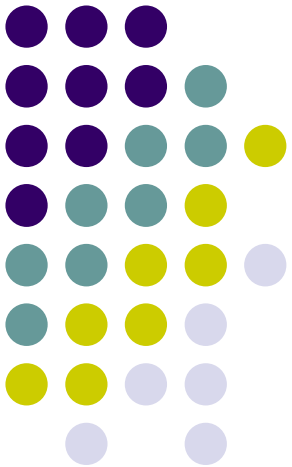
ME964
High Performance Computing for Engineering Applications

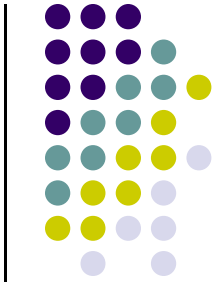
Debugging & Profiling CUDA Programs

February 28, 2012



"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"
Brian Kernighan

Before We Get Started…

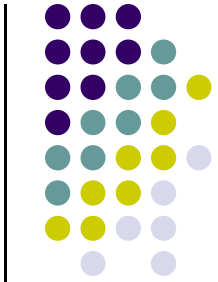


- Last time
  - Discussed about registers, local memory, and shared memory
  - Discussed execution scheduling on the GPU
    - Important concept: Warp – collection of 32 threads that are scheduled for execution as a unit

- Today
  - Quick intro, debugging and profiling CUDA applications
  - Wrap up on Th, with one debugging and profiling example

- Other issues
  - HW5 due Th at 11:59 PM
  - Midterm project topic needs to be chosen by March 8
    - Check the online syllabus for details, we'll discuss more on Th

Acknowledgments



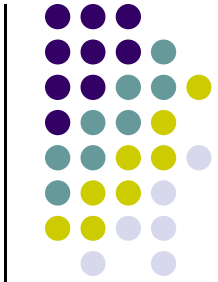
- Today's slides include material provided by Sanjiv Satoor of NVIDIA India

- All the good material in this presentation comes from him

- All the strange stuff and any mistakes belong to me

Terminology

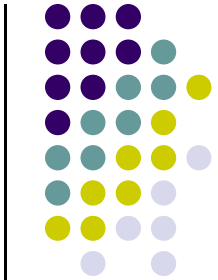
[Review]



- Kernel
  - Function to be executed in parallel on one CUDA device
  - A kernel is executed by multiple blocks of threads
- Block
  - 3-dimensional
  - Made up of a collection of threads
- Warp
  - Group of 32 threads scheduled for execution as a unit
- Thread
  - Smallest unit of work

Terminology

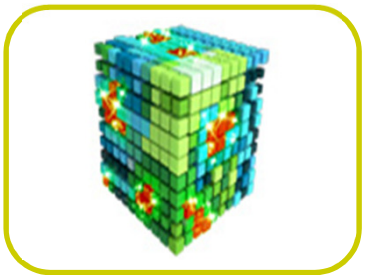
[Review]



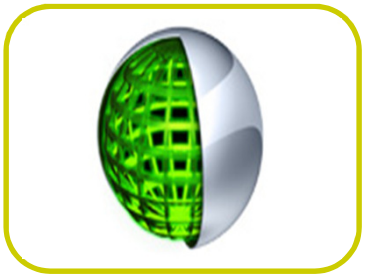
- Divergence
  - Occurs when any two threads on the same warp are slated to execute different instructions
  - Active threads within a warp: threads currently executing device code
  - Divergent threads within a warp: threads that are waiting for their turn or are done with their turn.
- Program Counter (PC)
  - A processor register that holds the address (in the virtual address space) of the next instruction in the instruction sequence
    - Each thread has a PC
  - Useful with **`cuda-gdb`** debugging to navigate the instruction sequence

Debugging Solutions

CUDA-GDB
(Linux & Mac)



CUDA-MEMCHECK
(Linux, Mac, & Windows)

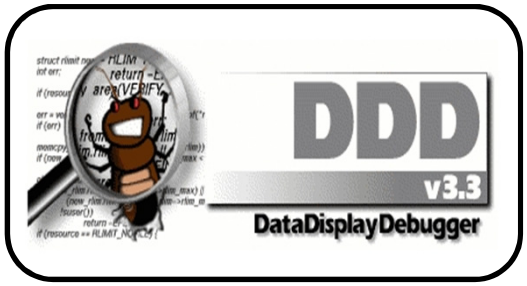


NVIDIA Parallel NSight
(Windows)

