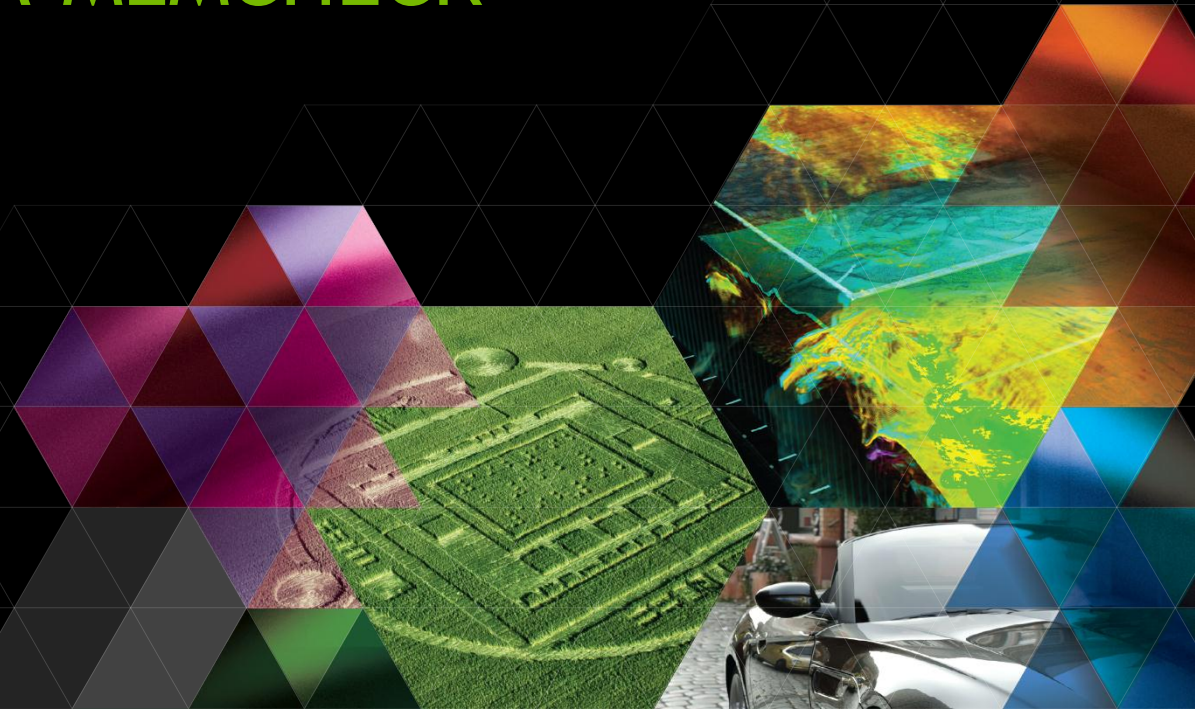


CUDA DEBUGGING TOOLS CUDA-GDB & CUDA-MEMCHECK

Vyas Venkataraman



NVCC COMPILER OPTIONS

- Device side debug : **-G**
 - Line number information
 - Full debug information (variables, functions etc)
 - **Disables Optimizations**
- Line number information : **-lineinfo**
 - Only line number information
 - No additional debug information (no variables)
 - **No impact on optimization**
- Host side options
 - Host debug information **-g**
 - Host symbol information **-Xcompiler -rdynamic**

WHAT IS CUDA-MEMCHECK ?

