Debugging Experience with CUDA-GDB and CUDA-MEMCHECK

Geoff Gerfin
Vyas Venkataraman
CUDA Debugging Solutions

CUDA-GDB
(Linux & Mac)

CUDA-Memcheck
(Linux, Mac, & Windows)

NVIDIA® Parallel Nsight™
Eclipse Edition (NEW!)
Visual Studio Edition

Allinea DDT

Rogue Wave TotalView
CUDA-GDB Overview

- What is it? What does it let you do?
  - Source and Assembly (SASS) Level Debugger
  - Simultaneous CPU and GPU debugging
    - Set Breakpoints and Conditional Breakpoints
    - Dump stack frames for thousands of CUDA threads
    - Inspect memory, registers, local/shared/global variables
  - Runtime Error Detection (stack overflow, ...)
    - Can’t figure out why your kernel launch is failing? Run cuda-gdb!
    - Integrated cuda-memcheck support for increased precision
  - Supports multiple GPUs, multiple contexts, multiple kernels
CUDA-GDB Overview

- Which hardware does it support?
  - All CUDA-capable GPUs SM1.1 and beyond
  - Compatible with NVIDIA Optimus laptops

- Which platforms does it support?
  - All CUDA-supported Linux distributions
  - Mac OS X
  - 32-bit and 64-bit platforms
Nsight Eclipse Edition
Debug View is powered by cuda-gdb

- Visualize device state
- Edit/Build/Debug/Profile
- Supported on Linux/Mac

Live demo Wed. @ 9am!
S0420 - Room A5
CUDA 101: Threads, Blocks, Grids

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
CUDA 101: Synchronization

1. First set of threads arrive
2. Second set of threads arrive
3. All threads resume

- `__syncthreads()` enforces synchronization within a block
  - Threads wait until all other threads in the same block have arrived
Execution Control

- Execution Control is identical to host debugging:
  - launch the application
    - (cuda-gdb) run
  - resume the application (all host threads and device threads)
    - (cuda-gdb) continue
  - kill the application
    - (cuda-gdb) kill
  - interrupt the application: CTRL-C
Execution Control

- Single-Stepping

<table>
<thead>
<tr>
<th>Single-Stepping</th>
<th>At the source level</th>
<th>At the assembly level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Over function calls</td>
<td>next</td>
<td>nexti</td>
</tr>
<tr>
<td>Into function calls</td>
<td>step</td>
<td>stepi</td>
</tr>
</tbody>
</table>

- Behavior varies when stepping `__syncthreads()`

<table>
<thead>
<tr>
<th>PC at a barrier?</th>
<th>Single-stepping applies to</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>All threads in the current block.</td>
<td>Required to step over the barrier.</td>
</tr>
<tr>
<td>No</td>
<td>Active threads in the current warp.</td>
<td></td>
</tr>
</tbody>
</table>
Breakpoints

- By name
  
  ```
  (cuda-gdb) break my_kernel
  (cuda-gdb) break _Z6kernel1IfiEvPT_PT0
  ```

- By file name and line number
  
  ```
  (cuda-gdb) break acos.cu:380
  ```

- By address
  
  ```
  (cuda-gdb) break *0x3e840a8
  (cuda-gdb) break *$pc
  ```

- At every kernel launch
  
  ```
  (cuda-gdb) set cuda break_on_launch application
  ```
Conditional Breakpoints

- Only reports hit breakpoint if condition is met
  - All breakpoints are still hit
  - Condition is evaluated every time for all the threads

- Condition
  - C/C++ syntax
  - supports built-in variables (blockIdx, threadIdx, …)
Thread Focus

- Some commands apply only to the thread in focus
  - Print local or shared variables
  - Print registers
  - Print stack contents

- Components
  - Kernel : unique, assigned at kernel launch time
  - Block : the application blockIdx
  - Thread : the application threadIdx
Thread Focus

