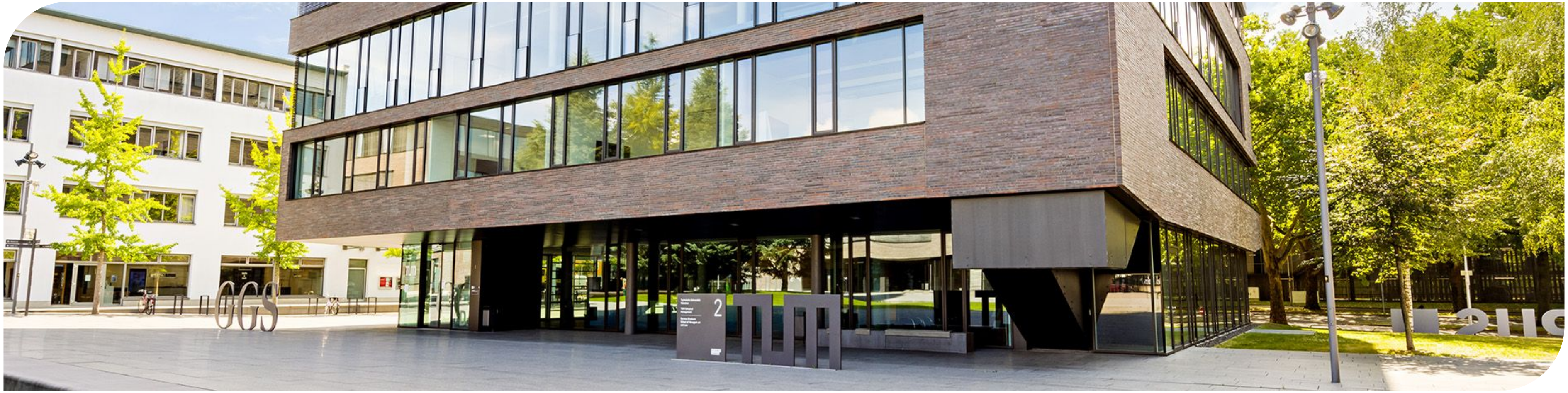
Technical University of Munich

Tutorial 8: Debugging CUDA programs

Informatik elective: GPU Computing

Pratik Nayak

Licensed under CC BY SA



Debugging GPU programs

- Debugging code on CUDA GPUs can be challenging, but CUDA provides some tools to help.
- Always check for error codes. All CUDA runtime API calls return an error code and can be easily converted to a human readable string with

```
__host__ __device__ const char * cudaGetErrorString(cudaError_t err);
```

- Note that CUDA kernel launches are asynchronous. So, an API call may return error code from a previous launch.

Checking errors from kernels

- CUDA kernels can produce two types of errors:
  - Synchronous: Which you can detect when launched
  - Asynchronous: Which occur during device code execution
- For synchronous errors you can just use `cudaGetLastError()` or `cudaPeekAtLastError()`
- For asynchronous errors, it is slightly more complicated:
  - You can force synchronization with `cudaDeviceSynchronize()` and then get the error immediately, but this can be expensive, but can be hidden under debug macros
  - You can also set an environment variable `export CUDA_LAUNCH_BLOCKING=1` to force the runtime to perform synchronous kernel execution

 CC BY SA

Recoverable v/s non-recoverable errors

- CUDA kernels can produce two types of errors:
  - Recoverable: Which you can catch and which do not corrupt the CUDA context, and subsequent calls behave normally (can return their own errors).
    - Example: `ptr = cudaMalloc(1<<64); // out of memory error`
  - Non-recoverable : Which occur only during device code execution. The context is corrupted in this case, and the CUDA runtime API is no-longer usable. The owning host process needs to be terminated.
    - Example: kernel time-out, illegal instruction, misaligned address etc.

CC BY SA

Debugging tools: Compute sanitizer

- Compute sanitizer is a functional correctness checking tool available with CUDA toolkit.
- It provides nice and automatic API error checking.
- It has support for various types of sub-tools:
    - Memcheck: detect illegal code activity: misaligned access, illegal access etc.
    - Racechecks: RAW, WAR, WAW hazards
    - Initcheck: Check accesses to uninitialized global memory
    - Synccheck: detects illegal use of synchronization primitives such as `__syncthreads()`
- More documentation: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/

Debugging tools: Compute sanitizer

- Using compute sanitizer is fairly simple:

```
$ compute-sanitizer [options] app_name [app_args]
```

- By default only Memcheck is used, but other sub-tools can be specified with

```
$ compute-sanitizer --tool <memcheck,racecheck,initcheck,synccheck>
```

- By default it runs kernels in non-blocking mode (concurrently). You can set that to be blocking as well with `--force-blocking-launches`
- You can also filter specific kernels: `--kernel-name` and `--kernel-name-exclude`

CC BY SA

Memcheck

- CUDA Memcheck detects runtime memory errors. It detects out of bounds and misaligned memory accesses in global, shared and even local memory.
- It also detects memory leaks: Malloc but no corresponding free.
- It is fairly comprehensive in reporting.
- Tip: Use `--print-limit <num>` to limit the amount of errors printed out to screen.
- It can also detect stream ordered allocation issues with `--track-stream-ordered-races all` option
    - Use before alloc and use after free errors between streams with incorrect synchronizations

CC BY SA

Racecheck

- Detects the Read-after-Write (RAW), Write-after-Write (WAW) and Write-after-Read (WAR) hazards.
- But only for shared memory accesses (!).
- First run memcheck and ensure no errors are produced, and then run racecheck.

