

The logo for GPU Technology Conference is located in the top-left corner. It consists of a green rectangular box with a small triangle pointing downwards on its left side. Inside the box, the text "GPU" is written in a large, bold, white sans-serif font, and "TECHNOLOGY CONFERENCE" is written in a smaller, white sans-serif font to its right.

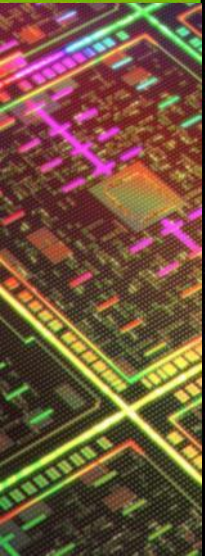
GPU TECHNOLOGY
CONFERENCE

The background of the slide is a detailed, top-down view of a GPU die. The die is dark, and its intricate circuitry is highlighted with vibrant, glowing lines in various colors including red, orange, yellow, green, cyan, blue, and purple. These lines form a complex grid and pattern across the surface of the die.

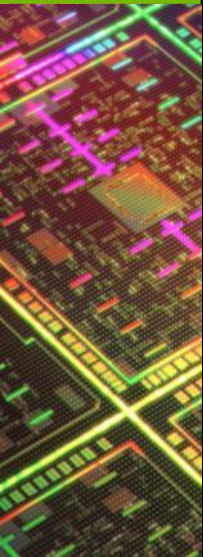
Debugging Your CUDA Applications With CUDA-GDB

Outline

- Introduction
- Installation & Usage
- Program Execution Control
- Thread Focus
- Program State Inspection
- Run-Time Error Detection
- Tips & Miscellaneous Notes
- Conclusion



Introduction



Debugging Solutions



CUDA-GDB
(Linux & Mac)



CUDA-MEMCHECK
(Linux, Mac, & Windows)



NVIDIA Parallel NSight
(Windows)



Allinea
DDT



Rogue Wave
TotalView

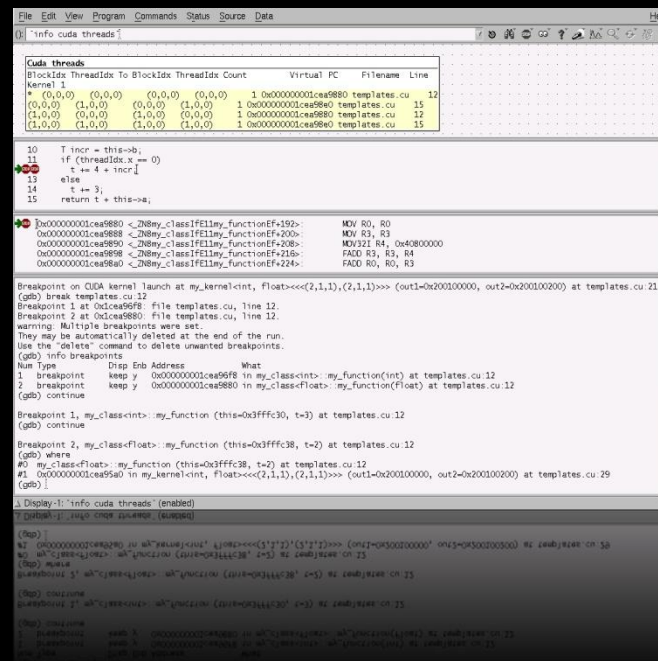
CUDA-GDB GUI Wrappers



GNU DDD

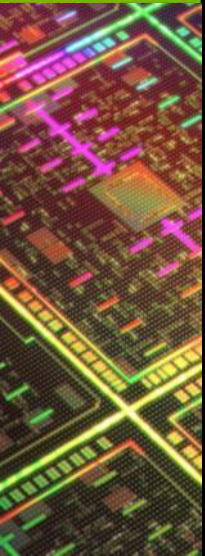


GNU Emacs

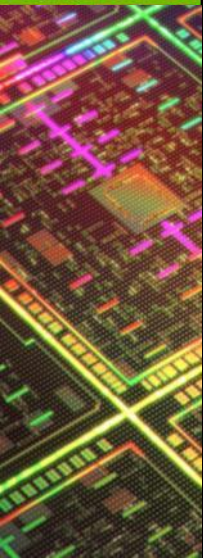


CUDA-GDB Main Features

- All the standard GDB debugging features
- Seamless 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,...)