- “Why did my kernel fail ?”
- The first tool you should run
- Functional correctness tool suite
- Run time error checker : *memcheck*
 - Precise errors : Memory access
 - Imprecise errors : Hardware reported (SM 2.0+)
- Shared memory hazard checker : *racecheck*
- Cross platform : Linux, Mac, Windows
- Also integrated into cuda-gdb (Linux / Mac Only)

RUNNING CUDA-MEMCHECK

- Standalone

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

- Default to *memcheck* tool
- Detects misaligned and out of bound access in GPU 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
```

— Multiple precise errors using *--destroy-on-device-error kernel*

RUNNING CUDA-MEMCHECK

- Imprecise errors

- Can be a few instructions away

```
Out-of-range Shared or Local Address
```

```
at 0x00000798 in kernel.cu:110:test(bool)  
by thread (0,0,0) in block (0,0,0)
```

- On SM 5.0, the PC of the error is precisely attributed **New in 6.0**

DEVICE MALLOC()/FREE() CHECKING

- Double free() / Invalid free()

```
Malloc/Free error encountered : Double free  
    at 0x0002de18  
    by thread (1,0,0) in block (0,0,0)  
    Address 0x50c8b99a0
```

LEAK CHECKING

- Enable with :

```
$ cuda-memcheck --leak-check full <my_app>
```

- Allocation not freed at cuCtxDestroy/cudaDeviceReset()

```
Leaked 64 bytes at 0x5047c0200
```

— Host backtrace at cudaMalloc time

- Device heap

```
Leaked 16 bytes at 0x5058bf2e4 on the device heap
```

CUDA API ERROR CHECKING

- Enabled by default

```
$ cuda-memcheck --report-api-errors yes <my_app>
```

- CUDA Driver API

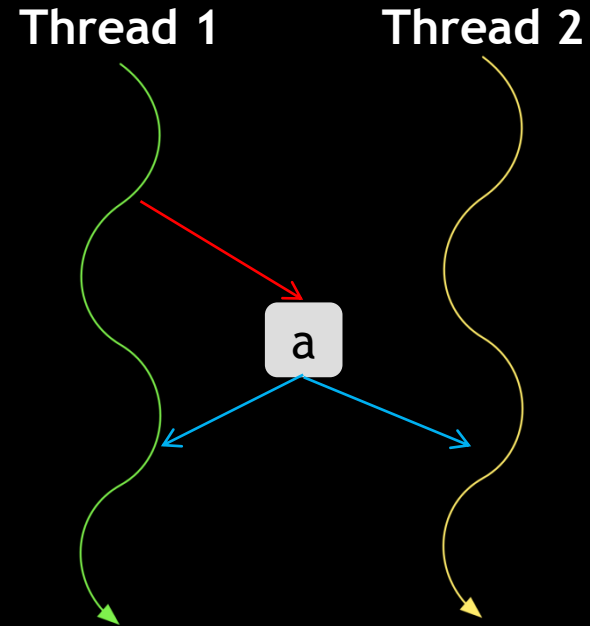
```
Program hit error 1 on CUDA API call to cuMemFree_v2
```

- CUDA Runtime API

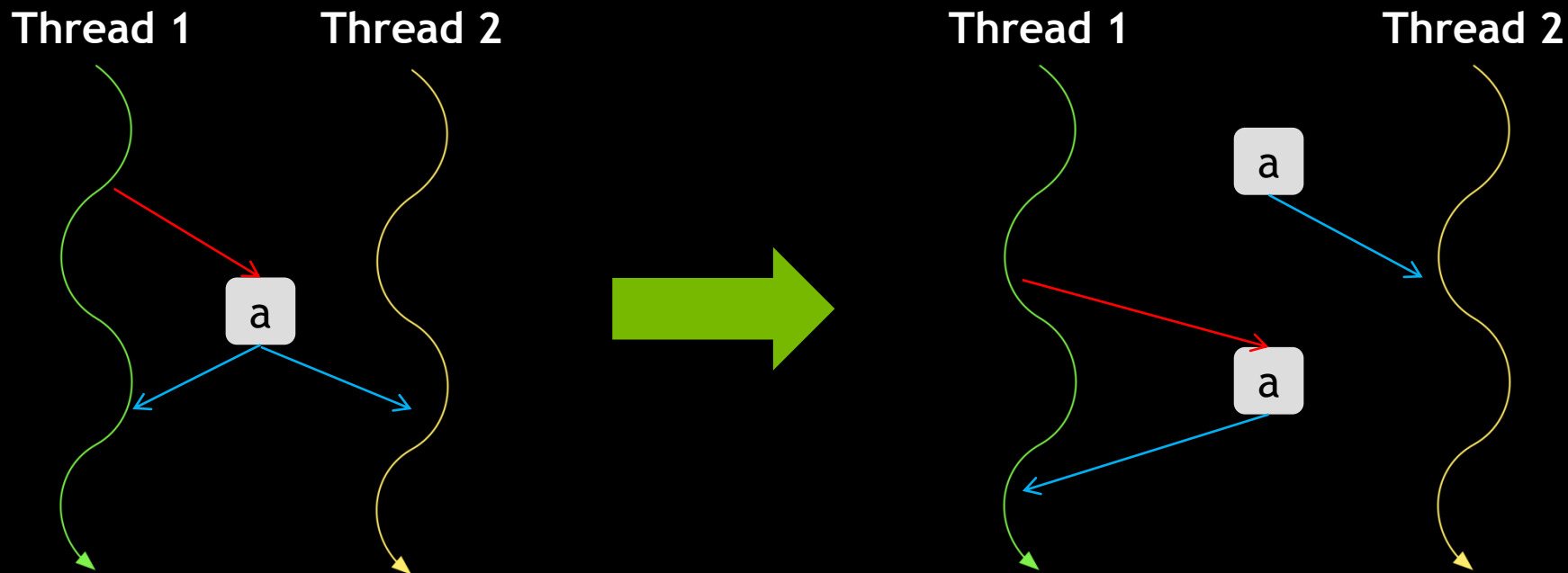
```
Program hit error 17 on CUDA API call to cudaFree
```


BROADCAST IMPLEMENTATION

