CUDA DEBUGGING

Bob Crovella, 9/14/2021

AGENDA

CUDA Error Management

compute-sanitizer

cuda-gdb

Further Study

Homework

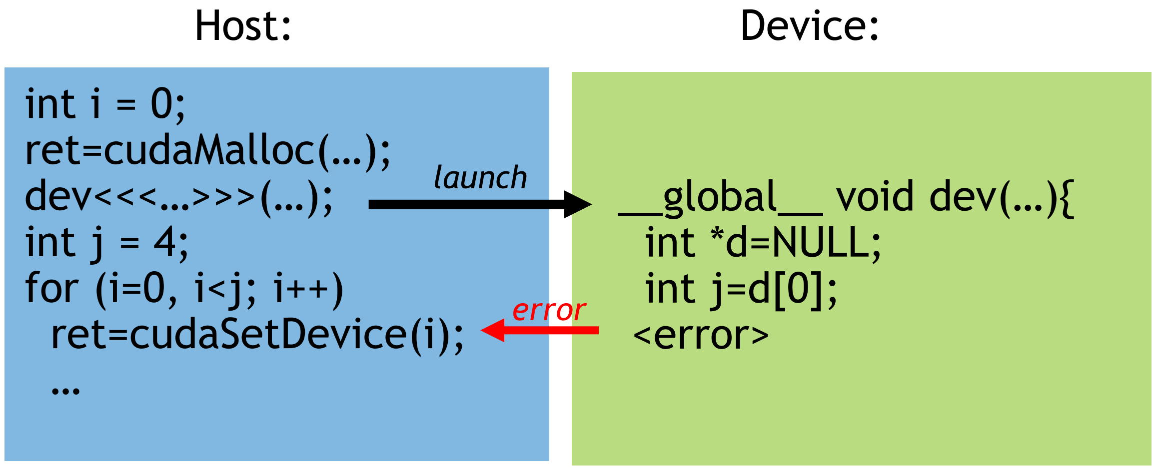
ERROR MANAGEMENT

BASIC CUDA ERROR CHECKING

- All CUDA runtime API calls return an error code.
    - CUDA runtime API: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html
    - Example: **cudaError_t cudaSetDevice (int device)**
    - **cudaError_t** is an enum type, with all possible error codes, examples:
        - **cudaSuccess** (no error)
        - **cudaErrorMemoryAllocation** (out of memory error)
- **cudaGetErrorString(cudaError_t err)** converts an error code to human-readable string
- Best practice is to always check these codes and handle appropriately. **Just do it!**
- The usual kernel launch syntax (**kernel_name<<<…>>>(…)**) is not a CUDA runtime API call and does not return an error code per-se

ASYNCHRONOUS ERRORS

- CUDA kernel launches are *asynchronous*
  - The kernel may not begin executing right away
  - The host thread that launches the kernel continues, without waiting for the kernel to complete
- It is possible for a CUDA error to be detected during kernel execution
- That error will be signalled at the *next* CUDA runtime API call, *after* the error is detected



KERNEL ERROR CHECKING

- CUDA kernel launches can produce two types of errors:
  - Synchronous: detectable right at launch
  - Asynchronous: occurs during device code execution
- Detect Synchronous errors right away with **cudaGetLastError()** or **cudaPeekAtLastError()**
- Asynchronous error checking involves tradeoffs
  - Can force immediate checking with a synchronizing call like **cudaDeviceSynchronize()** but this breaks asynchrony/concurrency structure
  - Optionally use a debug macro
  - Optionally set CUDA_LAUNCH_BLOCKING environment variable to 1

Kernel error checking example:

```
dev<<<…>>>(…);
ret = cudaGetLastError();
if (debug) ret = cudaDeviceSynchronize();
```

