Addressing the Increasing Challenges of Debugging on Accelerated Multi-Core Systems

Ed Hinkel
Senior Sales Engineer
Agenda

Overview - Rogue Wave & TotalView
Heterogeneous Systems - Then and Now
Debugging Accelerated Systems
  GPU - Nvidia CUDA
  MIC - Intel Phi
Rogue Wave Today

The largest independent provider of cross-platform software development tools and embedded components for the next generation of HPC applications

Visual Numerics®
Leader in embeddable math and statistics algorithms and visualization software for data-intensive applications.

acumem
Leading provider of intelligent software technology which analyzes and optimizes computing performance in single and multi-core environments.

TotalView
Industry-leading interactive analysis and debugging tools for the world's most sophisticated software applications.

Latest addition to the Rogue Wave family: Rogue Wave Visualization for C++
(Formerly IBM’s ILOG Visualization for C++ products)
Representative Customers
Heterogeneous Systems - Then and Now

Heterogeneous Systems and the Need for Speed

A couple of pioneers
Remember SiCortex?
An X-86 host with up to 972 MIPS nodes, with up to 5,832 cores and 7,776 GB of memory
Remember Cell?
RoadRunner

a hybrid design with 12,960 IBM PowerXCell 8i and 6,480 AMD Opteron dual-core processors
What is TotalView?

A comprehensive debugging solution for demanding parallel, heterogeneous, and multi-core applications

- Wide compiler & platform support
  - C, C++, Fortran 77 & 90, UPC
  - Unix, Linux, OS X
- Handles Concurrency
  - Multi-threaded Debugging
  - Multi-process Debugging
- Integrated Memory Debugging
- Reverse Debugging for Linux
- Supports Multiple Usage Models
  - Powerful and Easy GUI – Highly Graphical
  - CLI for Scripting
  - Long Distance Remote Debugging
  - Unattended Batch Debugging
GPU Debugging

with

TotalView
CUDA Port of TotalView

Full visibility of both Linux and GPU threads
- Device threads shown as part of the parent Unix process
- Handles all the differences between the CPU and GPU

Fully represent the hierarchical memory
- Display data at any level (registers, local, block, global or host memory)
- Making it clear where data resides with type qualification

Thread and Block Coordinates
- Built in runtime variables display threads in a warp, block and thread dimensions and indexes
- Displayed on the interface in the status bar, thread tab and stack frame

Device thread control
- Warps advance synchronously
- Handles CUDA function inlining
- Step into or over inlined functions
- Functions show on stack trace

Reports memory access errors
- CUDA memcheck

Multi-Device Support
- Can be used with MPI
Starting TotalView

- Once a new kernel is loaded TotalView provides the option to stop and set breakpoints.
- TotalView automatically configures the GUI for CUDA debugging.
- Debugging CUDA code is done by using normal TotalView commands and procedures.
GPU Device Status Display

Provides the “high-level” view

- Values automatically update as you step through code
- Shows what hardware is in use
- Helps to map between logical and hardware coordinates
GPU Device Status Display

Provides detailed information for:

- **Device and Type**
- **SMs**
- **Warps**
- **Lanes with PC**

Information updates as you step
### GPU Device Status Display

It also provides information for divergent GPU threads.

#### Different PC for two groups of Lanes

#### State of Lanes inside the Warp

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device 0/3</td>
<td></td>
</tr>
<tr>
<td>Device Type</td>
<td>gf100</td>
</tr>
<tr>
<td>Lanes</td>
<td>32</td>
</tr>
<tr>
<td>SM 2/1</td>
<td></td>
</tr>
<tr>
<td>Valid Warps</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>Warp 00/48</td>
<td>Block (0,0,0)</td>
</tr>
<tr>
<td>Lane 00/32</td>
<td>Thread (0,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34d8</td>
</tr>
<tr>
<td>Lane 01/32</td>
<td>Thread (1,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34d8</td>
</tr>
<tr>
<td>Lane 02/32</td>
<td>Thread (2,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 03/32</td>
<td>Thread (3,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 04/32</td>
<td>Thread (4,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 05/32</td>
<td>Thread (5,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 06/32</td>
<td>Thread (6,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 07/32</td>
<td>Thread (7,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 08/32</td>
<td>Thread (8,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Lane 09/32</td>
<td>Thread (9,0,0)</td>
</tr>
<tr>
<td>LPC</td>
<td>0000000019aa34f0</td>
</tr>
<tr>
<td>Valid/Active/Divergent</td>
<td>0000003ff, 000003fc, 00000003</td>
</tr>
<tr>
<td>SM Type</td>
<td>sm_20</td>
</tr>
<tr>
<td>SMs</td>
<td>14</td>
</tr>
<tr>
<td>Warps</td>
<td>48</td>
</tr>
<tr>
<td>Device 1/3</td>
<td></td>
</tr>
<tr>
<td>Device Type</td>
<td>gt200</td>
</tr>
<tr>
<td>Lanes</td>
<td>32</td>
</tr>
<tr>
<td>SM Type</td>
<td>sm_13</td>
</tr>
</tbody>
</table>
Information on GPU execution, location and data is readily available. ... the same as it is for Linux processes and threads.
Debugging CUDA