```
__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



- Data access hazard
- Data being read in **thread 2** can be stale
- Need ordering

CUDA-MEMCHECK TOOL : RACECHECK

- Built into cuda-memcheck
 - Use option *--tool racecheck*

```
$ cuda-memcheck --tool racecheck <my_app> <my_app_options>
```

- Default : Byte accurate hazard reports
- Can provide source file and line
- Other useful options :
 - *--save* to save output to a disk
 - *--print-level* to control output

RACECHECK ANALYSIS MODE

- Invoke with

```
$ cuda-memcheck --tool racecheck --racecheck-report analysis  
<my_app> <my_app_options>
```

- Analyzes thousands of hazard reports to produce simple user guidance

```
Race reported between Write access at 0x00000018 in  
raceGroupBasic.cu:13:Basic(void)  
    and Write access at 0x00000018 in  
raceGroupBasic.cu:13:Basic(void)
```

CUDA-MEMCHECK FEATURES

- Misaligned and out of bounds memory access
- Hardware error reporting
- Shared memory hazard detection
- Device side malloc()/free() error checking
- Device heap allocation leak checking
- Device + Host stack back traces
- CUDA API error checking
- Name demangling (with parameters) for kernels

CUDA-GDB OVERVIEW

- What is it? What does it let you do?
 - Command line source and assembly (SASS) level debugger
 - Feature set parity with Nsight Eclipse Edition
 - 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

EXECUTION CONTROL

- 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
 - Applies to 32 threads at a time (a warp)

Single-Stepping	At the source level	At the assembly level
Over function calls	<code>next</code>	<code>nexti</code>
Into function calls	<code>step</code>	<code>stepi</code>

- Behavior varies when stepping `__syncthreads()`

PC at a <i>barrier</i> ?	Single-stepping applies to	Notes
Yes	All threads in the current <u>block</u> .	Required to step over the barrier.
No	<u>Active threads</u> in the current warp.	