STICKY VS. NON-STICKY ERRORS

- A non-sticky error is recoverable
  - Example: ret = cudaMalloc(10000000000000000000000000000);   (out of memory error)
  - Such errors do not “corrupt the CUDA context”
  - Subsequent CUDA runtime API calls behave normally
- A sticky error is not recoverable
  - A sticky error is usually (only) resulting from a kernel code execution error
  - Examples: kernel time-out, illegal instruction, misaligned address, invalid address
  - CUDA runtime API is no longer usable in that process
  - All subsequent CUDA runtime API calls will return the same error
  - Only “recovery” process is to terminate the owning host process (i.e. end the application).
  - A multi-process application can be designed to allow recovery: https://stackoverflow.com/questions/56329377

EXAMPLES

- shared_mem_size=32768;
- k<<<1024, 1024, shared_mem_size*sizeof(double), stream>>>(…);
- cudaGetLastError() gets the last error *and clears it if it is not sticky*
- cudaPeekAtLastError() gets last error but does not clear it
- cudaMemcpy(dptr, hptr, size, cudaMemcpyDeviceToHost);
- ret = cudaMemcpy(dptr2, hptr2, size2, cudaMemcpyHostToDevice);

EXAMPLES

- Macro example - macro instead of function

```
#include <stdio.h>

#define cudaCheckErrors(msg) \
    do { \
        cudaError_t __err = cudaGetLastError(); \
        if (__err != cudaSuccess) { \
            fprintf(stderr, "Fatal error: %s (%s at %s:%d)\n", \
                msg, cudaGetErrorString(__err), \
                __FILE__, __LINE__); \
            fprintf(stderr, "*** FAILED - ABORTING\n"); \
            exit(1); \
        } \
    } while (0)
```

COMPUTE-SANITIZER TOOL

COMPUTE-SANITIZER

- A functional correctness checking tool, installed with CUDA toolkit
- Provides “automatic” runtime API error checking – even if your code doesn’t handle errors
- Can work with various language bindings: CUDA Fortran, CUDA C++, CUDA Python, etc.
- Sub-tools:
  - memcheck (default): detects illegal code activity: illegal instructions, illegal memory access, misaligned access, etc.
  - racecheck: detects shared memory race conditions/hazards: RAW, WAW, WAR
  - initcheck: detects accesses to global memory which has not been initialized
  - synccheck: detects illegal use of synchronization primitives (e.g. **__syncthreads()**)
- Many command line options to modify behavior:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html#command-line-options

MEMCHECK SUB-TOOL

- The “default” tool – its recommended to run this tool first, before using other tools
- Basic usage: **compute-sanitizer ./my_executable**
- Kernel execution errors:
  - Invalid/out-of-bounds memory access
  - Invalid PC/Invalid instruction
  - Misaligned address for data load/store
- Provides error localization when your code is compiled with **-lineinfo**
  - This is useful for other tools also, e.g. source-level work in the profilers (nsight compute)
- Has a performance impact on speed of kernel execution
- Can also do leak checking for device-side memory allocation/free
- Error checking is “tighter” than ordinary runtime error checking

MEMCHECK EXAMPLE

Out-of-bounds detection

```
$ cat t1866.cu
__global__ void k(char *d){
  d[43] = 0;
}
int main(){
  char *d;
  cudaMalloc(&d, 42);
  k<<<1,1>>>(d);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ ./t1866
$
```

```
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= Invalid __global__ write of size 1 bytes
=========     at 0x40 in
/home/user2/misc/t1866.cu:2:k(char*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7fe035a0002b is out of bounds
=========     Saved host backtrace …
=========     Host Frame:cuLaunchKernel
[0x7fe0685de728]
…
=========     Host Frame: [0x4034b1]
=========                in /home/user2/misc/./t1866
=========
========= Program hit unspecified launch failure
(error 719) on CUDA API call to cudaDeviceSynchronize.
…
========= ERROR SUMMARY: 2 errors
```

