GPGPU efforts at Los Alamos, Status and Tool needs
- plus, CBTF for monitoring efforts

David Montoya
Input from: Laura Monroe, Phil Romero, Bob Robey, Scott Pakin, Pat McCormick, Mike Mason

CScADS 2012
6/26/12
Resources

- Testing TLCC2 machines with GPGPUs
- Working out libraries and drivers to support initial production implementation into TOSS
- Working on visualization capability

- DoE "mini-app" developed to facilitate collaboration between hardware and software development

- Tools to identify code transition to GPGPUs
- Compiler Infrastructure research

Visualization clusters
- TLCC2 GPGPU cluster - Recent production addition
- Cray XK6 Development - August target

Sandia’s Exascale Research Cluster
- Research Cluster
  - manycore
  - Dense node communication
  - Access to pre-release HW, MIC, Fusion

CS Research
- HPC Integration
- Apps Team
- Lightshow
- Viewmaster
- Moonlight
- Helios
- Teller
- Darwin
- Intel MIC
Production/Development Machines

**Moonlight** (TLCC2 - first Production Machine)
- Intel Sandy Bridge processor + Nvidia M2090 GPU
- Each CPU has dedicated PCIe link to Tesla M2090
- Peak Compute Performance: 488 TF/s
- Compute nodes: 308
- Compute cores: 4,928 CPU + 315,392 GPU

**Helios** – August (Cray XK6 Development machine)
- 16 blades, 4 nodes per blade = 64 nodes total.
  - 3 blades are I/O blades
  - 13 blades are compute blades
- Compute blade:
  - 4 amd G34 8 core CPU, 32 gb memory per node.
  - 4 NVIDIA 512 core GPUs (Tesla X2090) 6gb built in memory per GPU node.
- Each GPU estimated 1331 Gigaflops peak single precision floating point performance.

**Viewmaster** (Visualization cluster)
- 92 Back-end Rendering Nodes
  - 2 NVIDIA Quadro 6000s (6 GB GDDR5 on-board memory)
  - 96 GB DDR3-1333 memory
- 12 Advanced User Nodes
  - 1 NVIDIA Quadro 6000
  - 96 GB DDR3-1333 memory
- 56 Standard User Nodes
  - 1 NVIDIA Quadro 5000 (2 GB GDDR5 on-board memory)
  - 24 GB DDR3-1333 memory
- 39 Facilities Nodes to support CAVE, PowerWall, etc
  - 1 NVIDIA Quadro 6000, with g-sync daughterboard for frameloact
  - 96 GB DDR3-1333 memory

**CS Testbeds**
- **Darwin**
  - It has 48 AMD Opteron 6168 cores on each node,
    - with some of the nodes having 2 NVIDIA Tesla C2050 GPUs and others with 2 ATI FirePro 7800 GPUs.
- **Intel MIC** – 7 node
- **Access to Sandia’s Exascale Research Cluster**
Software Stack and Testing - Moonlight

GPGPU Software Stack

- Integrated into tri-lab standard TOSS stack
  - Both Moonlight and VM2 are running TOSS
  - Standard release includes CUDA toolkit 4.1, OpenCL
- Investigating further packages
  - OpenACC
  - DSLs like Liszt
- Debuggers: Allinea, TotalView
- User-specified tools: in “experimental” space
- Libraries as needed: Thrust, CUDAMat, CUSP library
- Testing: ORNL’s Scalable Heterogeneous benchmark suite, future development for monitoring and diagnostics

Testing

- Ran Nbody, SHOC test and HPL
- HPL
  - 488 TFLOPS peak
  - 1041 GFLOPS realized on a single node
  - Peak power 937 watts
- Saw several double-bit errors
  - HW replacement on those GPUs

