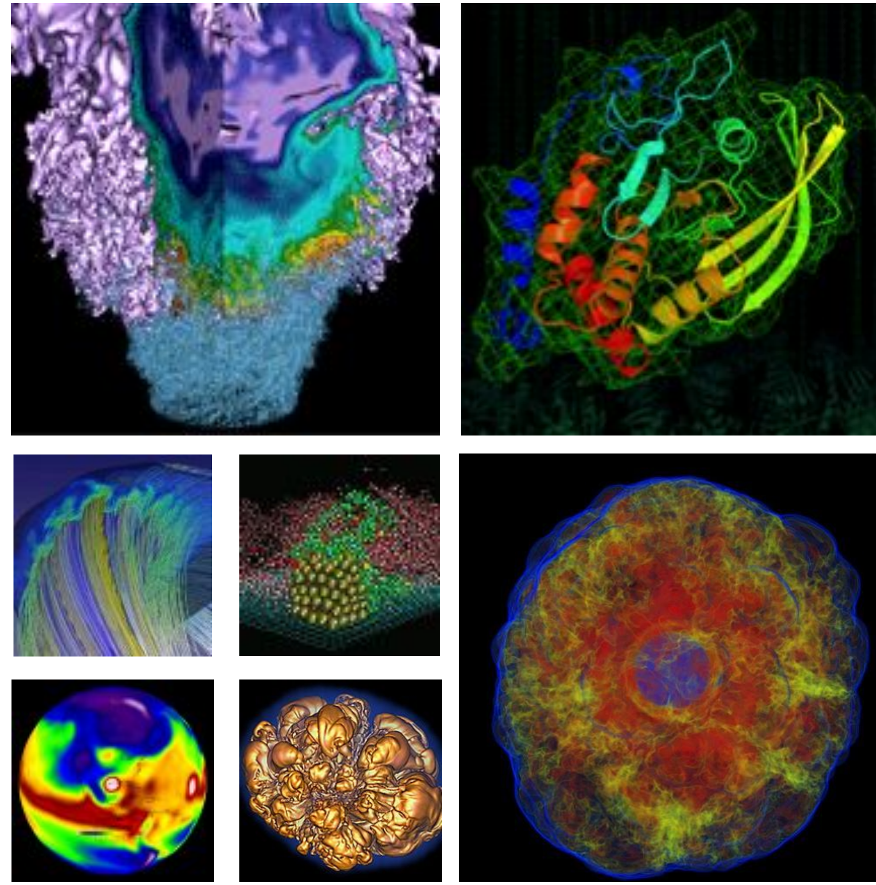
Debugging on GPU

Introduction to GPU

February, 2020



Woo-Sun Yang
NERSC User Engagement Group

February 28, 2020

Debugging on GPUs

- **Hundreds or thousands of threads**
  - Difficult to maintain who is doing what or where an error occurs with which thread
  - Using print statements becomes out of the question
  - Use debugging tools!

- **Tools that covered today:**
  - CUDA-GDB
  - CUDA-MEMCHECK
  - TotalView

CUDA-GDB

- **Extension of GNU GDB for debugging CUDA codes**
  - Debugging both GPU and CPU code within the same app
  - Command-line mode
  - For non-MPI (OK, see the manual for a trick for running a small MPI app)
  - Add 'cuda' for commands for CUDA, as in '`cuda thread 170`'
- **User's manual:**
  - CUDA-GDB CUDA Debugger: $CUDA_ROOT/doc/pdf/cuda-gdb.pdf
- **Useful notes**
  - Set Breakpoints
    - Watchpoints on CUDA code not supported
  - Control code execution ('`run`', '`continue`')
  - Print code status or variable values
  - Can run CUDA-MEMCHECK's memcheck tool under CUDA-GDB
  - Autostep
    - Specify a range of source code lines for single stepping on GPU (slow)
    - Quickly identify the warp responsible for causing an error
  - To enable a GPU coredump when GPU exception happens
    - $ export `CUDA_ENABLE_COREDUMP_ON_EXCEPTION=1`

CUDA-GDB (cont'd)

- **Compile: inside an interactive batch session**

```
$ module load cuda
$ nvcc -g -G -o foo foo.cu
```

```
$ module load pgi
$ pgfortran -g -Mcuda=nordc -o foo foo.cuf
```

- **Run: inside an interactive batch session**

```
$ module load cuda
$ srun --pty cuda-gdb ./a.out  # --pty to run commands interactively
```