- To switch focus to any currently running thread

```
(cuda-gdb) cuda kernel 2 block 1,0,0 thread 3,0,0
[Switching focus to CUDA kernel 2 block (1,0,0), thread (3,0,0)]

(cuda-gdb) cuda kernel 2 block 2 thread 4
[Switching focus to CUDA kernel 2 block (2,0,0), thread (4,0,0)]

(cuda-gdb) cuda thread 5
[Switching focus to CUDA kernel 2 block (2,0,0), thread (5,0,0)]
```
Thread Focus

- To obtain the current focus:

  (cuda-gdb) cuda kernel block thread
  kernel 2 block (2,0,0), thread (5,0,0)

  (cuda-gdb) cuda thread
  thread (5,0,0)
Devices

- To obtain the list of devices in the system:

```
(cuda-gdb) info cuda devices

<table>
<thead>
<tr>
<th>Dev</th>
<th>Desc</th>
<th>Type</th>
<th>SMs</th>
<th>Wps/SM</th>
<th>Lns/Wp</th>
<th>Regs/Ln</th>
<th>Active SMs Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0</td>
<td>gf100</td>
<td>14</td>
<td>48</td>
<td>32</td>
<td>64</td>
<td>0xfff</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>gt200</td>
<td>30</td>
<td>32</td>
<td>32</td>
<td>128</td>
<td>0x0</td>
</tr>
</tbody>
</table>
```

- The * indicates the device of the kernel currently in focus
Kernels

- To obtain the list of running kernels:

  ```
  (cuda-gdb) info cuda kernels
  
  Kernel  Dev  Grid  SMs  Mask  GridDim  BlockDim  Name  Args
  *  1   0  2  0x3fff (240,1,1) (128,1,1) acos  parms=...
  2   0  3  0x4000 (240,1,1) (128,1,1) asin  parms=...
  ```

- The * indicates the kernel currently in focus
Threads

- To obtain the list of running threads for kernel 2:

```
(cuda-gdb) info cuda threads kernel 2
```

<table>
<thead>
<tr>
<th>Block</th>
<th>Thread</th>
<th>To</th>
<th>Block</th>
<th>Thread</th>
<th>Cnt</th>
<th>PC</th>
<th>Filename</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0,0)</td>
<td>(0,0,0)</td>
<td>(3,0,0)</td>
<td>(7,0,0)</td>
<td>32</td>
<td>0x7fae70</td>
<td>acos.cu</td>
<td>380</td>
<td></td>
</tr>
<tr>
<td>(4,0,0)</td>
<td>(0,0,0)</td>
<td>(7,0,0)</td>
<td>(7,0,0)</td>
<td>32</td>
<td>0x7fae60</td>
<td>acos.cu</td>
<td>377</td>
<td></td>
</tr>
</tbody>
</table>

- Threads are displayed in (block,thread) ranges
- Divergent threads are in separate ranges
- The * indicates the range where the thread in focus resides
Stack Trace

- Applies to the thread in focus

(cuda-gdb) info stack
#0  fibo_aux (n=6) at fibo.cu:88
#1  0x7bbda0 in fibo_aux (n=7) at fibo.cu:90
#2  0x7bbda0 in fibo_aux (n=8) at fibo.cu:90
#3  0x7bbda0 in fibo_aux (n=9) at fibo.cu:90
#4  0x7bbda0 in fibo_aux (n=10) at fibo.cu:90
#5  0x7cfdb8 in fibo_main<<<(1,1,1),(1,1,1)>>>(...) at fibo.cu:95
Accessing variables and memory

- Read a source variable

  ```
  (cuda-gdb) print my_variable
  $1 = 3
  (cuda-gdb) print &my_variable
  $2 = (@global int *) 0x200200020
  ```

- Write a source variable

  ```
  (cuda-gdb) print my_variable = 5
  $3 = 5
  ```

- Access any GPU memory segment using storage specifiers
  - @global, @shared, @local, @generic, @texture, @parameter
Hardware Registers

- CUDA Registers
  - virtual PC: $pc (read-only)
  - SASS registers: $R0, $R1,…

- Show a list of registers (blank for all)

```
(cuda-gdb) info registers R0 R1 R4
R0      0x6    6
R1      0xffffffff 16776296
R4      0x6    6
```

