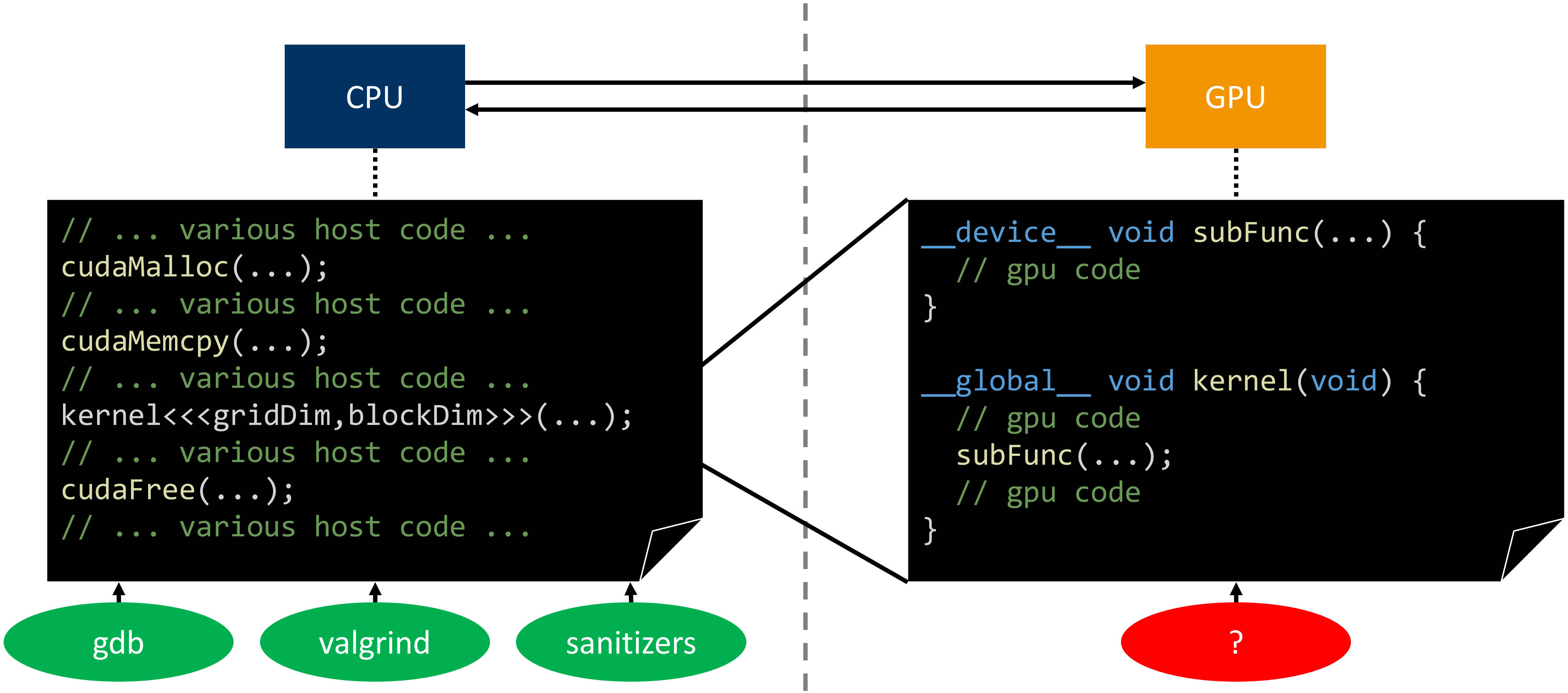
Using the CUDA Debugger
(content adapted from Dave Goodwin and Nvidia)

Philipp Gschwandtner

Why is GPU Debugging so Difficult?



Why is GPU Debugging so Difficult? cont'd

- GPUs…
  - Are physically and logically a separate device
  - Have a highly parallel, non-x86 hardware architecture
  - Have their own memory address space
  - Run their own CUDA threads with (partially) CPU-independent execution flow

- Standard debugging tools…
  - Do not have access to these devices
  - Are not equipped for dealing with these special properties

Debugging with CUDA

- Debugging is required
  - No reasonable code development without it
- Debugging tools supporting GPUs are available for CUDA
  - Part of why CUDAs was/is so successful compared to e.g. OpenCL
- Nvidia offers extensions of standard tools
  - Minimally-invasive approach
  - Improves user adoption compared to developing fully distinct tools

Selection of Available Tools

- cuda-gdb
  - Extension of gdb
- cuda-memcheck
  - Similar to valgrind
- Parallel Nsight
  - Graphical tool
  - Visual Studio / Eclipse integration

