GPU clusters for high-performance computing

Conference Paper · January 2009

DOI: 10.1109/CLUSTR.2009.5289128 · Source: DBLP

8 authors, including:

Jeremy Enos
University of Illinois, Urbana-Champaign
11 PUBLICATIONS **252** CITATIONS
SEE PROFILE

John E. Stone
University of Illinois, Urbana-Champaign
62 PUBLICATIONS **3,258** CITATIONS
SEE PROFILE

James C. Phillips
University of Illinois, Urbana-Champaign
78 PUBLICATIONS **16,594** CITATIONS
SEE PROFILE

Wen-mei W. Hwu
University of Illinois, Urbana-Champaign
388 PUBLICATIONS **13,775** CITATIONS
SEE PROFILE

GPU Clusters for High-Performance Computing

Volodymyr V. Kindratenko [#1], Jeremy J. Enos [#1], Guochun Shi [#1], Michael T. Showerman [#1], Galen W. Arnold [#1], John E. Stone [*2], James C. Phillips [*2], Wen-mei Hwu [§3]

[#] *National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign*
1205 West Clark Street, Urbana, IL 61801, USA
[1] {kindr|jenos|gshi|mshow|arnoldg}@ncsa.uiuc.edu

[*] *Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign*
405 North Mathews Avenue, Urbana, IL 61801, USA
[2] {johns|jim}@ks.uiuc.edu

[§] *Coordinated Science Laboratory, University of Illinois at Urbana-Champaign*
1308 West Main Street, Urbana, IL 61801, USA
[3] hwu@crhc.uiuc.edu

Abstract—**Large-scale GPU clusters are gaining popularity in the scientific computing community. However, their deployment and production use are associated with a number of new challenges. In this paper, we present our efforts to address some of the challenges with building and running GPU clusters in HPC environments. We touch upon such issues as balanced cluster architecture, resource sharing in a cluster environment, programming models, and applications for GPU clusters.**

I. Introduction

Commodity graphics processing units (GPUs) have rapidly evolved to become high performance accelerators for data-parallel computing. Modern GPUs contain hundreds of processing units, capable of achieving up to 1 TFLOPS for single-precision (SP) arithmetic, and over 80 GFLOPS for double-precision (DP) calculations. Recent high-performance computing (HPC)-optimized GPUs contain up to 4GB of on-board memory, and are capable of sustaining memory bandwidths exceeding 100GB/sec. The massively parallel hardware architecture and high performance of floating point arithmetic and memory operations on GPUs make them particularly well-suited to many of the same scientific and engineering workloads that occupy HPC clusters, leading to their incorporation as HPC accelerators [1], [2], [4], [5], [10].

Beyond their appeal as cost-effective HPC accelerators, GPUs also have the potential to significantly reduce space, power, and cooling demands, and reduce the number of operating system images that must be managed relative to traditional CPU-only clusters of similar aggregate computational capability. In support of this trend, NVIDIA has begun producing commercially available "Tesla" GPU accelerators tailored for use in HPC clusters. The Tesla GPUs for HPC are available either as standard add-on boards, or in high-density self-contained 1U rack mount cases containing four GPU devices with independent power and cooling, for attachment to rack-mounted HPC nodes that lack adequate internal space, power, or cooling for internal installation.

Although successful use of GPUs as accelerators in large HPC clusters can confer the advantages outlined above, they present a number of new challenges in terms of the application development process, job scheduling and resource management, and security. In this paper, we describe our experiences in deploying two GPU clusters at NCSA, present data on performance and power consumption, and present solutions we developed for hardware reliability testing, security, job scheduling and resource management, and other unique challenges posed by GPU accelerated clusters. We also discuss some of our experiences with current GPU programming toolkits, and their interoperability with other parallel programming APIs such as MPI and Charm++.

II. GPU Cluster Architecture

Several GPU clusters have been deployed in the past decade, see for example installations done by GraphStream, Inc., [3]. However, the majority of them were deployed as visualization systems. Only recently attempts have been made to deploy GPU compute clusters. Two early examples of such installations include a 160-node "DQ" GPU cluster at LANL [4] and a 16-node "QP" GPU cluster at NCSA [5], both based on NVIDIA QuadroPlex technology. The majority of such installations are highly experimental in nature and GPU clusters specifically deployed for production use in HPC environments are still rare.