Allinea DDT

Rogue Wave TotalView

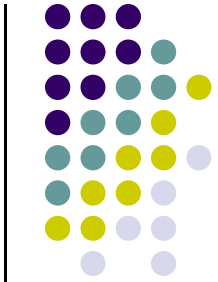
CUDA-GDB GUI Wrappers



GNU DDD

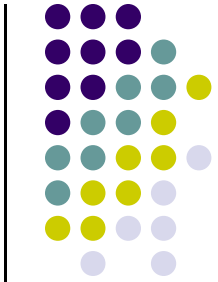
GNU Emacs

CUDA-GDB Main Features



- All the standard GDB debugging features
- Integrated CPU and GPU debugging within a single session
- Breakpoints and Conditional Breakpoints
- Inspect memory, registers, local/shared/global variables
- Supports multiple GPUs, multiple contexts, multiple kernels
- Source and Assembly (SASS) Level Debugging
- Runtime Error Detection (stack overflow,...)

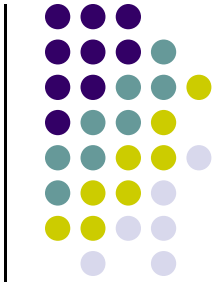
Recommended Compilation Flags



- Compile code for your target architecture:
  - Tesla : `-gencode arch=compute_10,code=sm_10`
  - Fermi : `-gencode arch=compute_20,code=sm_20`
- Compile code with the debug flags:
  - Host code : `-g`
  - Device code : `-G`
- Example:

```
$ nvcc -g -G -gencode arch=compute_20,code=sm_20 test.cu -o test
```

Usage
[On your desktop]

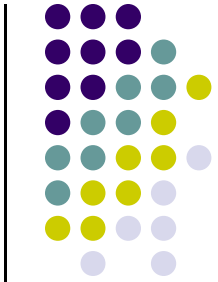


- Invoke **`cuda-gdb`** from the command line:

```
$ cuda-gdb my_application
(cuda-gdb) _
```

Usage, Further Comments…

[Invoking `cuda-gdb` on Euler]



- When you request resources for running a debugging session on Euler don't reserve the GPU in exclusive mode (use "`default`"):

```
>> qsub -I -l nodes=1:gpus=1:default -X
```
(qsub –eye –ell node…)

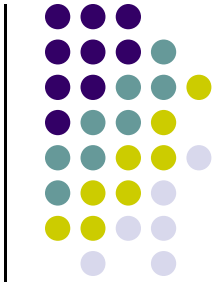
```
>> cuda-gdb myApp
```

- You can use `ddd` as a front end (you need the `-X` in the `qsub` command):

```
>> ddd --debugger cuda-gdb myApp
```

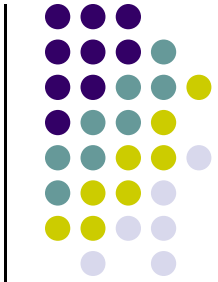
- If you don't have `ddd` around, use at least

```
>> cuda-gdb -tui myApp
```



Program Execution Control

Execution Control



- Execution Control is identical to host debugging:
  - Launch the application

```
(cuda-gdb) run
```

  - Resume the application (all host threads and device threads)

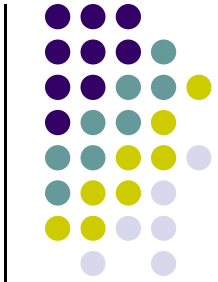
```
(cuda-gdb) continue
```

  - Kill the application

```
(cuda-gdb) kill
```

  - Interrupt the application: CTRL-C

Execution Control



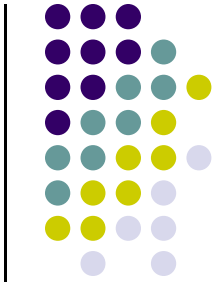
- Single-Stepping

Single-Stepping	At the source level	At the assembly level
Over function calls	`next`	`nexti`
Into function calls	`step`	`stepi`

- Behavior varies when stepping `__syncthreads()`

PC at a *barrier?*	Single-stepping applies to	Notes
Yes	Active and divergent threads of the warp in focus and all the warps that are running the same **block**.	Required to step over the barrier.
No	**Active threads** in the warp in focus only.	

Breakpoints



- By name

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

- By file name and line number

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

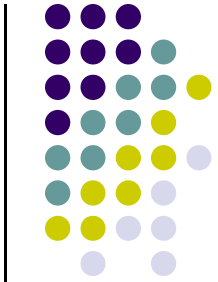
- By address

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

- At every kernel launch

```
(cuda-gdb) set cuda break_on_launch application
```

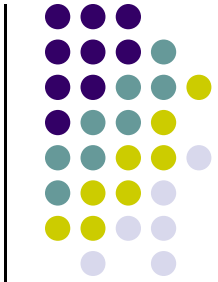
Conditional Breakpoints



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

- Condition
  - C/C++ syntax
  - No function calls
  - Supports built-in variables (`blockIdx`, `threadIdx`, ...)