- Arm FORGE (Alinea DDT)
- TotalView
- Others

There are also Performance Debugging Tools

- Nsight (multiple variants)
  - Profilers for CUDA kernels, API calls and GPU hardware metrics
- Visual Profiler
  - GUI with “timeline” view
- CUPTI
  - CUDA Profiling Tools Interface
  - Enables hardware counter access for third-party tools
- PAPI
  - C library for reading hardware counters
- Score-P
  - CPU/GPU performance analysis tool
- Cube, Vampir, …
  - Performance reporting and visualization tools

Compilation Flags

- Add flags for debug information
  - -g for the CPU code
  - -G for the GPU code (turns off all optimizations, considerable slowdown!)
    - Alternative: -lineinfo for the GPU code (line numbers only), use when profiling

- Example:
  - `nvcc -g -G prog.cu -o prog`

`cuda-memcheck`

- Stand-alone run-time error checker tool
  - Stack overflows, out-of-bounds accesses, misaligned accesses, memory leaks, etc.
  - Similar to valgrind
  - Also offers racecheck, synccheck, and initcheck tools
  - https://docs.nvidia.com/cuda/cuda-memcheck/
- Does not require recompilation
  - But needs debug information for proper error location indication
- Not all error reports are precise
- Can be used from within `cuda-gdb`

Execution

- Part of CUDA installation
  - `cuda-memcheck prog_name`
- Also works with MPI
  - `mpiexec -n 8 xterm -e cuda-memcheck prog_name`
  - `mpiexec -n 1 cuda-memcheck prog_name : -n 7 ./prog_name`
  - `mpiexec -n 8 cuda-memcheck prog_name`

Example Output

```
==== Invalid __global__ write of size 4
====     at 0x00000010 in demo.cu:8:out_of_bounds_kernel(void)
====     by thread (0,0,0) in block (0,0,0)
====     Address 0xffffffff87654320 is out of bounds
====     Saved host backtrace up to driver entry point at kernel launch time
====     Host Frame:/usr/local/lib/libcuda.so (cuLaunchKernel + 0x3ae) [0xddbee]
====     Host Frame:/usr/local/lib/libcudart.so.5.0 [0xcd27]
====     Host Frame:/usr/local/lib/libcudart.so.5.0 (cudaLaunch + 0x1bb) [0x3778b]
====     Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1eb1d]
......... snip .........
====     Host Frame:memcheck_demo [0x9b9]
```

__global__ device memory
write operation
4 bytes (SP float, integer, etc.)

program counter address
source file, line no., kernel name

thread and block indices (x,y,z)

memory address
type of error

Synchronization Checking

- `cuda-memcheck` offers a `synccheck` tool
  - Can identify incorrect use of synchronization primitives such as `__syncthreads()`
  - Needs to be enabled with `--tool synccheck`
- Does NOT check for memory errors
  - When debugging, first run `memcheck`
  - Afterwards, run `synccheck` if required

Example Output

always “barrier error”

divergent threads
within same block

```
==== Barrier error detected. Divergent thread(s) in block
====     at 0x00000130 in divergence.cu:61:threadDivergence(int*)
====     by thread (37,0,0) in block (0,0,0)
```

program counter address
source file, line no., kernel name

thread and block indices (x,y,z)

`printf()` Debugging

- Yes, CUDA allows `printf()` to be used inside GPU code
  - Arguments and format specifiers (%d, `%.5f`, …) just like C-library-`printf()`
  - Returns number of arguments parsed (not number of characters printed)
- Behaves like any other device function
  - Executed by every (!) thread
  - in the current context

```
__global__ void mallocTest() {
  printf("Thread %d\n", threadIdx.x);
}

// Output:
// Thread 0
// Thread 1
// Thread 2
// ...
```

cuda-gdb

- Built around GDB
  - All standard GDB debugging features (set breakpoints, inspect memory/variables/registers, …)
  - Allows debugging both CPU and GPU code
  - Supports multiple GPUs, contexts, kernels
  - https://docs.nvidia.com/cuda/cuda-gdb/

- Graphical wrappers available (e.g. GNU DDD, Emacs)
  - We'll focus on the command line though

- Careful on PCs: breakpoints can freeze the GPU (and output of connected screens)!
  - No issue when remotely debugging via ssh, when using two GPUs, or with compute capability >= 6.0
  - Mitigated when enabling software preemption (beta, compute capability ≥ 3.5)

Execution

- Part of CUDA installation
  - `cuda-gdb prog_name`
- Also works with MPI
  - `mpiexec -n 8 xterm -e cuda-gdb prog_name`
  - `mpiexec -n 1 cuda-gdb prog_name : -n 7 ./prog_name`
  - `mpiexec -n 8 cuda-gdb --batch --command=script.txt prog_name`