- Modify one register

```
(cuda-gdb) print $R3 = 3
```
(cuda-gdb) x/10i $pc

0x123830a8 <__Z9my_kernel10params+8>: MOV R0, c [0x0] [0x8]
0x123830b0 <__Z9my_kernel10params+16>: MOV R2, c [0x0] [0x14]
0x123830b8 <__Z9my_kernel10params+24>: IMUL.U32.U32 R0, R0, R2
0x123830c0 <__Z9my_kernel10params+32>: MOV R2, R0
0x123830c8 <__Z9my_kernel10params+40>: S2R R0, SR_CTAid_X
0x123830d0 <__Z9my_kernel10params+48>: MOV R0, R0
0x123830d8 <__Z9my_kernel10params+56>: MOV R3, c [0x0] [0x8]
0x123830e0 <__Z9my_kernel10params+64>: IMUL.U32.U32 R0, R0, R3
0x123830e8 <__Z9my_kernel10params+72>: MOV R0, R0
0x123830f0 <__Z9my_kernel10params+80>: MOV R0, R0
CUDA-GDB 5.0 Features

- Attach to a running CUDA process (SM 2.0 and beyond)
- Attach upon GPU exceptions (SM 2.0 and beyond)
- Separate Compilation Support (SM 2.0 and beyond)
- Inlined Subroutine Debugging (SM 2.0 and beyond)
- CUDA API error reporting
- Enhanced interoperation with cuda-memcheck
CUDA-GDB 5.0 Features - Attach

Attach at any point in time!

CUDA GDB

CPU threads
GPU kernels, blocks, threads
CPU + GPU memory state
CPU + GPU register state

CPU + GPU

CUDA GDB
CUDA-GDB 5.0 Features - Attach

- Run your program at full speed, then attach with cuda-gdb
- No environment variables required!
- Inspect CPU and GPU state at any point in time
  - List all resident CUDA kernels
  - Utilize all existing CUDA-GDB commands
- Attach to CUDA programs forked by your application
- Detach and resume CPU and GPU execution
Attaching to a running CUDA process

1. Run your program, as usual

   $ myCudaApplication

2. Attach with cuda-gdb, and see what’s going on

   $ cuda-gdb myCudaApplication PID

   Program received signal SIGTRAP, Trace/breakpoint trap.
   [Switching focus to CUDA kernel 0, grid 2, block (0,0,0), thread (0,0,0),
   device 0, sm 11, warp 1, lane 0]

   0xae6688 in acos_main<<<(240,1,1),(128,1,1)>>> (parms=...) at acos.cu:383
   383     while (!flag);
   (cuda-gdb) p flag
   $1 = 0
Attaching on GPU Exceptions

1. Run your program, asking the GPU to wait on exceptions

   ```
   $ CUDA_DEVICE_Waits_ON_EXCEPTION=1 myCudaApplication
   ```

2. Upon hitting a fault, the following message is printed

   ```
   The application encountered a device error and CUDA_DEVICE_Waits_ON_EXCEPTION is set. You can now attach a debugger to the application for inspection.
   ```

3. Attach with `cuda-gdb`, and see which kernel faulted

   ```
   $ cuda-gdb myCudaApplication PID
   Program received signal CUDA_EXCEPTION_10, Device Illegal Address.
   (cuda-gdb) info cuda kernels
   ```

   ```
   Kernel Dev Grid SMs Mask GridDim BlockDim Name Args
   • 0 0 1 0x00000800 (1,1,1) (1,1,1) exception_kernel data=...
   ```
CUDA-GDB 5.0 Features - Error Reporting

- CUDA API error reporting (three modes)
  1. Trace all CUDA APIs that return an error code (default)
  2. Stop in the debugger when any CUDA API fails
  3. Hide all CUDA API errors (do not print them)

- Enhanced interoperation with cuda-memcheck
  - Display faulting address and memory segment

```bash
warning: CUDA API error detected: cudaMalloc returned (0xb)

(cuda-gdb) set cuda api failures [ignore | stop | hide]

Memcheck detected an illegal access to address (@global)0x500200028
```
CUDA-MEMCHECK
What is CUDA-MEMCHECK?