RACECHECK SUB-TOOL

- CUDA specifies no order of execution among threads
- Shared memory is commonly used for inter-thread communication
- In this scenario, ordering of reads and writes often matters for correctness
- Basic usage: **compute-sanitizer --tool racecheck ./my_executable**
- Finds shared memory (only) race conditions:
  - WAW – two writes to the same location that don't have intervening synchronization
  - RAW – a write, followed by a read to a particular location, without intervening synchronization
  - WAR – a read, followed by a write, without intervening synchronization
- Detailed reporting is available:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html#racecheck-report-modes

RACECHECK EXAMPLE

RAW hazard

```
$ cat t1866.cu
const int bs = 256;
__global__ void reverse(char *d){
  __shared__ char s[bs];
  s[threadIdx.x] = d[threadIdx.x];
  d[threadIdx.x] = s[bs-threadIdx.x-1];
}
int main(){
  char *d;
  cudaMalloc(&d, bs);
  reverse<<<1,bs>>>(d);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
$
```

```
$ compute-sanitizer --tool racecheck ./t1866
========= COMPUTE-SANITIZER
========= ERROR: Race reported between Write
access at 0x70 in
/home/user2/misc/t1866.cu:4:reverse(char*)
=========     and Read access at 0x80 in
/home/user2/misc/t1866.cu:5:reverse(char*) [256
hazards]
=========
========= RACECHECK SUMMARY: 1 hazard displayed (1
error, 0 warnings)
$
```

INITCHECK SUB-TOOL

Detects use of uninitialized device global memory

```
$ cat t1866.cu
const int bs = 1;
__global__ void k(char *in, char *out){
  out[threadIdx.x] = in[threadIdx.x];
}
int main(){
  char *d1, *d2;
  cudaMalloc(&d1, bs);
  cudaMalloc(&d2, bs);
  k<<<1,bs>>>(d1, d2);
  cudaDeviceSynchronize();
}
$ nvcc -o t1866 t1866.cu -lineinfo
$ compute-sanitizer ./t1866
========= COMPUTE-SANITIZER
========= ERROR SUMMARY: 0 errors
$
```

```
$ compute-sanitizer --tool initcheck ./t1866
========= COMPUTE-SANITIZER
========= Uninitialized __global__ memory read of
size 1 bytes
=========     at 0x50 in
/home/user2/misc/t1866.cu:3:k(char*,char*)
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x7fc543a00000
=========     Saved host backtrace up to driver
entry point at kernel launch time
=========     Host Frame:cuLaunchKernel
[0x7fc57546a728]
=========                 in /lib64/libcuda.so.1
…
========= ERROR SUMMARY: 1 error
$
```

SYNCCHECK SUB-TOOL

- Applies to usage of **__syncthreads()**, **__syncwarp()**, and CG equivalents (e.g. **this_group.sync()**)
- Typical usage is for detection of illegal use of synchronization, where not all necessary threads can reach the sync point:
  - Threadblock level
  - Warp level
- In addition, the __syncwarp() intrinsic can take a mask parameter, which specifies expected threads
  - Detects invalid usage of the mask
- Basic usage: **compute-sanitizer --tool synccheck ./my_executable**
- Applicability is limited on cc 7.0 and beyond due to volta execution model relaxed requirements
- Example:
  - https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html#synccheck-demo-illegal-syncwarp

DEBUGGING WITH CUDA-GDB

CUDA-GDB

- Based on widely-used **gdb** debugging tool (part of gnu toolchain). (This is not a tutorial on **gdb**)
- "command-line" debugger, allows for typical operations like:
  - setting breakpoints (e.g. **b**)
  - single-stepping (e.g. **s**)
  - inspection of data (e.g. **p**)
  - And others
- cuda-gdb uses the same command syntax where possible, and provides certain command extensions
- Generally, you want to build a debug code to use with the debugger
- The focus here will be on debugging device code. Assumption is you already know how to debug host code
- Supports debug of both CUDA C++ and CUDA Fortran applications