CUDA grid and block dimensions, lanes/warp, warps/SM,

Parameter, register, local and shared variables
GPU focus thread logical coordinates in the header...
Debugging CUDA

... as well as in the Process Window
Debugging CUDA

PC arrow shows the Program Counter for the warp
Debugging CUDA

Dive on any variable name to open a variable window
Address 0x10 is an offset within parameter storage

“@parameter” type qualifier indicates that variable “A” is in parameter storage

Pointer value 0x110000 is an offset within global storage

“elements” is a pointer to a float in global storage
Storage Qualifiers

- **Denotes location in hierarchical memory**
  - Part of the type – using “@” notation
  - Each memory space has a separate address space so 0x00001234 could refer to several places

- **Used throughout expression system**
  - You can cast to switch between different spaces
Navigate through your CUDA code in the Process Window as you wish... Using either of two coordinate systems:
Debugging CUDA - Navigation

CUDA GPU threads have a negative TotalView thread ID

• Logical: 2 or 3D Grid of Blocks, 3D Thread Within Grid

User-controlled “spinboxes” allow selection and display of any part of your GPU execution

GPU focus thread selector for changing the logical block and thread indexes of the CUDA thread.
Debugging CUDA - Navigation

User-controlled “spinboxes” allow selection and display of any part of your GPU execution.

- GPU focus selector for changing physical indexes of the CUDA thread.
  - Physical: Device, SM, Warp, Lane
• Single-step operation advances all of the GPU hardware threads in the same warp

• To advance the execution of more than one warp:
  – set a breakpoint and continue the process, or
  – select a line number in the source pane and select “Run To”.

• Warps advance synchronously
  – Warps share a PC

• Single stepping
  – Advances the warp containing the focus thread
  – Stepping over a __syncthreads() call advances all the relevant threads

• Continue and runto
  – Continues more than just the warp

• Halt
  – Stops all the host and device threads
CUDA Segmentation Faults

- TotalView displays segmentation faults as expected
  - Enable CUDA memory checking in New Program dialog window
What’s New

Intel Phi Debugging

with

TotalView
A Spectrum of Programming Use Models

Xeon-Centric
- Xeon-native
- Offload

General Purpose Serial and Parallel Codes
- Main()
- MPI_Foo()

Scalar codes with highly parallel phases
- Main()
- MPI_Foo()
- offload<MIC>
- Foo()

Symmetric
- Codes with balanced needs
- Main()
- Foo()

Xeon-native
- Reverse Offload
- Main()
- Foo()

MIC-Centric
- MIC-native
- Offload<Xeon>
- Foo()

MIC-native
- Highly parallel codes
- Main()
- MPI_Foo()
Intel MIC Port of TotalView

- Full visibility of both host and Phi threads
- Full support of MPI programs
- Symmetric Debugging of heterogeneous applications with offloaded code
- Remote Debugging of Phi-native applications
- Asynchronous thread control on both Xeon and Phi
Debugging MPI Applications

- Attach to subset of processes on Phi
- Set breakpoints
- Debug MPI “as usual”
Remote Debugging of Applications on Phi

- Start application on Phi card
- Attach to running application
- See thread private data
- Investigate individual threads
- Kill stuck processes on Phi
Debugging Applications with Offloaded Code

One debugging session for Phi-accelerated code
TotalView provides a full spectrum of debugging solutions

Code debugging
- Highly scalable interactive GUI debugger
  - Easy to use -- without sacrificing detail that users need to debug
  - Used from workstations to the largest supercomputers
- Powerful features for debugging multi-threaded, multi-process, and MPI parallel programs
- Compatible with wide variety of compilers across several platforms and operating systems

Memory Debugging
- Parallel memory analysis and error detection
  - Intuitive for both intensive and infrequent users
  - Easily integrated into the validation process

Reverse Debugging
- Parallel record and deterministic replay within TotalView
  - Users can run their program “backwards” to find bugs
  - Allows straightforward resolution of otherwise stochastic bugs

GPU CUDA Debugging
- Full Hybrid Architecture Support
- Asynchronous Warp Control
- Multi-Device and MPI Support
Thanks!

Thank You

Download a TotalView evaluation at:
www.roguewave.com/products

ehinkel@roguewave.com