- “Why did my kernel fail?”
- Lightweight tool
- Run time error checker
  - Precise errors: Memory access
  - Imprecise errors: Hardware reported (SM 2.0+)
- Cross platform: Linux, Mac, Windows
- Integrated into cuda-gdb (Linux / Mac Only)
Running CUDA-MEMCHECK

- Standalone

```bash
$ cuda-memcheck [options] <my_app> <my_app_options>
```

- Misaligned and Out of bound access in global memory

```
Invalid __global__ read of size 4
at 0x000000b8 in basic.cu:27:kernel2
by thread (5,0,0) in block (3,0,0)
Address 0x05500015 is misaligned
```
Running CUDA-MEMCHECK

- Imprecise errors
- Out-of-range Shared or Local Address
  at 0x00000798 in kernel1
  by thread (0,0,0) in block (0,0,0)
- Multiple precise errors using continue mode
- Leak checking of cudaMalloc() allocations
  - Allocation that has not been cudaFree()’d at context destroy
- Integrated mode in CUDA-GDB

(cuda-gdb) set cuda memcheck on
New features in 5.0

- Shared memory hazard detection (racecheck)
- Improved precise detection in address spaces
- Device side malloc()/free() error checking
- Device heap allocation leak checking
- Stack back traces
- CUDA API error checking
- Better reporting inside cuda-gdb
- Improved precision for device heap checks
- Name demangling (with parameters) for kernels
Threads revisited

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Memory hierarchy

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory

- All blocks:
  - Global memory
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory
Memory hierarchy

- **Thread:**
  - Registers
  - Local memory

- **Block of threads:**
  - Shared memory

- **All blocks:**
  - Global memory
Memory hierarchy

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory

- All blocks:
  - Global memory
Shared memory

- Allocated per thread block
- Same lifetime as the block
- Accessible by any thread in the block
- Low latency
- High aggregate bandwidth
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing global memory accesses)
Sharing data between threads

• Broadcast a value
• One writer thread
• Multiple reader threads
• Value is scoped to the grid
Sharing data between threads

- Broadcast a value
- One writer thread
- Multiple reader threads
- Value is scoped to the grid
Sharing data between threads

- Broadcast a value
- One writer thread
- Multiple reader threads
- Value is scoped to the grid
Sharing data between threads

- Broadcast a value
- One writer thread
- Multiple reader threads
- Value is scoped to the grid
Broadcast Implementation

```c
__global__ int bcast(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x;
    x = a;
    // do some work
}
```
Sharing data between threads
Sharing data between threads

- Data access hazard
- Data being read in thread 2 can be stale
- Need ordering
Racecheck : Overview

- Mutations
  - Inconsistent data

- Detect three types of hazards
  - Write after Write (WAW)
  - Read after Write (RAW)
  - Write after Read (WAR)

- Internal heuristics
  - Reduce false positives
  - Prioritize hazards
Racecheck : Usage

- Built into cuda-memcheck
  - Use option --tool racecheck
    
    ```
    $ cuda-memcheck --tool racecheck <my_app> <my_app_options>
    ```

- Byte accurate
- Can provide source file and line
- Other useful options :
  - `save` to save output to a disk
  - `print-level` to control output
Racecheck : Internal Heuristic Filters

- Each report is assigned a priority
  - Error
    - Highest priority
  - Warning
    - Usually hit only by advanced users
  - Information
    - Same data for a Write After Write conflict (WAW)

- Hazard visibility can be controlled using `--print-level` option
Racecheck : Broadcast

```c
__global__ int bcast(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x;
    x = a;
}
```

- Launch of 64 threads
- Ran app with Racecheck
Racecheck : Broadcast

- On a 16 SM GF100
- 4 errors found (1 report per byte)
- RAW (Read after Write) hazards
  - Based on executed interleaving
- Identified bad accesses to shared memory

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x0000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x0000000e8 in race.cu:27:bcast(void)
  Current Value : 0
ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
  Current Value : 0
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value: 0