BUILDING DEBUG CODE

- Fundamentally, the compile command line for nvcc should include:
  - **-g** – standard gnu switch for building a debug (**host**) code
  - **-G** – builds debug **device** code
- This makes the necessary symbol information available to the debugger so that you can do “source-level” debugging.
- The –G switch has a substantial impact on device code generation. Use it for debug purposes only.
  - **Don’t do performance analysis on device code built with the -G switch**
  - The –G switch will often make your code run slower
  - In rare cases, the –G switch may change the behaviour of your code
- Make sure your code is compiled for the correct target: e.g. **-arch=sm_70**

ADDITIONAL PREP SUGGESTIONS

- If possible, make sure your code completes the various sanitizer tool tests
- If possible, make sure your host code is "sane" e.g. does not seg fault
- If possible, make sure your kernels are actually being launched, e.g:
  - nsys profile --stats=true ./my_executable (and check e.g. "CUDA Kernel Statistics"

CUDA SPECIFIC COMMANDS

- set cuda …   <used to set general options and advanced settings>
    - launch_blocking (on/off)   <make launches pause the host thread>
    - break_on_launch (option) <break on every new kernel launch>
- info cuda …   <get general information on system configuration>
    - devices, sms, warps, lanes, kernels, blocks, threads, …
- cuda … <used to inspect or set current focus>
    - (cuda-gdb) cuda device sm warp lane block thread   <display current focus coordinates>
    - block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0
    - (cuda-gdb) cuda thread (15) <change coordinate(s)>

DEMO

ADDITIONAL NOTES, TIPS, TRICKS

- synccheck tool may have limited usefulness due to Volta execution model – relaxed sync requirements
- CUDA Fortran debugging “print” commands not working correctly – expected to be fixed in a future tool chain
- Cannot inspect device memory (e.g. with “print”) unless stopped at a breakpoint in device code
- compute-sanitizer host backtrace will be improved in the future
- How to “look up” an error code (e.g. 719), two ways:
    - Search in …/cuda/include/driver_types.h
    - Docs: runtime API section 6.36, Data types

FURTHER STUDY

- CUDA error checking:
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#error-checking
  - https://stackoverflow.com/questions/14038589/what-is-the-canonical-way-to-check-for-errors-using-the-cuda-runtime-api
  - CUDA context: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context
- compute-sanitizer:
  - https://docs.nvidia.com/cuda/sanitizer-docs/ComputeSanitizer/index.html
- cuda-gdb:
  - https://docs.nvidia.com/cuda/cuda-gdb/index.html
- Simple gdb tutorial:
  - https://www.cs.cmu.edu/~gilpin/tutorial/

HOMEWORK

- Log into Summit (ssh username@home.ccs.ornl.gov -> ssh summit)
- Clone GitHub repository:
  - Git clone git@github.com:olcf/cuda-training-series.git
- Follow the instructions in the readme.md file:
  - https://github.com/olcf/cuda-training-series/blob/master/exercises/hw12/readme.md
- Prerequisites: basic linux skills, e.g. ls, cd, etc., knowledge of a text editor like vi/emacs, and some knowledge of C/C++ programming

