
- Run the example with `compute-sanitizer ./spmv.bin`
- What do you think went wrong?
- In which kernel and file/line is the error encountered?
- What are the block and thread where the error happens?

Example

```
$> compute-sanitizer ./spmv.bin
===== Invalid __global__ read of size 8
=====     at 0x00000870 in .../spmv.c:50:main_42_gpu
=====     by thread (26,0,0) in block (123,0,0)
=====     Address 0x23aba45b60 is out of bounds
=====     Saved host backtrace up to driver entry point at kernel launch time
=====     Host Frame:/usr/lib64/nvidia/libcuda.so (cuLaunchKernel + 0x2c5) [0x204235]
[...]
=====     Host Frame:./spmv.bin [0xe69]
=====
===== Invalid __global__ read of size 8
=====     at 0x00000870 in .../spmv.c:50:main_42_gpu
=====     by thread (25,0,0) in block (123,0,0)
=====     Address 0x23aba45b58 is out of bounds
=====     Saved host backtrace up to driver entry point at kernel launch time
=====     Host Frame:/usr/lib64/nvidia/libcuda.so (cuLaunchKernel + 0x2c5) [0x204235]
[...]
=====     Host Frame:./spmv.bin [0x192b]
```

cuda-gdb

Symbolic debugger

- Extension to GDB, a powerful symbolic debugger.

cuda-gdb

Symbolic debugger

- Extension to GDB, a powerful symbolic debugger.
- Works with C, C++, FORTRAN, and many more.

cuda-gdb

Symbolic debugger

- Extension to GDB, a powerful symbolic debugger.
- Works with C, C++, FORTRAN, and many more.
- Highly recommended to learn for all developers.

cuda-gdb

Symbolic debugger

- Extension to GDB, a powerful symbolic debugger.
- Works with C, C++, FORTRAN, and many more.
- Highly recommended to learn for all developers.

→ https://www.gnu.org/software/gdb/documentation/

cuda-gdb

Symbolic debugger

- Extension to GDB, a powerful symbolic debugger.
- Works with C, C++, FORTRAN, and many more.
- Highly recommended to learn for all developers.

→ https://www.gnu.org/software/gdb/documentation/

→ http://docs.nvidia.com/cuda/cuda-gdb/

Task 2

TASK
C F90

- Obtain an interactive shell
- Run the example under GDB `cuda-gdb ./spmv.dbg`
- Try out this sequence

```
(gdb) run              # Start; assume error here.
(gdb) backtrace        # show call stack
(gdb) list             # source code at break
```

Recap so far

Now we have the following tools available to us to find

- `NV_ACC_NOTIFY` lightweight trace of kernels.

This should be enough to find almost 80% of all bugs (by volume!); what follows is for the remaining 20%.

Recap so far

Now we have the following tools available to us to find

- `NV_ACC_NOTIFY` lightweight trace of kernels.
- `compute-sanitizer` location of erroneous memory accesses.

This should be enough to find almost 80% of all bugs (by volume!); what follows is for the remaining 20%.

Recap so far

Now we have the following tools available to us to find

- `NV_ACC_NOTIFY` lightweight trace of kernels.
- `compute-sanitizer` location of erroneous memory accesses.
- `cuda-gdb` sequence of calls leading to the error site.

This should be enough to find almost 80% of all bugs (by volume!); what follows is for the remaining 20%.

Breakpoints

Interrupt execution when a certain (source) location is reached.

```
(gdb) break foo       # break at function/kernel/template
(gdb) break file:line # break at line
```

Helpful to look at programm state outside of corruption, including variables and callstacks.

Interlude: OpenACC and cuda-gdb

- We are using CUDA tools with OpenACC.
- This works, but requires some *name mangling*.
- Pattern: *function_line*_gpu

```
      C main_42_gpu
Fortran spmv_26_gpu
```

- Recipe for extracting all kernel names

```
$> strings ./app `# Extract runs of ≥ 4 printable characters` \
 | grep .*_gpu   `# Filter out those ending on _gpu`          \
 | sort          `# Sort and reduce to unique elements`      \
 | uniq
```

At The Breakpoint

When execution is halted the program state can be inspected.

```
(gdb) backtrace          # show stack of functions to here
(gdb) list               # show source code context
(gdb) print bar          # print value of variable
(gdb) print arr[0]@4     # print first 4 values in array
```

Breakpoint ctd

When done, continue execution in steps or normally. Unused breakpoints can be removed.

```
(gdb) continue    # resume until next breakpoint (same if in loop)
(gdb) step        # resume until next line (descend into calls)
(gdb) next        # same, but do not follow calls
(gdb) delete <id> # remove breakpoint `id` returned by `break`
```

Task 3

TASK
C F90

- Start the program using `cuda-gdb ./spmv.dbg`. (NB. different suffix!)
- Set a breakpoint at the broken kernel.
- Try stepping through the kernel and inspect the local variables.
- **NB.** Use the `.dbg` suffix! Check that the kernel name is correct.
- **NB.** nvc is extremely aggressive in optimising out local variables and many of them will not be inspectable.