Conditional Breakpoints

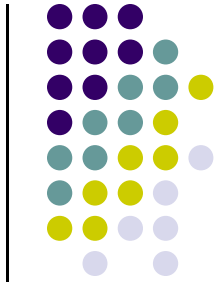


- Set at breakpoint creation time

```
(cuda-gdb) break my_kernel if threadIdx.x == 13
```

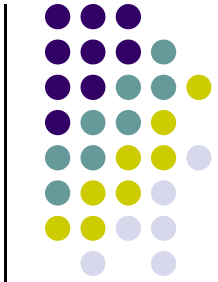
- Set after the breakpoint is created
  - Breakpoint 1 was previously created

```
(cuda-gdb) condition 1 blockIdx.x == 0 && n > 3
```



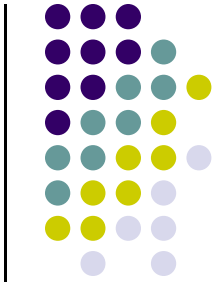
Thread Focus

Thread Focus



- There are thousands of threads to deal with. Can't display all of them
- Thread focus dictates which thread you are looking at
- Some commands apply only to the thread in focus
  - Print local or shared variables
  - Print registers
  - Print stack contents
- Attributes of the "thread focus"
  - Kernel index : unique, assigned at kernel's launch time
  - Block index : the application `blockIdx`
  - Thread index : the application `threadIdx`

Devices

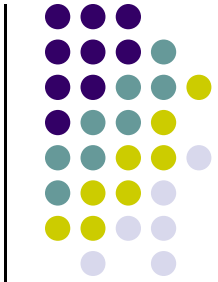


- To obtain the list of devices in the system:

```
(cuda-gdb) info cuda devices

 Dev   Desc    Type  SMs Wps/SM Lns/Wp Regs/Ln Active SMs Mask
*  0  gf100  sm_20   14     48     32      64            0xfff
   1  gt200  sm_13   30     32     32     128              0x0
```

- The * indicates the device of the kernel currently in focus
- Provides an overview of the hardware that supports the code

Kernels

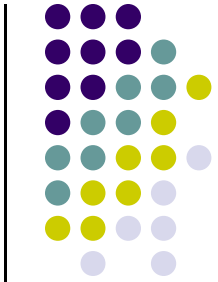


- To obtain the list of running kernels:

```
(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
- There is a one-to-one mapping between a kernel id (unique id across multiple GPUs) and a (dev,grid) tuple. The grid id is unique per GPU only
- The name of the kernel is displayed as are its size and its parameters
- Provides an overview of the code running on the hardware

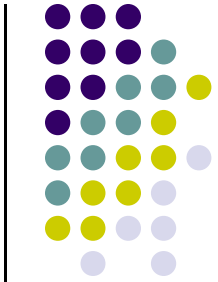
Thread Focus



- 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)
```

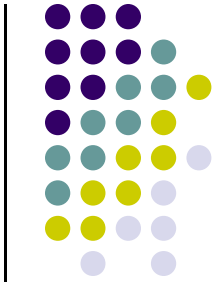
Thread Focus



- 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)
```

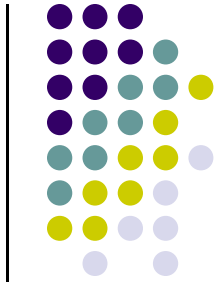
Threads



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

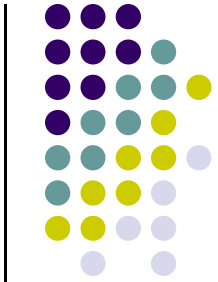
```
(cuda-gdb) info cuda threads kernel 2

    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
- Threads displayed as (block, thread) ranges
- **`Cnt`** indicates the number of threads within each range
  - All threads in the same range are contiguous (no hole)
  - All threads in the same range shared the same PC (and filename/line number)



Program State Inspection

Stack Trace



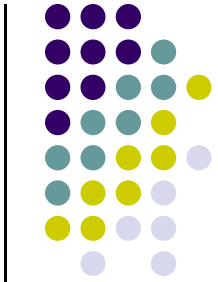
- Same (aliased) commands as in **gdb**:

```
(cuda-gdb) where
(cuda-gdb) bt
(cuda-gdb) info stack
```