Kernel focus

- **We are dealing with hundreds/thousands of threads on GPUs, but one thread in focus**
- **To examine the program execution associated with a particular thread, need to change focus to that thread**
- **Can do that using one of the 2 coordinates**
  - Hardware coordinates
    - Device, SM (Streaming Multiprocessor), warp, and lane
  - Software coordinates
  - Kernel, grid, block, and thread
    - If one is changed, the other is changed automatically
- **Example**

```
(cuda-gdb) cuda device sm warp lane             # display HW coords
device 0, sm 0, warp 0, lane 0
(cuda-gdb) cuda kernel block thread             # display SW coords
kernel 1, block (0,0,0), thread (0,0,0)
(cuda-gdb) cuda device 0 sm 1 warp 2 lane 3    # switch focus
[Switching focus to CUDA kernel 1, grid 2, block (8,0,0), thread (67,0,0), device 0, sm 1,
warp 2, lane 3]
374 int totalThreads = gridDim.x * blockDim.x
```

Debugging examples

- **From Chapter 11.1 'Example: bitreverse' in the manual**

```
$ module load cuda
$ nvcc -g -G -o bitreverse bitreverse.cu
$ srun --pty cuda-gdb ./bitreverse
...
(cuda-gdb) break main              # create a breakpoint at main
(cuda-gdb) break bitreverse        # create a breakpoint at bitreverse (kernel)
(cuda-gdb) break 21                # create a breakpoint at line 21
(cuda-gdb) run
...
Breakpoint 1, main () at bitreverse.cu:25
25 void *d = NULL; int i;
(cuda-gdb) continue
...
Thread 1 "bitreverse" hit Breakpoint 2, bitreverse<<<(1,1,1),(256,1,1)>>> (
data=0x2aaae1a00000) at bitreverse.cu:12
12 array[threadIdx.x] = idata[threadIdx.x];
(cuda-gdb) info cuda threads
  BlockIdx ThreadIdx To BlockIdx ThreadIdx Count         Virtual PC      Filename Line
Kernel 0
*  (0,0,0)    (0,0,0)     (0,0,0) (255,0,0)    256 0x0000000000dae7a0 bitreverse.cu   12
(cuda-gdb) backtrace
#0  bitreverse<<<(1,1,1),(256,1,1)>>> (data=0x2aaae1a00000) at bitreverse.cu:12
```

Debugging examples (cont'd)

```
(cuda-gdb) info cuda kernels
  Kernel Parent Dev Grid Status   SMs Mask GridDim  BlockDim Invocation
*      0      -   0    1 Active 0x00000001 (1,1,1) (256,1,1) bitreverse(data=0x2aaae1a00000)
(cuda-gdb) print blockIdx
$1 = {x = 0, y = 0, z = 0}
(cuda-gdb) print gridDim
$2 = {x = 1, y = 1, z = 1}
(cuda-gdb) next                               # do this 4 times
14  array[threadIdx.x] = ((0xf0f0f0f0 & array[threadIdx.x]) >> 4) |
...
(cuda-gdb) print array[0]@12                  # check 12 elements
$3 = {0, 128, 64, 192, 32, 160, 96, 224, 16, 144, 80, 208}
(cuda-gdb) print &data                        # parameter of the kernel
$4 = (@generic void * @parameter *) 0x160
(cuda-gdb) print *(@generic void * @parameter *) 0x160
$5 = (@generic void * @parameter) 0x2aaae1a00000
(cuda-gdb) cuda thread 170                    # switch focus to thread 170
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (170,0,0), device 0, sm 0,
warp 5, lane 10]
12 array[threadIdx.x] = idata[threadIdx.x];
(cuda-gdb) ...DO SOMETHING...

(cuda-gdb) quit
```

Debugging examples (cont'd)

- The 'autostep' example (11.2 Example: autostep) code doesn't work as described in the manual
- But it clearly demonstrates usefulness of the functionality in GPU debugging

CUDA-MEMCHECK

- **A suite of tools**
  - Memcheck: detects memory errors, etc.
  - Racecheck: detects race conditions
  - Initcheck: detects use of uninitialized variables
  - Synccheck: detects sync errors
- **User's manual**
  - CUDA-MEMCHECK
    - $CUDA_ROOT/doc/pdf/CUDA_Memcheck.pdf
- **Compile**
  - `-G`: To generate debug info for CUDA app
  - `-lineinfo`: Generate line number information
  - `-rdynamic`: The host compiler retains function symbols
  - `-Xcompiler`: To specify flags to the host compiler

```
$ module load cuda
$ nvcc -g -G foo.cu -o foo
$ nvcc -Xcompiler -rdynamic -lineinfo -o foo foo.cu
```

memcheck

- **Detect**
  - Memory access error
    - malloc/free error, double free, invalid pointer to free, heap corruption
  - Hardware exception
  - Leak detection
    - Detect memory leaks (allocated but never deallocated)
  - Device side allocation checking
    - malloc inside a kernel
    - Can be disabled by '--check-device-heap no'
  - CUDA API error checks
- **To run**

```
$ module load cuda
$ srun cuda-memcheck [memcheck_options] ./a.out
```