 CC BY SA

Initcheck and Synccheck

- Initcheck detects uninitialized global memory accesses.
- Detects if a memory is being accessed without a memcpy or memset being called.
- Can also detect memory allocated but that was unused : `--track-unused-memory`
- Synccheck checks if the application is correctly using the synchronization primitives:
    - `__syncthreads()`, `__syncwarp()` and also cooperative group syncs

CC BY SA

Stack backtraces

- Back traces can be useful to trace back to the call site. It contains a list of frames at the time the error occurred.
- Build the application with symbol support: `-rdynamic`
- Also a good idea to add `-G` or `-lineinfo` to get source line information in the backtraces and error outputs.

Coredumps

- You can also tell it to dump a core file which you can analyze later with cuda-gdb.
- With --generate-coredump it will generate a coredump and then abort.
- You can then use it within cuda-gdb

```
$ (cuda-gdb) target cudacore core.name.nvcudmp
```

CUDA-GDB

- CUDA also provides an extension to gdb, enabling GNU debugger type debugging for CUDA applications.
- If you have used gdb before, then this is very similar. Most commands are the same.
- Using a gdbserver, you can connect to a remote and then run applications on the remote while debugging on your local machine.

```
$(host) ssh username@remote -L host_port:localhost:remote_port # connect to remote and
                                                               # open a port on remote to
                                                               # connect back from
$(username@remote) cuda-gdbserver :remote_port ./app_exec [app_args]
```

```
$(host) cuda-gdb
$(host)(cuda-gdb) target remote :host_port
```

Compilation for cuda-gdb and using it

- For correct cuda-gdb debugging, to ensure the symbol information is available, compile your code with:
  - `-g`: host code debug flag
  - `-G`: device code debug flag
- And ensure correct architecture flags are added for the GPU you have. `-arch=sm_80` (for our cluster)
- `set cuda [options]` : To set options for the debug run
- `info cuda [options]`: Get info on system config
- `cuda device sm warp lane block thread`: Get the current focus (thread you are looking at)
- `cuda device 0 sm 1 warp 2 lane 3:` Switch the focus to CUDA kernel 1, grid 2, block (8,0,0), thread

Break points and introspection

- A general workflow is to set break points
  - on functions
    - `(cuda-gdb) break func_name`
    - `(cuda-gdb) break class::func_name`
  - on line numbers
    - `(cuda-gdb) break file.cu:34`
  - on addresses
    - `(cuda-gdb) break *0x1afe34b0`
- In case you are not familiar with gdb: https://www.cs.cmu.edu/~gilpin/tutorial/
- More documentation here: https://docs.nvidia.com/cuda/cuda-gdb/index.html#breakpoints-and-watchpoints

Break points and introspection

- Once you have set a break point and are inside a kernel, you can print variables

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

- Or also memory indexed from the starting offset

```
(cuda-gdb) print *(@shared int*)0x20
$3 = 0
(cuda-gdb) print *(@shared int*)0x24
$4 = 128
(cuda-gdb) print *(@shared int*)0x28
$5 = 64
```

 CC BY SA

Information with the `info` command

- `info` command is useful to gather information on current focus

```
(cuda-gdb) info cuda sms
 SM Active Warps Mask
Device 0
* 0 0xffffffffffffffff
  1 0xffffffffffffffff
...
```

- Also more targeted information

```
(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)
 ...
```


cc BY SA

Information with the `info` command

- Active kernel info is also available

```
(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=...
```

- And also registers

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

Disassembly

- Use standard gdb disassembly instructions (`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] // current pc (=>)
```

- Ensure `cuobjdump` is installed and in `$PATH`
- `$pc` gives the program counter, which you can use to disassemble the next/previous instructions

CC BY SA

An easier approach

- Use your IDE integration:
  - VSCode: https://docs.nvidia.com/nsight-visual-studio-code-edition/cuda-debugger/index.html
  - Eclipse: https://docs.nvidia.com/cuda/nsight-eclipse-plugins-guide/index.html
  - Emacs: https://docs.nvidia.com/cuda/cuda-gdb/index.html#gui-integration
- Initial setup is necessary, but probably simplifies your overall workflow.