Drivers

- LANL goal: one driver to handle
  - Compute (on Moonlight)
  - GPU rendering, display framelocking (on Viewmaster 2)
- Not there yet
  - Framelocking an ongoing issue
    - There’s a fix in a very recent driver, haven’t tested
    - Earlier drivers showed some rendering performance issues on certain specific use cases
    - Later drivers show better performance
Testing Results – Phil Romero

- GPUs have different performance characteristics than CPUs.
- Performance in transferring data from GPUs to CPUs varies widely even when correct affinities are utilized. Not true when transferring data from CPUs to GPUs.
- GPUs much more sensitive to memory usage on the host than CPUs, sometimes even producing non-linear results on performance.
- CPUs perform in a tight performance range approximating a normal distribution, GPUs do not.
- Three tools that monitor GPU based clusters are now under development.
Moonlight Single Node HPL Performance Distribution by Node

Nodes with slow GPUs
Moonlight Single Node HPL Performance Distribution Utilizing Only CPUs

Utilizing GPUs and CPUs w/o slow node GPUs

Utilizing GPUs and CPUs – w/ slow node GPUs
Monitoring GPUs with CBTF – Phil Romero

Three tools that utilize CBTF (Component Based Tool Framework) and NVML (NVIDIA Management Libraries) are currently under development:

**checkGpuMemory** - finds GPUs that have unrecoverable double-bit ECC errors, and monitors single-bit ECC errors. GPUs with double-bit ECC errors are defective parts that need to be exchanged. The serial number and bus slot are also identified to ensure the correct GPU is identified. Sample output is shown below:

- FAILED Node ml048 Gpu0 2bitMemoryErrors 48 BusId 0000:03:00.0 SerialNumber 0324811024641
- WARNING Node ml159 Gpu1 1bitMemoryErrors 5286 BusId 0000:81:00.0 SerialNumber 0324811023640
- OK Node ml040  [ will be omitted from final output ]

**checkGpuConfigs** - finds GPU configuration outliers. Sample output is shown below:

- 283 Configurations with Vbios 70.10
- 2 Configurations with Vbios 70.09 ml013,ml301
- 284 Configurations with PersistenceMode 1
- 1 Configuration with PersistenceMode 0 ml273

**checkGpuUtilization** - find GPUs whose current utilization state is significantly underperforming with regard to its counterparts. This tool is designed to be run while jobs are in progress, unlike the other tools that run on idle nodes. Several different forms of outputs are possible, for example...

- Identification of statistical outliers:
  - 31 Nodes with GpuPowerUsage mean 895.3 kW
  - 1 Node with GpuPowerUsage mean 413.7 kW ml083

- Continuously polled Graphical output of various GPU, CPU and memory utilizations by node
- Graphical output summarizing various GPU, CPU and memory utilizations by node aggregated over a single job.
- Mixing data from IPMI sources with data from NVIDIA Management Library
LANL Programming Models Research

• Improved compiler infrastructure and supporting toolchain
• Compiler-based Application Analysis – Scott Pakin
The focus is a set of common intermediate representations (IR) to coordinate feedback across the entire toolchain (including runtime components). Overall design matches that of a modern compiler with front-end, middle, and back-end stages.

**Common Representations:** Front-ends map the source language into a language-independent intermediate representation (IR) that is semantically richer (a higher level of abstraction) than is typical practice (e.g., capture concurrency, parallelism, memory locality, scheduling, data layout, etc.)

**Middle:** The middle stage takes the HLIR as input and runs a series of analysis and optimization passes and also provides a natural access point for both static and dynamic performance analysis and debugging tools. HLIR is then lowered to LLIR (low-level IR – using LLVM IR as a building block).