- Applies to the thread in focus

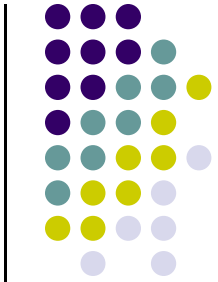
- On Tesla, all the functions are always inlined

Stack Trace



```
(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
```

Source Variables



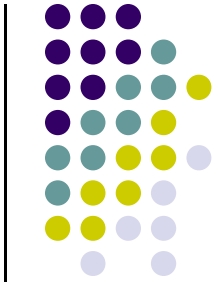
- Source variable must be live (in the scope)
- Read a source variable

```
(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
```

Memory



- Memory read & written like source variables

```
(cuda-gdb) print *my_pointer
```

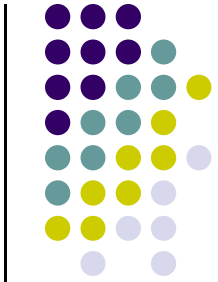
- May require storage specifier when ambiguous

  @global, @shared, @local

  @generic, @texture, @parameter

```
(cuda-gdb) print * (@global int *) my_pointer

(cuda-gdb) print ((@texture float **) my_texture)[0][3]
```

Hardware Registers



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

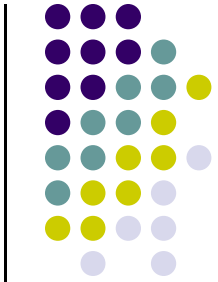
- Show a list of registers (no argument to get all of them)

```
(cuda-gdb) info registers R0 R1 R4
R0             0x6        6
R1             0xfffc68   16776296
R4             0x6        6
```

- Modify one register

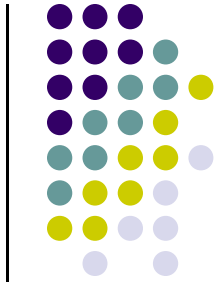
```
(cuda-gdb) print $R3 = 3
```

Code Disassembly



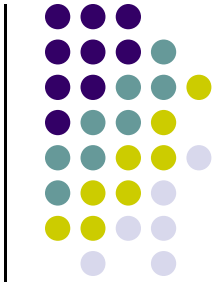
- Must have **`cuobjdump`** in **`$PATH`**

```
(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
```



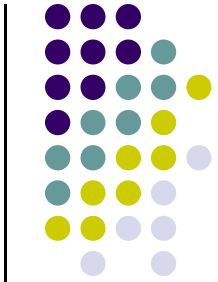
Run-Time Error Detection

CUDA-MEMCHECK



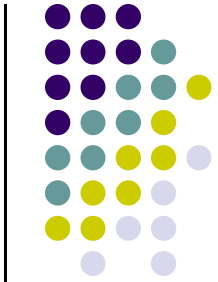
- Stand-alone run-time error checker tool
- Detects memory errors like stack overflow, illegal global address,...
- Similar to `valgrind`
- No need to recompile the application
- Not all the error reports are precise
- Once used within **`cuda-gdb`**, the kernel launches are blocking

CUDA-MEMCHECK ERRORS



Illegal global address
Misaligned global address
Stack memory limit exceeded
Illegal shared/local address
Misaligned shared/local address
Instruction accessed wrong memory
PC set to illegal value
Illegal instruction encountered
Illegal global address

CUDA-MEMCHECK

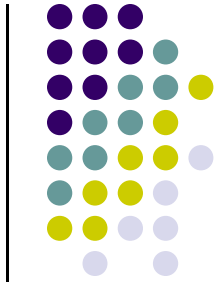


- Integrated in **`cuda-gdb`**
  - More precise errors when used from **`cuda-gdb`**
  - Must be activated before the application is launched

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

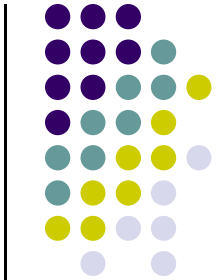
  - What does it mean “more precise”?
    - Precise
      - Exact thread idx
      - Exact PC
    - Not precise
      - A group of threads or blocks
      - The PC is several instructions after the offending load/store

Example



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

(cuda-gdb) run
[Launch of CUDA Kernel 0 (applyStencil1D) on Device 0]
Program received signal CUDA_EXCEPTION_1, Lane Illegal Address.
applyStencil1D<<<(32768,1,1),(512,1,1)>>> at stencil1d.cu:60