  - For detecting memory leaks: add '**`--leak-check=full`**'
- **Can be run under CUDA-GDB**

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

  - In this case, kernel launches become synchronous (that is, blocking)
- **See 10.1 'Example Use of Memcheck' & 10.2 'Integrated CUDA-MEMCHECK Example' in the manual**

racecheck

- **Detects race conditions**
  - Currently for shared memory only
- **To run**

```
$ srun cuda-memcheck --tool racecheck [options] ./a.out
```

- **Types of reporting**
  - Hazard reports
    - Detailed info on one particular hazard
    - **`--racecheck-report hazard`**
  - Analysis reports
    - Overall analysis over multiple hazard reports
    - **`--racecheck-report analysis`** (default)
  - All
    - **`--racecheck-report all`**
- **See 10.3 ‘Example Use of Racecheck’ in the manual**

initcheck

- **Detects use of uninitialized variables**
  - For variables in global memory memory only
- **To run**

```
$ srun cuda-memcheck --tool initcheck [options] ./a.out
```

- **See 10.4 'Example Use of Initcheck' in the manual**

synccheck

- **Detects synchronization error conditions**
  - Divergent thread(s) in block
  - Divergent thread(s) in warp
  - Invalid arguments: incorrect mask parameter for a sync statement
- **To run**

```
$ srun cuda-memcheck --tool synccheck [options] ./a.out
```

- **See 10.5 ‘Example Use of Synccheck’ in the manual**

TotalView

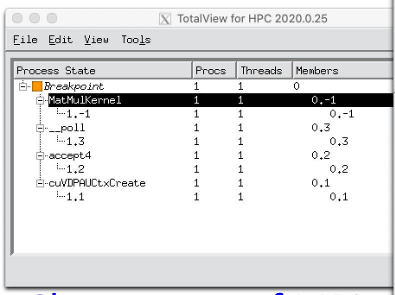
- **Graphical parallel debugger**
- **Supports**
  - OpenMP for GPU: Cray, GCC, Clang, IBM (more?)
  - OpenACC for GPU: Cray (and others?)
  - CUDA
  - MPI
- **To run**

```
$ module load totalview
$ totalview ./a.out
```

TotalView windows

GPU focus thread selector (Logical or Physical)