Installation & Usage



Installation

- Install the CUDA Toolkit:

<http://developer.nvidia.com/cuda-toolkit>

- Invoke CUDA-GDB from the command line:

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


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:

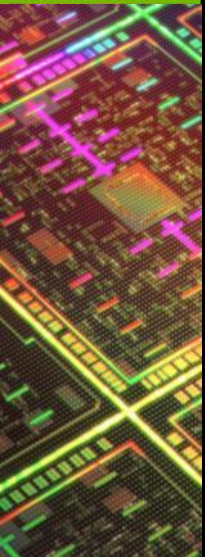
```
$ nvcc -g -G -gencode arch=compute_20,code=sm_20 acos.cu -o acos
```

Usage

CUDA application at a breakpoint == Frozen display

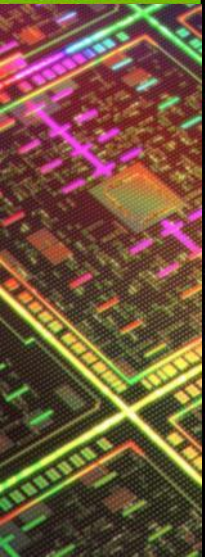
Multiple Solutions:

- Console mode: no X server
- Multiple GPUs: one for display, one for compute
- Remote Debugging: SSH, VNC, ...



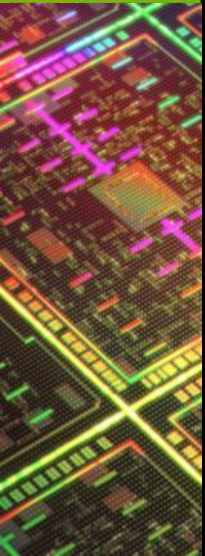
Terminology

- Program Counter (PC)
 - address in the host virtual address space
 - always use virtual PC in cuda-gdb commands
- Divergence
 - if 2 threads on the same warp must execute different instructions, the other must wait
 - active threads: threads currently executing device code
 - divergent threads: threads that are waiting for their turn or are done with their turn.

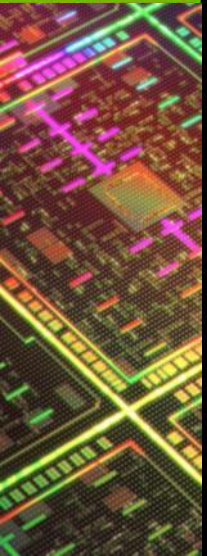


Terminology

- Kernel
 - Function to be executed in parallel on one CUDA device
 - A kernel is executed in multiple blocks of threads
- Block
 - 3-dimensional
 - Executes on 1 or more warps
 - Made of multiple threads
- Warp
 - Group of 32 threads
- Thread
 - Smallest unit of work



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	<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	Active and divergent threads of the warp in focus and all the warps that are running the same <u>block</u> .	Required to step over the barrier.
No	<u>Active threads</u> 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 acos.cu:380
```

- By address

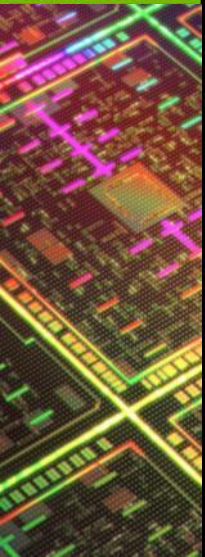
```
(cuda-gdb) break *0x3e840a8  
(cuda-gdb) break *$pc
```

- 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
 - May slow down execution
- Condition
 - C/C++ syntax
 - no function calls
 - support built-in variables (blockIdx, threadIdx, ...)



Conditional Breakpoints

- Set at breakpoint creation time

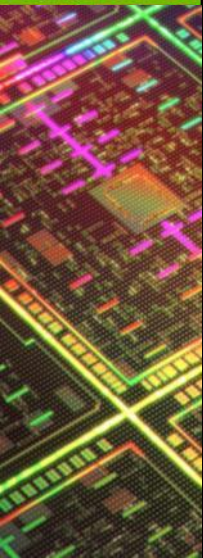
```
(cuda-gdb) break my_kernel if threadIdx.x == 13
```

- Set after the breakpoint is created

- Breakpoint 1 was previously created

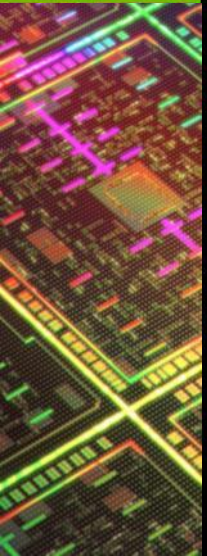
```
(cuda-gdb) condition 1 blockIdx.x == 0 && n > 3
```

Thread Focus



Thread Focus

- 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
```