BREAKPOINTS

- By name

```
(cuda-gdb) break my_kernel  
(cuda-gdb) break _Z6kernelIfiEvPT_PT0
```

- By file name and line number

```
(cuda-gdb) break acos.cu:380
```

- By address

```
(cuda-gdb) break *0x3e840a8  
(cuda-gdb) break *target_var
```

- 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, ...)

```
(cuda-gdb) break acos.cu:380 if (...)
```

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)]
```

- Can also switch by HW coordinates : device/SM/warp/lane

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)
```

THREADS

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

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

	Block	Thread	To	Block	Thread	Cnt	PC	Filename	Line
*	(0,0,0)	(0,0,0)	(3,0,0)	(7,0,0)	32	0x7fae70		acos.cu	380
	(4,0,0)	(0,0,0)	(7,0,0)	(7,0,0)	32	0x7fae60		acos.cu	377

- 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  
  
(cuda-gdb) set my_variable = 6  
$4 = 6
```

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

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                0xffffc68    16776296
R4                0x6          6
```

- Modify one register

```
(cuda-gdb) print $R3 = 3
```

CODE DISASSEMBLY

```
(cuda-gdb) x/10i $pc
```

```
0x123830a8 <_Z9my_kernel+8>:  MOV R0, c [0x0] [0x8]
0x123830b0 <_Z9my_kernel+16>: MOV R2, c [0x0] [0x14]
0x123830b8 <_Z9my_kernel+24>:  IMUL.U32.U32 R0, R0, R2
0x123830c0 <_Z9my_kernel+32>:  MOV R2, R0
0x123830c8 <_Z9my_kernel+40>:  S2R R0, SR_CTAid_X
0x123830d0 <_Z9my_kernel+48>:  MOV R0, R0
0x123830d8 <_Z9my_kernel+56>:  MOV R3, c [0x0] [0x8]
0x123830e0 <_Z9my_kernel+64>:  IMUL.U32.U32 R0, R0, R3
0x123830e8 <_Z9my_kernel+72>:  MOV R0, R0
0x123830f0 <_Z9my_kernel+80>:  MOV R0, R0
```

GPU ATTACH

- Run your program at full speed, then attach with cuda-gdb/Nsight EE
- 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 ERROR REPORTING IN CUDA-GDB

- CUDA API error reporting (three modes)