Example

```
$> cuda-gdb ./spmv.dbg
NVIDIA (R) CUDA Debugger
[...]
Reading symbols from .../spmv.dbg...done.
(cuda-gdb) break main_42_gpu
Function "main_42_gpu" not defined.
Make breakpoint pending on future shared library load! (y or [n]) y
Breakpoint 1 (main_42_gpu) pending.
(cuda-gdb) run
Starting program: .../spmv.dbg
[...]
[New Thread 0x2aab581ff700 (LWP 14126)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), [...]]
Breakpoint 1, main_42_gpu<<<(65535,1,1),(128,1,1)>>> ([...]) at spmv.c:43
43            for (int row=0; row<num_rows; ++row)
(cuda-gdb) print x[0]@8
[...]
(cuda-gdb) continue
```

The CUDA in cuda-gdb

Information

- grids, blocks, threads (logically)
- devices, SMs, warps, lanes (physically)
- CUDA contexts (driver)
- Managed variables and registers

Some examples:

```
(cuda-gdb) info cuda <item>     # item = devices, sms, warps, lanes
(cuda-gdb) info cuda <item>     # item = kernels, blocks, threads
```

The CUDA in cuda-gdb

Information

- grids, blocks, threads (logically)
- devices, SMs, warps, lanes (physically)
- CUDA contexts (driver)
- Managed variables and registers

Some examples:

```
(cuda-gdb) info cuda <item>     # item = devices, sms, warps, lanes
(cuda-gdb) info cuda <item>     # item = kernels, blocks, threads
```

NB

Remember the GPU execution model and use -Minfo=accel output for mapping.

The CUDA in cuda-gdb

Focus

cuda-gdb is working with a subset of threads called the ‘focus’.

```
(cuda-gdb) # Show current focus
(cuda-gdb) cuda device sm warp lane block thread
(cuda-gdb) # Change focus
(cuda-gdb) cuda device <d> sm <s> warp <w> lane <l> block <b> thread
↪  <t>
(cuda-gdb) # in both cases, you can leave out items
```

Working with the focus is extremely helpful when only some threads produce an error, eg out-of-bounds access.

Task 4

TASK
C F90

- Observe the kernel run as before, but now switch the focus.
- Try to find (the set of) threads where the error occurs.
- Can you refine the hypothesis of what is wrong?

Enhance Debugging for GPUs

Notify on Kernel Entry

Issue

I want to know when a kernel starts or start debugging the first kernel launched.

Enhance Debugging for GPUs

Notify on Kernel Entry

Issue

I want to know when a kernel starts or start debugging the first kernel launched.

```
(cuda-gdb) set cuda kernel_events <value> # where <value> can be
(cuda-gdb) # none          (default) no notifications
(cuda-gdb) # application   directly launched kernels
(cuda-gdb) # system        kernels launched by driver, eg memset
(cuda-gdb) # all           any kernel launch
(cuda-gdb)
(cuda-gdb) # Set a breakpoint on first instruction of all kernels
(cuda-gdb) set cuda break_on_launch application
```

The CUDA in cuda-gdb

Printing Special Memory

CUDA data arrays can live in global, shared, or texture memory.
Parameters in particular are situated in global memory.

```
(cuda-gdb) # Given some address `0xc0ffee' and an array `array'
(cuda-gdb) # Print value located at address
(cuda-gdb) print *(@shared  float*) 0xc0ffee  # shared memory
(cuda-gdb) print *(@texture float*) 0xc0ffee  # texture
(cuda-gdb) # NB. These work regardless of shared/texture/...
(cuda-gdb) print &array                       # address of
(cuda-gdb) print array[0]@4                   # first four items
```

Enhance Debugging for GPUs

Locating Launch and API Errors

Issue

Kernel and API errors may be reported asynchronously, due to the way launching kernels works. This can be disabled for debugging, however, if your bug is timing related, it might get hidden.
Similarly, `cuda-gdb` can check for memory errors.

Enhance Debugging for GPUs

Locating Launch and API Errors

Issue

Kernel and API errors may be reported asynchronously, due to the way launching kernels works. This can be disabled for debugging, however, if your bug is timing related, it might get hidden.
Similarly, cuda-gdb can check for memory errors.

```
(cuda-gdb) set cuda launch_blocking on     # synchronous kernels
(cuda-gdb) set cuda api_failure stop       # break at API errors
(cuda-gdb) set cuda memcheck on            # check for violations
(cuda-gdb) # of segments and alignment. Implies blocking launches.
```

Enhance Debugging for GPUs

Getting rid of `optimized out`

Issue

Printing some variables will only show an error. These have been spilled and are not currently visible.
Some can still be printed by requesting the *last known value*.

Enhance Debugging for GPUs

Getting rid of `optimized out`

Issue

Printing some variables will only show an error. These have been spilled and are not currently visible.
Some can still be printed by requesting the *last known value*.

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

Enhance Debugging for GPUs

Auto-stepping a Region

Issue

Errors will not be localised accurately if not single-stepping using `step` or `next`. However, this is slow.

Enhance Debugging for GPUs

Auto-stepping a Region

Issue

Errors will not be localised accurately if not single-stepping using step or next. However, this is slow.

```
(cuda-gdb) set autostep <start> for <count> lines
(cuda-gdb) # every time line <start> is hit, the programm will be
(cuda-gdb) # single stepped for <count> lines.
(cuda-gdb) # Errors will report immediately.
```

Summary and Conclusion

- Never use `printf` again for debugging.
- Simplify before using a debugger.
- Ask more complex questions about the location of the issue.
- In general: All the CUDA tools work
  - `compute-sanitizer`
  - `cuda-gdb`
- Some effort needed to translate, eg autogenerated kernels

Summary and Conclusion

- Never use `printf` again for debugging.
- Simplify before using a debugger.
- Ask more complex questions about the location of the issue.
- In general: All the CUDA tools work
  - `compute-sanitizer`
  - `cuda-gdb`
- Some effort needed to translate, eg autogen… …els

Task 5: Finally

TASK
C F90

- Armed with what you have learned so far, fix that bug.
- You should have a pretty good idea of what is going on by now.
- Do you understand what the program is trying to do?