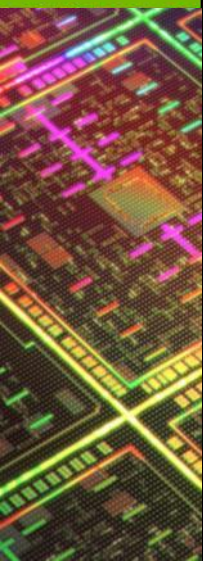
```
[Switching focus to CUDA kernel 2 block (2,0,0), thread (5,0,0)
```

Thread Focus

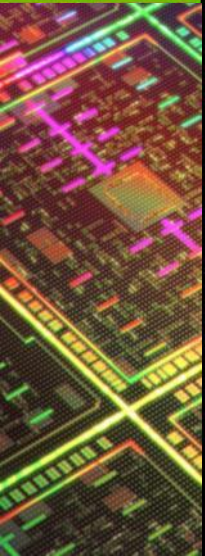
- To obtain the current focus:

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

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



Program State Inspection



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			0xffff
1	gt200	sm_13	30	32	32	128			0x0

- 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
```

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

- 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
- 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 (blank for all)

```
(cuda-gdb) info registers R0 R1 R4
R0          0x6          6
R1          0xffffc68 16776296
R4          0x6          6
```

- Modify one register

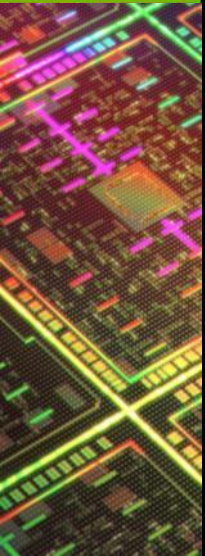
```
(cuda-gdb) print $R3 = 3
```

Code Disassembly

- Must have `cuobjdump` in `$PATH`

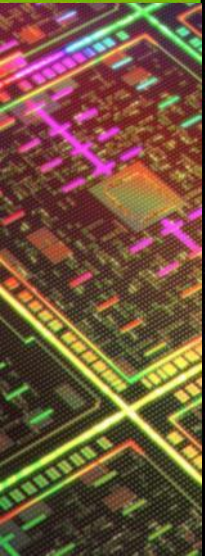
```
(cuda-gdb) x/10i $pc
0x123830a8 <_Z9my_kernel110params+8>:    MOV R0, c [0x0] [0x8]
0x123830b0 <_Z9my_kernel110params+16>:   MOV R2, c [0x0] [0x14]
0x123830b8 <_Z9my_kernel110params+24>:   IMUL.U32.U32 R0, R0, R2
0x123830c0 <_Z9my_kernel110params+32>:   MOV R2, R0
0x123830c8 <_Z9my_kernel110params+40>:   S2R R0, SR_CTAid_X
0x123830d0 <_Z9my_kernel110params+48>:   MOV R0, R0
0x123830d8 <_Z9my_kernel110params+56>:   MOV R3, c [0x0] [0x8]
0x123830e0 <_Z9my_kernel110params+64>:   IMUL.U32.U32 R0, R0, R3
0x123830e8 <_Z9my_kernel110params+72>:   MOV R0, R0
0x123830f0 <_Z9my_kernel110params+80>:   MOV R0, R0
```


Run-Time Error Detection



CUDA-MEMCHECK

- Stand-alone run-time error checker tool
- Detects memory errors like stack overflow,...
- Same spirit as 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

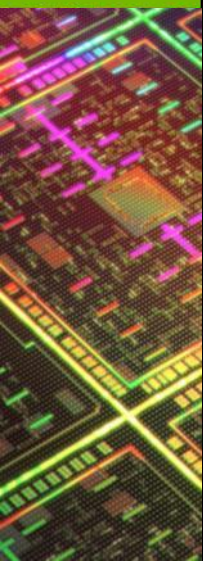
Illegal instruction encountered

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
```