**Back-end:** The back-end includes traditional analysis steps (e.g., data and control flow) as well as lower-level generic optimizations such as dead code elimination and loop transformations. Current work matches closely with LLVM (http://llvm.org).
Example: Supporting Domain Specific Languages

- Co-Design activities with vendors for common LLIR (low-level IR) based on LLVM IR
  - Effort underway with AMD, Intel, NVIDIA)
  - NVIDIA CUDA layers 4.x/5.0 already supported via open-source LLVM release and compiler SDK (beta)
  - AMD and Intel work in progress

- Leverage vendors’ toolchain (reduced effort)
  - Also provides an alternative path for tying into future architectural simulation
Goal: Develop tool to identify functions (or whole applications) that are likely to run well at exascale - and those that need immediate attention.

- Implement a form of software performance counter that measures application characteristic not available at the hardware level. Instrument the code at compile time, adding counters of various operations of interest.
- Byfl – initial ability to measure byte and flop counts – integrates with the LLVM compiler toolchain as a compiler pass.

Modify the compiler to inject instrumentation code. Data are output as easy-to-parse “BYFL”-prefixed lines.
High Performance Computing

KCachegrind view of Byfl measurements

HPCToolkit view of Byfl measurements
Application Team Efforts – Bob Robey

• The Application Team have an shallow-water AMR mini-app with thousands of lines of OpenCL code and dozens of GPU kernels.
  – It also has an MPI layer over the OpenCL to give a hybrid parallel implementation.
  – It is a DoE "mini-app" developed to facilitate collaboration between hardware and software development. It is available at http://www.github.com/losalamos/CLAMR

• The experience developing this mini-app is the basis for the much of the thoughts here.
  – Just starting to look at the performance issues at the MPI/GPU interface.
  – Expect significant performance issues at the interface and will be working on how to improve it.

• Robustness and crash behavior still needs work.
  – Development is in OpenCL so that portability and performance can be assessed on different hardware platforms.
  – This is made difficult when one system doesn't like a few lines of kernel code and we have to hang nodes repeatedly to work out how to recode it. On some hardware, we even have to power cycle the node or computer to fix it. Perhaps an emulator or "safe" operating mode that prevents crashes or hangs and gives better reports on the location of the problem could be developed.
Application Team Efforts – tool needs

• Memory tools for GPU -- to develop an application they need memory tools similar to those developed for the CPU. Tools/functionality that are needed include memory leaks, memory use, out-of-bounds. GPU memory and objects are allocated in a linked-list structure so they can "walk the heap" (ezcl package in CLAMR at github site). But haven't yet implemented the reporting calls. A general package for this functionality would be useful.

• Debugging -- a "mailbox" technique for debugging GPUs is used. (the idea for this technique came from a Stanford graduate student). The process is:
  – Send in an extra array
  – Write debug information to it in the kernel
  – Read back on CPU
  – Print.

It is very effective, but time-consuming to setup. (an aside -- could a similar process be used to get partial timings for kernels).
Update on CBTF deployment – Mike Mason/TJ Machado

• At LANL – assessing use as an administrative tool at large scales and as a component in the monitoring infrastructure
• Component Based Tool Framework (CBTF - a tri-lab effort with Krell Institute) is flexible and general enough to be used for any tool that needs to “do something” on a large number of nodes and filter or collect the results.
• Sysadmin Tools
  – Poll information on a large number of nodes.
  – Run commands or manipulate files on the Backends.
  – Make decisions at the Filter level to reduce output or interaction
• Debugging Tools
  – Use cluster analysis to reduce thousands (or more) processes into a small number of groups
• Monitoring Tool
  – Memory

UNCLASSIFIED LA-UR-12-22355
- PS Tool (tools/contrib/psTool)
  - Run `ps` on all Backends, use filters to identify the procs running on all nodes and what procs are unique to each node.
• TBON-FS implementation (tools/contrib/tbonFS)
  – Perform group file operations on the Backend local file system.
  – Run top on all Backends.
  – Run grep, tail, read or write files on all Backends.
  – Copy a file from NFS to the local /tmp on all Backends.
• Simple memory tool - memTool (tools/contrib/memTool)
  – Displays memory info for each node and total memory used for each MPI process.