- Priority level of report
Racecheck : Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
  Current Value : 0

- Priority level of report
- Type of hazard
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value : 0

- Priority level of report
- Type of hazard
- Location of hazard
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value : 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
Racecheck : Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value : 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
- Per thread
  - Access type
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value : 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
- Per thread
  - Access type
  - Thread index (x, y, z)
Racecheck : Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value : 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
- Per thread
  - Access type
  - Thread index (x, y, z)
  - Instruction offset in kernel
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)
Current Value: 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
- Per thread
  - Access type
  - Thread index (x, y, z)
  - Instruction offset in kernel
  - File name and line number (if available)
Racecheck: Anatomy of a report

ERROR: Potential RAW hazard detected at __shared__ 0x3 in block (0, 0, 0):
  Write Thread (0, 0, 0) at 0x000000d8 in race.cu:25:bcast(void)
  Read Thread (35, 0, 0) at 0x000000e8 in race.cu:27:bcast(void)

Current Value: 0

- Priority level of report
- Type of hazard
- Location of hazard
- Block index (x, y, z)
- Per thread
  - Access type
  - Thread index (x, y, z)
  - Instruction offset in kernel
  - File name and line number (if available)
- Kernel name
Broadcast Implementation Revisited

```c
__global__ int kernel(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x; \text{ Write}
    x = a; \text{ Read}
    // do some work
}
```

- Unsafe read, write skipped for some threads
- Fix by forcing an order
Fixed Broadcast Implementation

```c
__global__ int kernel(void) {
    int x;
    __shared__ int a;
    if (threadIdx.x == WRITER)
        a = threadIdx.x;
    __syncthreads();
    x = a;
    // do some work
}
```
Stack Back Traces

- Saved host back trace at call site
  - Precise errors: Kernel launch site
  - Global Leaks: cudaMalloc site
  - CUDA API errors: CUDA API call site
- Device function call back trace at error
- Supported host OS: Linux, Mac, Windows
- Supported devices: Fermi+
  - Only in non blocking launch mode
- Enabled by default
Invalid __local__ write of size 4
at 0x000000e8 in localRecursive.cu:24:recursive(int*)
by thread (6,0,0) in block (0,0,0)
Address 0x00fffbfc is out of bounds
Device Frame:recursive(int*) (fibonacci(int, int) : 0xe0)
Device Frame:recursive(int*) (fibonacci(int, int) : 0xe0)
Device Frame:recursive(int*) (fibonacci(int, int) : 0xe0)
Device Frame:recursive(int*) (recursive(int*) : 0x28)
Saved host backtrace up to driver entry point at kernel launch time
Host Frame:libcudat.so (cuLaunchKernel + 0x3ae) [0xcb8ae]
Host Frame:libcudart.so.5.0 [0x11dd4]
Host Frame:libcudart.so.5.0 (cudaLaunch + 0x182) [0x3ad82]
Host Frame:localRecursive (__Z28__device_stub__Z9recursivePiPi + 0x33) [0xfa3]
Host Frame:localRecursive (main + 0x2cd) [0x12ad]
Host Frame:/lib64/libc.so.6 (__libc_start_main + 0xfd) [0x1eb1d]
Host Frame:localRecursive [0xdc9]
CUDA API Error Checking

- Checks all CUDA API calls
- Message when call will return an error
- Application will not terminate
- Standalone only
- Enable using --report-api-errors yes
Improved Precise Checking

- Improved precise error reporting
  - Shared loads and stores
  - Local loads and stores
  - Global atomics and reductions
- Error messages now have an address space qualifier
- Enabled in both integrated and standalone modes
- Enabled on all supported architectures
Summary

- CUDA-GDB
  - Usage
  - Attach
  - API error checking

- CUDA-MEMCHECK
  - Usage
  - Shared memory data access hazard detection (race check)
  - Stack back traces
  - API error checking
Thank You

- Availability:
  - CUDA toolkit: http://www.nvidia.com/getcuda

- CUDA experts table

- For more questions, come to our booth on the demo floor

- Repeat session on Wednesday @ 2 pm