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

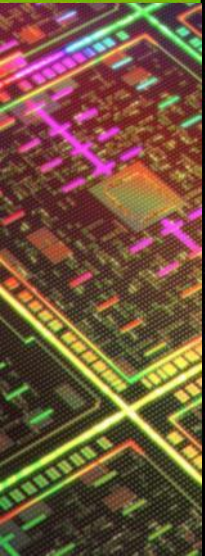
- Single-stepping
 - Every exception is automatically precise
- The “autostep” command
 - Define a window of instructions where we think the offending load/store occurs
 - Cuda-gdb will single-step all the instructions within that window automatically and without user intervention

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

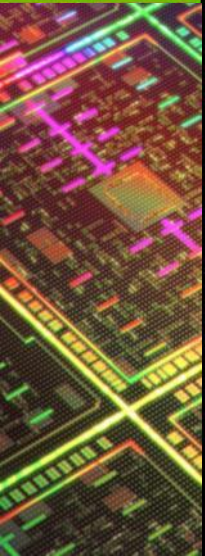
```
(cuda-gdb) autostep *$pc for 20 instructions
```

New in 4.1

- Source base upgraded to GDB 7.2
- Simultaneous cuda-gdb sessions support
- Multiple context support
- Device assertions support
- New “autostep” command



Tips & Miscellaneous Notes



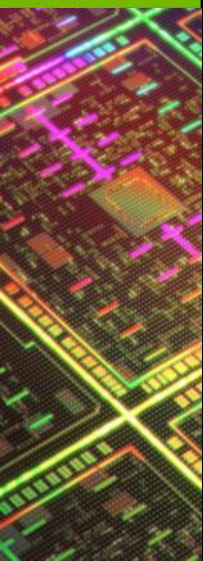
Best Practices

1. Determine scope of the bug

- Incorrect result
- Unspecified Launch Failure (ULF)
- Crash
- Hang
- Slow execution

2. Repro with a debug build

- Compile your app with `-g -G`
- Rerun



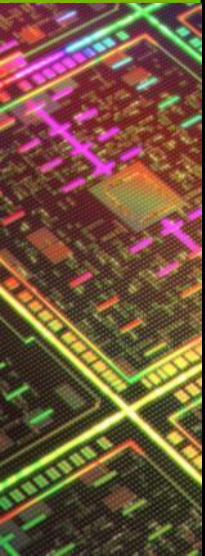
Best Practices

3. Performance Issues

- Use the visual profiler

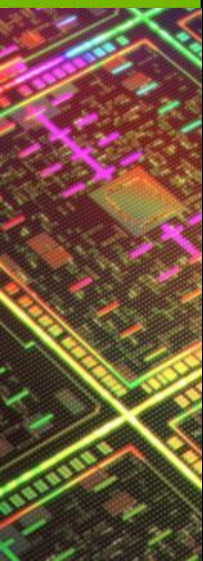
4. Correctness Issues

- First cuda-memcheck stand-alone
- Then CUDA-GDB if needed
- Printf



Tips

- Always check the return code of the CUDA API routines!
- Use `printf` from the device code
 - make sure to synchronize so that buffers are flushed



Tips

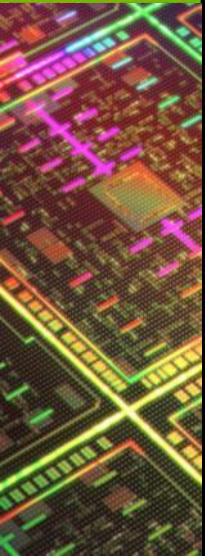
- To hide devices, launch the application with

```
CUDA_VISIBLE_DEVICES=0,1
```

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
```

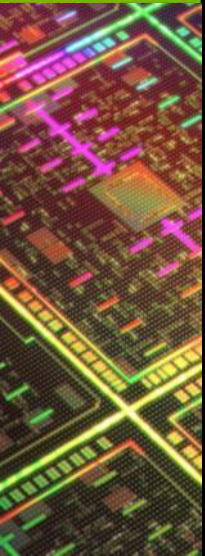
- To find the mangled name of a function