`cuda-gdb`: Execution control

- Launch application
  - `(cuda-gdb) run`
- Resume after any halt
  - `(cuda-gdb) continue`
- Kill application
  - `(cuda-gdb) kill`
- Interrupt application
  - `CTRL+C`
- Set breakpoint in line 7
  - `(cuda-gdb) break prog.cu:7`
- Where in my program are we?
  - `(cuda-gdb) backtrace`
- Run step-by-step (over function calls)
  - `(cuda-gdb) next`
- Run step-by-step (into function calls)
  - `(cuda-gdb) step`

cuda-gdb: Inspecting Data

- Print content of a variable
  - `(cuda-gdb) print variable_name`

- Print address of a variable
  - `(cuda-gdb) print &variable_name`

- Print content of a pointer
  - `(cuda-gdb) print *pointer_name`

- Print consecutive elements of array
  - `print array_name[3] @ 4`

`cuda-gdb`: Dealing with Threads

- List and switch CPU threads
  - `info threads`
  - `thread 3`

- List and switch CUDA threads
  - `info cuda threads`
  - `cuda thread (20,0,0)`
  - `cuda kernel 0 grid 1 block (0,0,0) thread (20,0,0)`

`cuda-gdb`: Misc

- List all devices and device in focus
  - `(cuda-gdb) info cuda devices`
- List all running kernels
  - `(cuda-gdb) info cuda kernels`
- Change data while debugging
  - `(cuda-gdb) print my_variable = 5`
  - `(cuda-gdb) print $R3 = 5`

Usual `(cuda-)gdb` Workflow

1. Set a breakpoint or enable a tool
   - `(cuda-gdb) break my_program.cu:27`
   - `(cuda-gdb) set cuda memcheck on`
2. `(cuda-gdb) run`
3. `(cuda-gdb) backtrace`
4. Inspect current state, e.g.
   - `(cuda-gdb) print $variable`
5. Figure out what went wrong

Best Practice

- 1. Determine type and scope of bug
  - Incorrect result
  - Failure to launch
  - Crash
  - Hang
  - Slow execution
    (➔ performance debugging)
- 2. Try to reproduce with debug build
  - Re-compile with -g -G and re-run
- 3. Try to create a minimum working example (MWE)
  - Problem size, involved components, etc.
- 4. Investigate and fix the bug
  - Try `cuda-memcheck` alone (fast)
  - `cuda-gdb` if needed (more information but slower)
  - `printf`-debugging also possible

Tips

- Try to maximize reproducibility
  - Fix input data
  - Fix seeds of random number generators
  - Etc.

- Increase determinism by launching kernels synchronously
  - CUDA_LAUNCH_BLOCKING=1

- Limit available devices
  - CUDA_VISIBLE_DEVICES=0,1

Conclusion

- Debugging parallel programs is difficult
  - Debugging on GPUs even more so
- CUDA offers some handy tools for the job
  - Most notably `cuda-gdb` and `cuda-memcheck`
- Heed coding guidelines and best practice
  - Takes effort in the beginning
  - Large pay-off down the road

`cuda-memcheck`: Practical Exercise 1

- Compile `day_2/debugging/vector.cu` with debugging symbols
- Run with `cuda-memcheck`
- Interpret the results!
  - What is the problem?
  - How can we fix it?

`cuda-memcheck`: Practical Exercise 1 Solution

- We work on an array of size 256 but we are starting 265 threads!

- 2 possible solutions
  - correct 265 to 256, or
  - suspend threads with ID ≥ ARRAYDIM

```
__global__ void KrnlDmmy(int *x) {
    int i;
    i = (blockIdx.x*blockDim.x) + threadIdx.x;

    if(i >= ARRAYDIM) { return; }

    x[i] = i;
    return;
}

// OR

thrds_per_block.x = 256;
```

`cuda-memcheck --tool syncheck`: Practical Exercise 2

- Compile `day_2/ho1/synchthreads.cu` with debugging symbols
- Run with `cuda-memcheck --tool synccheck`
- Check the results!
- Solution? See morning lecture!

`cuda-gdb`: Practical Exercise 3

- Compile `day_2/debugging/stencil.cu` with debugging symbols
- Run in gdb with
  - `(cuda-gdb) set cuda memcheck on`
- Check the results!
  - What is the problem?
  - How can we fix it?

cuda-gdb: Practical Exercise 3 Solution

- We have two size variables, one for the size of the domain (`size`) and one for the size of our stencil (`wsize`).
- We allocate all three buffers of size `wsize`
  - Should be d_weights only
  - d_in and d_out should be `size` long instead
- We also copy all buffers with `size`!

```
int size = N * sizeof(float);
int wsize = (2 * RADIUS + 1) * sizeof(float);

cudaMalloc(&d_weights, wsize);
cudaMalloc(&d_in, wsize); // Incorrect!
cudaMalloc(&d_out, wsize); // Incorrect!
```