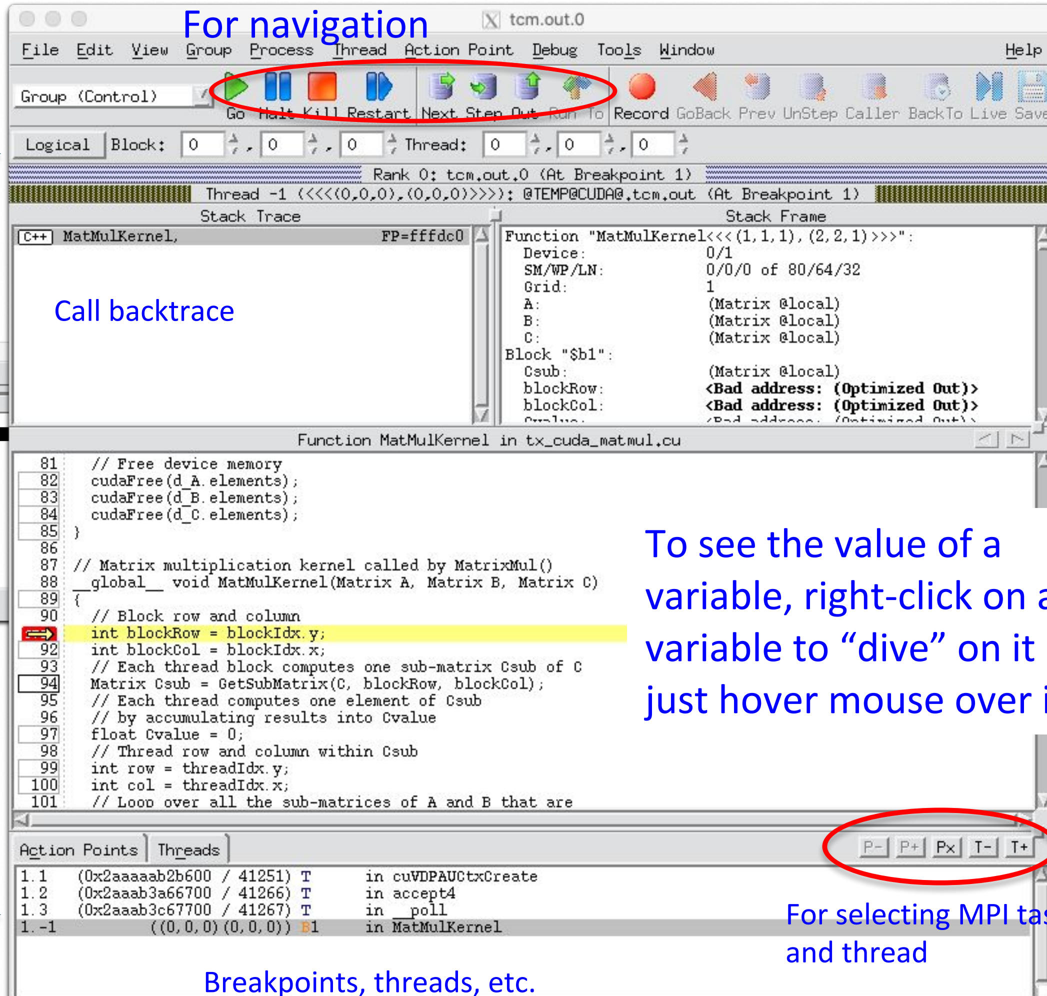
Root window



Shows state of MPI tasks and threads

A CPU thread assigned a positive thread ID; A CUDA thread assigned a negative ID

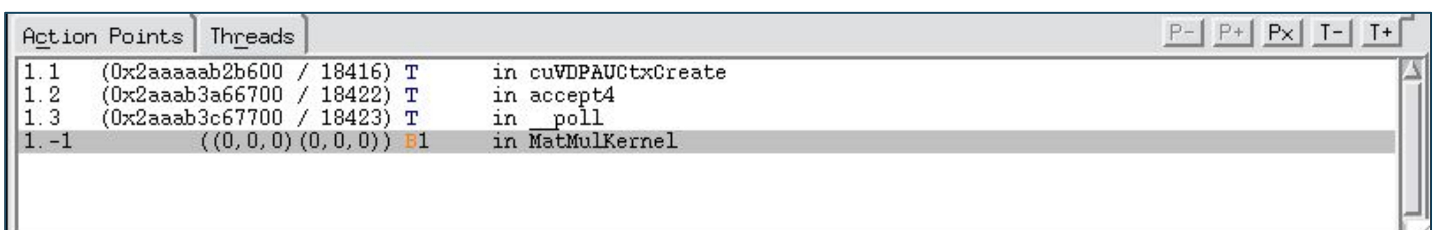
Process window



GPU focus thread ID (negative) and its coords

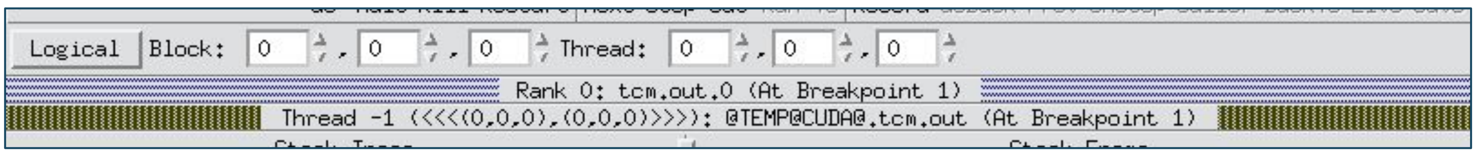
TotalView (cont'd)

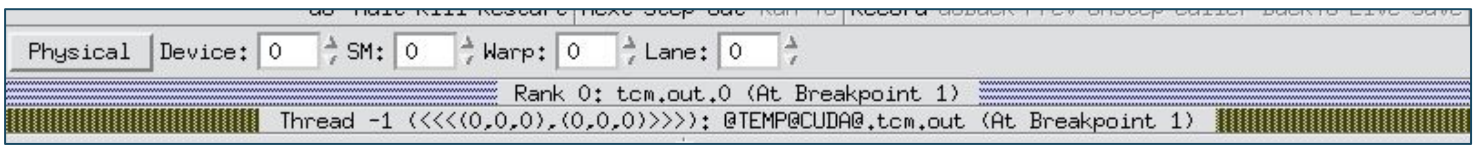
- **Unified source pane and breakpoint display**
  - Click on the number in the source pane to set a breakpoint there
  - A breakpoint set in CUDA code slides to the next host line in the source file
  - Once CUDA code is loaded, TotalView plants a breakpoint at the proper location in the CUDA code
- **Debugger thread ID**
  - A host (CPU) thread is assigned a positive ID
  - A CUDA thread is assigned a negative ID



TotalView (cont'd)

- **A single-step operation in CUDA code steps the entire warp associated with the GPU focus thread**
- **Can use either logical coords ("SW coords") and physical cords ("HW coords")**





- **"Dive" on a variable to display its value**
- **Plot array elements**
- **Get stats for array elements**

NERSC

National Energy Research Scientific Computing Center

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB
Lawrence Berkeley National Laboratory