BACKUP: BASIC GDB SYNTAX

BASIC GDB

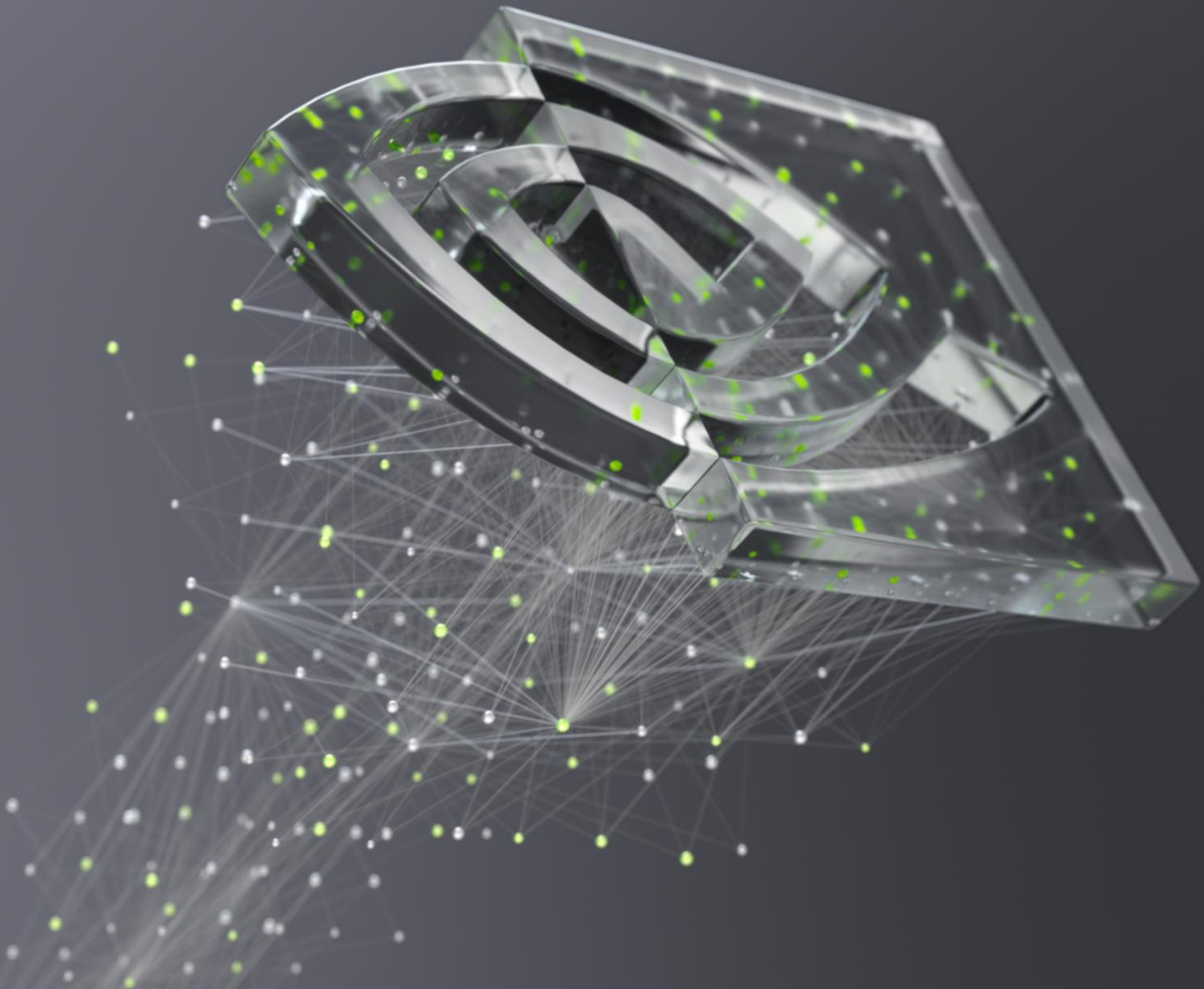
Getting started, setting a breakpoint, running, single-step, continuing

- Compile your code with –g (host debug) and –G (device debug)
- gdb ./my_executable
- Set a breakpoint: **b** command
  - if only one file: (gdb) b <line_number>
  - If multiple source files: (gdb) b <file_name:line_number>
- Run-from-start: **r** command
- Single step: **s** command ("step into")
- Step next: **n** command ("step over")
- Continue : **c** command

BASIC GDB

Inspecting data, clearing breakpoints, conditional breakpoints

- Print data: **p** command
  - symbolically: p s[0]
  - multiple values: p s[0]@8
- Removing breakpoints:
  - clear <file-name:line-number> (removes breakpoint based on location)
  - delete <breakpoint-number> (removes breakpoint based on id)
- Conditional breakpoints:
  - Set a breakpoint first
  - condition <breakpoint-id> <Boolean-test>
  - condition 1 i<32



NVIDIA