```
(cuda-gdb) set demangle-style none
```

```
(cuda-gdb) info function my_function_name
```

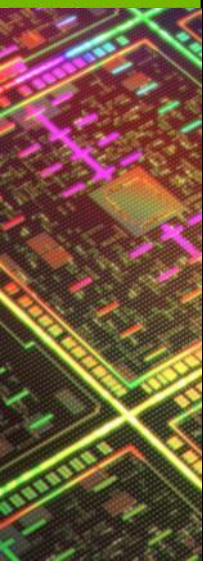
Miscellaneous Notes

- On sm_1x architectures, device functions are always inlined.
 - no stepping over a function call
 - stack trace depth always 1
- Grid Index
 - PTX concept, mostly unused at this point
 - similar to kernel index
 - grid index: unique per device
 - kernel index: unique across all devices



Bug Reporting

- Several Media
 - NVIDIA Forum: <http://developer.nvidia.com/forums>
 - file a bug directly (must be a registered developer)
 - email address: cuatools@nvidia.com

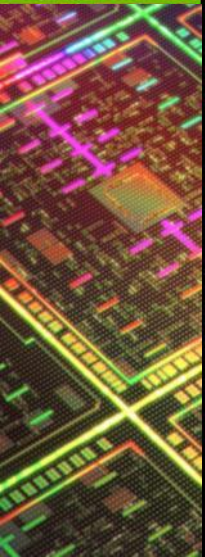


Bug Reporting

- Include as much data as possible
 - output of `nvidia-bug-report.sh`
 - small repro test case with instructions
 - session log with:

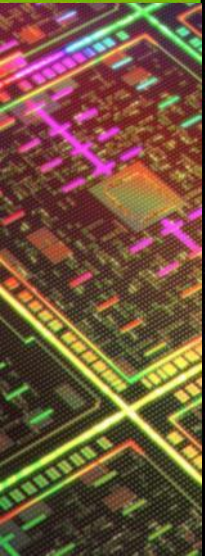
```
(cuda-gdb) set debug cuda general 1  
(cuda-gdb) set debug cuda notifications 1  
(cuda-gdb) set debug infrun 1
```


Conclusion



CUDA-GDB Main Features

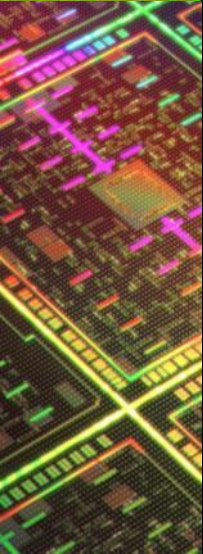
- All the standard GDB debugging features
- Seamless CPU and GPU debugging within a single session
- Breakpoints and Conditional Breakpoints
- Inspect memory, registers, local/shared/global variables
- Supports multiple GPUs, multiple contexts, multiple kernels
- Source and Assembly (SASS) Level Debugging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



Going Further

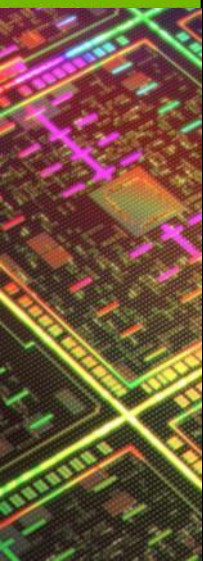
- **More resources:**
 - Prior CUDA tutorials video/slides at GTC
 - <http://www.gputechconf.com/>
 - CUDA webinars covering many introductory to advanced topics
 - <http://developer.nvidia.com/gpu-computing-webinars>
 - CUDA Tools Download: <http://www.nvidia.com/getcuda>
- Other sessions you may be interested in:
 - Performance Optimization Using the NVIDIA Visual Profiler

Questions



Source Variables

- Source variable must be live
 - compiler optimizes code, even with debug builds
 - required because of resource constraints
 - if variable not live at some location, try at another location



CUDA-MEMCHECK

Error Type	Precise?	Description
Out of Bounds Global Access	Yes	Illegal global address
Misaligned Global Access	Yes	Misaligned global address
User Stack Overflow	Yes	Stack memory limit exceeded
Out of Bounds Shared/Local Address	No	Illegal shared/local address
Misaligned Shared/Local Address	No	Misaligned shared/local address
Invalid Address Space	No	Instruction accessed wrong memory
Invalid PC	No	PC set to illegal value
Hardware Stack Overflow	No	Hardware stack overflowed
Illegal Instruction	No	Illegal instruction encountered
Illegal Address	No	Illegal global address

CUDA-MEMCHECK

- Precise
 - Exact thread idx
 - Exact PC
- Not precise
 - A group of threads or blocks
 - The PC is several instructions after the offending load/store