1. Trace all CUDA APIs that return an error code (default)

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

2. Stop in the debugger when any CUDA API fails
3. Hide all CUDA API errors (do not print them)

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

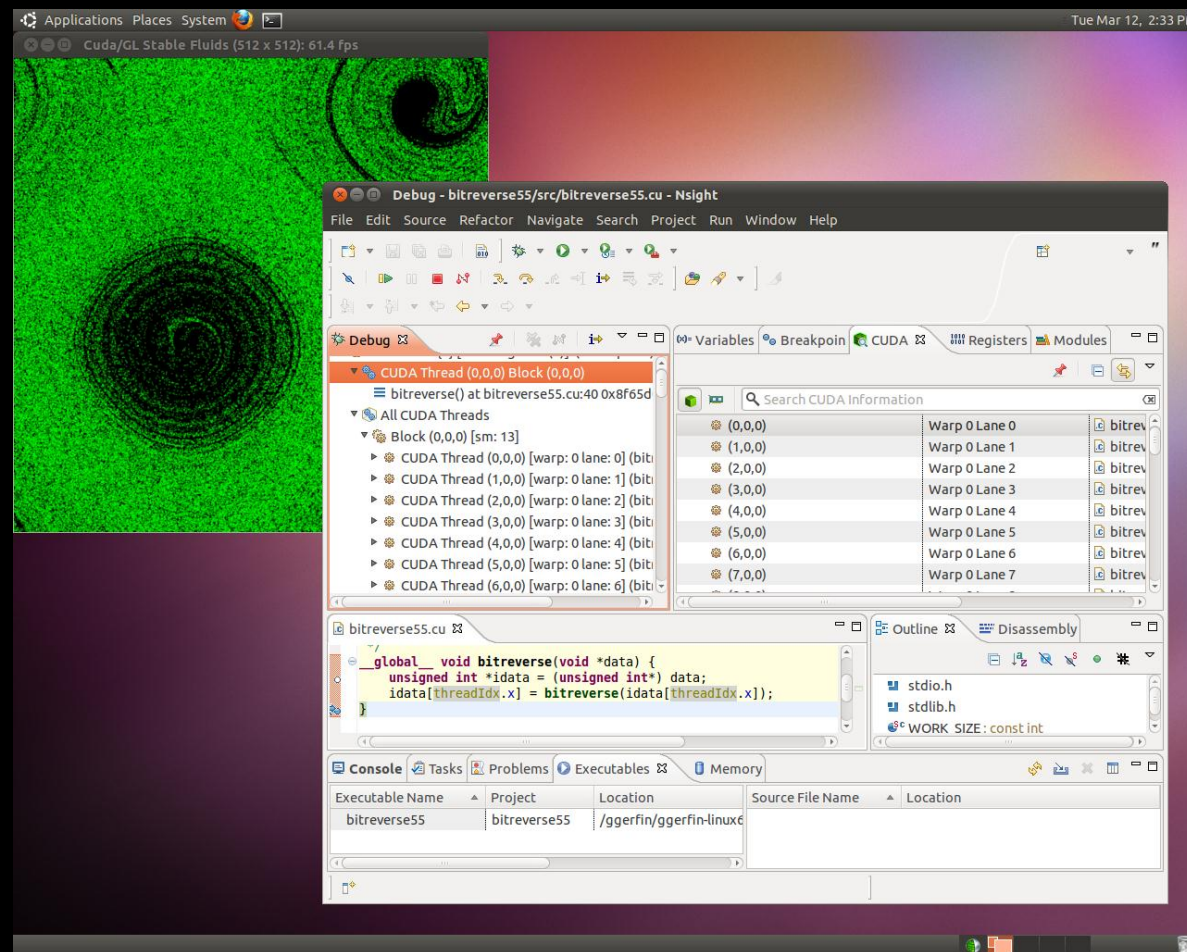
- Enhanced interoperation with cuda-memcheck

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

```
Memcheck detected an illegal access to address (@global)0x500200028
```

SINGLE GPU DEBUGGING

- BETA feature in CUDA-GDB and in Nsight EE
- SM 3.5+ Only
- CDP debugging supported



CUDA DYNAMIC PARALLELISM LAUNCH TRACE

- Examine ancestors of Cuda Dynamic Parallelism GPU launched kernels

```
(cuda-gdb) info cuda launch trace
```

```
Lvl Kernel Dev Grid Status GridDim BlockDim Invocation
```

```
* #0 5 0 -7 Active (1,1,1) (1,1,1) cdp_launchtrace(int) (depth = 3)
#1 3 0 -6 Sleeping (1,1,1) (1,1,1) cdp_launchtrace(int) (depth = 2)
#2 2 0 -5 Sleeping (1,1,1) (1,1,1) cdp_launchtrace(int) (depth = 1)
#3 1 0 2 Sleeping (1,1,1) (1,1,1) cdp_launchtrace(int) (depth = 0)
```

UNIFIED MEMORY

- New in CUDA 6.0
- Transparent host and device access
- Removes the need for `cudaMemcpy`
- Global/file-scope static variables `__managed__`
- Dynamic allocation : *`cudaMallocManaged`*

UNIFIED MEMORY

```
void sortfile(FILE *fp, int N) {  
    char *gpu_data, *host_data;  
    cudaMalloc(&gpu_data, N);  
  
    char *host_data = (char *)malloc(N);  
    fread(host_data, 1, N, fp);  
    cudaMemcpy(gpu_data, host_data, N, ...);  
  
    sort<<< ... >>>(gpu_data, N);  
    cudaDeviceSynchronize();  
  
    cudaMemcpy(host_data, gpu_data, N, ...);  
    use_data(host_data);  
  
    free(host_data);  
    cudaFree(gpu_data);  
}
```



```
void sortfile(FILE *fp, int N) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    sort<<< ... >>>(data, N);  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

UNIFIED MEMORY IN CUDA-GDB

- Print variables