(cuda-gdb) info line stencil1d.cu:60
out[ i ] += weights[ j + RADIUS ] * in[ i + j ];
```

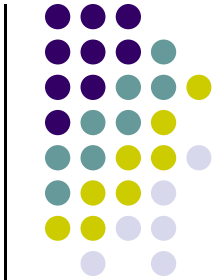
Increase precision



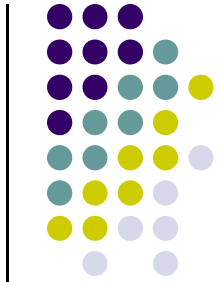
- Single-stepping
  - Every exception is automatically precise
- The "**`autostep`**" command
  - Define a window of instructions where we think the offending load/store occurs
  - **`cuda-gdb`** will single-step all the instructions within that window automatically and without user intervention

```
(cuda-gdb) autostep foo.cu:25 for 20 lines

(cuda-gdb) autostep *$pc for 20 instructions
```

New in CUDA 4.1

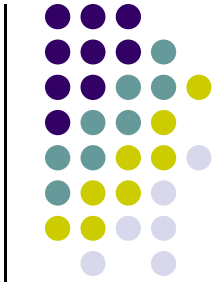


- Source base upgraded to **`gdb`** 7.2
- Simultaneous **`cuda-gdb`** sessions support
- Multiple context support
- Device assertions support
- New "**`autostep`**" command
- More info:
  - http://sbel.wisc.edu/Courses/ME964/2012/Documents/cuda-gdb41.pdf



Tips & Miscellaneous Notes

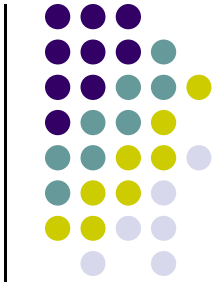
Best Practices



1. Determine scope of the bug
   - Incorrect result
   - Unspecified Launch Failure (ULF)
   - Crash
   - Hang
   - Slow execution

2. Reproduce with a debug build
   - Compile your app with –g –G
   - Rerun, hopefully you can reproduce problem in debug model

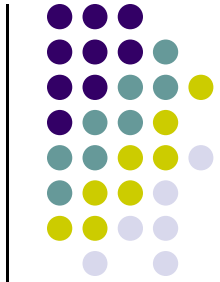
Best Practices



3. Correctness Issues
   - First throw **`cuda-memcheck`** at it in stand-alone
   - Then **`cuda-gdb`** and **`cuda-memcheck`** if needed
   - **`printf`** is also an option, but not recommended

4. Performance Issues (once no bugs evident)
   - Use a profiler (discussed next)
   - Change the code, might have to go back to Step 1. above…

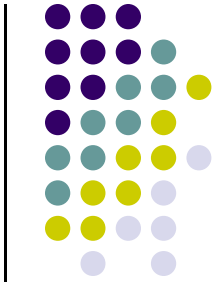
Tips



- Always check the return code of the CUDA API routines!

- If you use **`printf`** from the device code…
  - Make sure to synchronize so that buffers are flushed

Tips



- To hide devices, launch the application with

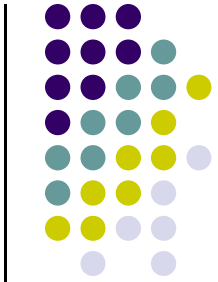
```
CUDA_VISIBLE_DEVICES=0,3
```

  where the numbers are device indexes.

- To increase determinism, launch the kernels synchronously:

```
CUDA_LAUNCH_BLOCKING=1
```

Tips



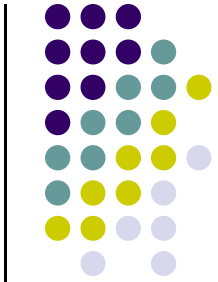
- To print multiple consecutive elements in an array, use @:

```
(cuda-gdb) print array[3]@4
```

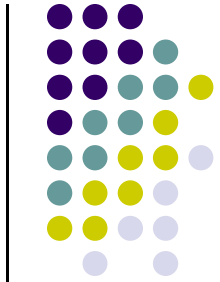
- To find the mangled name of a function

```
(cuda-gdb) set demangle-style none
(cuda-gdb) info function my_function_name
```

Further Information



- **More resources:**
  - CUDA tutorials video/slides at GTC
    - http://www.gputechconf.com/
  - CUDA webinars covering many introductory to advanced topics
    - http://developer.nvidia.com/gpu-computing-webinars
  - CUDA Tools Download: http://www.nvidia.com/getcuda
- **Other related topic:**
  - Performance Optimization Using the NVIDIA Visual Profiler



End Quick Overview, Debugging

Start Quick Overview, Profiling