```
(cuda-gdb) print managed_ptr
```

```
$1 = (@managed int *) 0x204600000
```

```
(cuda-gdb) print managed_var
```

```
$2 = 3
```

```
(cuda-gdb) print &managed_var
```

```
$3 = (@managed int **) 0x204500000
```

- Info cuda managed

```
(cuda-gdb) info cuda managed
```

```
Static managed variables on host are:
```

```
managed_var = 3
```

UNIFIED MEMORY

- Access rules
 - CPU cannot access memory while GPU is accessing it
 - CPU signal delivered

```
__managed__ int x;  
__global__ int kern(void) {  
    x = 2;  
}  
int main(void) {  
    x = 1; // Legal  
    kern<<<1,1>>>();  
    x = 3; // Illegal  
}
```

UNIFIED MEMORY

- Access rules
 - CPU cannot access memory while GPU is accessing it
 - CPU signal delivered

```
__managed__ int x;  
__global__ int kern(void) {  
    x = 2;  
}  
  
int main(void) {  
    x = 1; // Legal  
    kern<<<1,1>>>();  
    cudaDeviceSynchronize();  
    x = 3; // Legal  
}
```

UNIFIED MEMORY

- CUDA-GDB will detect signals from bad CPU accesses
- Special signal information printed

```
$ cuda-gdb myApplication
```

```
Program received signal CUDA_EXCEPTION_15, Invalid Managed Memory Access.  
0x0000000000402800 in main () at uvm.cu:10  
10          x = 3;
```

```
(cuda-gdb) p &x  
$1 = (@managed int **) 0x204500000
```

UNIFIED MEMORY

- Assign memory visibility
 - Host only
 - All streams on Device
 - Per stream
- Default can be set at creation time
 - 3rd parameter to `cudaMallocManaged`
 - Default only allows Host only or all streams
- Can be changed dynamically
 - `cudaStreamAttachMemAsync`
- Controls access from GPU
 - No enforced correctness : Use `cuda-memcheck` !

CUDA-MEMCHECK + MANAGED MEMORY

- Check GPU accesses to managed memory
 - Out of Bounds access
 - Attachment based invalid access
 - Misaligned access
 - Leak checking

SIMPLE KERNEL

```
__global__ int kern(int *x) {  
    *x = 2;  
}  
  
int main(void) {  
    int *x;  
    cudaMallocManaged((void**)&x, sizeof(*x), cudaMemAttachHost);  
    *x = 1;  
    kern<<<1,1>>>(x);  
    cudaDeviceSynchronize();  
}
```

PRECISE ERROR DETECTION

```
Invalid __global__ read of size 4  
    at 0x00000028 in uvm.cu:2:kern(int*)  
    by thread (0,0,0) in block (0,0,0)  
    Address 0x20450000 is out of bounds
```

SUPPORT FOR MPS (NEW IN 6.0)

- CUDA Multi Process Service (MPS)
 - Allows multiple software CUDA contexts to share a single device
- Memcheck can run on any MPS client
- Precise errors do not affect other MPS clients

CUDA-GDB NEW FEATURES IN 6.0

- Support for SM 5.0
 - Supports precise attribution of hardware exceptions

```
The exception was triggered at PC 0xa8d080 (test.cu:94)
```

- Single stepping optimizations

```
(cuda-gdb) set cuda single_stepping_optimizations off
```

- No launch notifications

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

THANK YOU

- CUDA 6.0 : <http://www.nvidia.com/getcuda>
- Recordings from GTCs (<http://gputechconf.com>)
- Demo booth @GTC
- Experts Table @GTC
- Online documentation (<http://docs.nvidia.com/cuda/>)
- Forums (<http://devtalk.nvidia.com/>)
- Email : cudatools@nvidia.com