At NCSA we have deployed two GPU clusters based on the NVIDIA Tesla S1070 Computing System: a 192-node production cluster "Lincoln" [6] and an experimental 32-node cluster "AC" [7], which is an upgrade from our prior QP system [5]. Both clusters went into production in 2009.

There are three principal components used in a GPU cluster: host nodes, GPUs, and interconnect. Since the expectation is for the GPUs to carry out a substantial portion of the calculations, host memory, PCIe bus, and network interconnect performance characteristics need to be matched with the GPU performance in order to maintain a well-balanced system. In particular, high-end GPUs, such as the NVIDIA Tesla, require full-bandwidth PCIe Gen 2 x16 slots that do not degrade to x8 speeds when multiple GPUs are used. Also, InfiniBand QDR interconnect is highly desirable to match the GPU-to-host bandwidth. Host memory also needs to at least match the amount of memory on the GPUs in order to enable their full utilization, and a one-to-one ratio of CPU

cores to GPUs may be desirable from the software development perspective as it greatly simplifies the development of MPI-based applications.

However, in reality these requirements are difficult to meet and issues other than performance considerations, such as system availability, power and mechanical requirements, and cost may become overriding. Both AC and Lincoln systems are examples of such compromises.

A. AC and Lincoln Cluster Nodes

AC cluster host nodes are HP xw9400 workstations, each containing two 2.4 GHz AMD Opteron dual-core 2216 CPUs with 1 MB of L2 cache and 1 GHz HyperTransport link and 8 GB (4x2GB) of DDR2-667 memory per host node (Figure 1, Table I). These hosts have two PCIe Gen 1 x16 slots and a PCIe x8 slot. The x16 slots are used to connect a single Tesla S1070 Computing System (4 GPUs) and x8 slot is used for an InfiniBand QDR adapter. The HP xw9400 host system is not ideal for Tesla S1070 system due to the absence of PCIe Gen 2 interface. Since the AC cluster started as an upgrade to QP cluster, we had no choice but to reuse QP's host workstations.

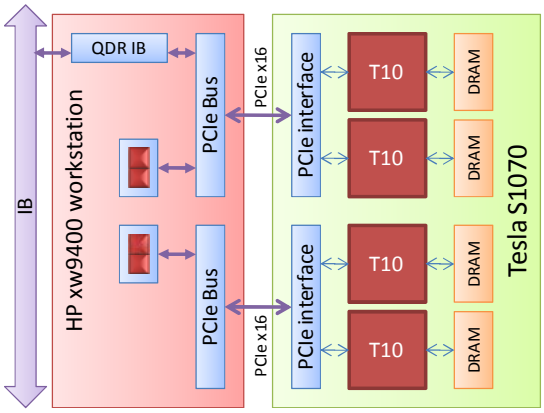

Fig. 1. AC cluster node.

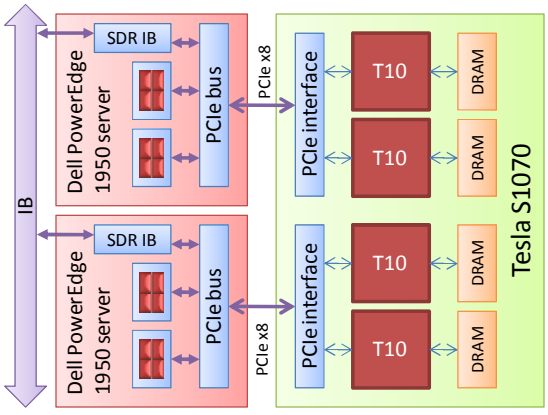

Fig. 2. Two Lincoln cluster nodes share singe Tesla S1070.

Lincoln cluster host nodes are Dell PowerEdge 1950 III servers, each containing two quad-core Intel 2.33 GHz Xeon processors with 12 MB (2x6MB) L2 cache, and 16 GB (4x4GB) of DDR2-667 memory per host node (Figure 2, Table I). The two processor sockets are supported by the Greencreek (5400) chipset, which provides a 1333 MHz independent front side bus (FSB). The hosts have two PCIe Gen 2 x8 slots: One slot is used to connect 2 GPUs from Tesla S1070 Computing System, the other slot is used for an InfiniBand SDR adapter. The Dell PowerEdge 1950 III servers are far from ideal for Tesla S1070 system due to the limited PCIe bus bandwidth. The Lincoln cluster originated as an upgrade to an existing non-GPU cluster "Abe," and thus there was no negotiating room for choosing a different host.

TABLE I
COMPARISON OF AC AND LINCOLN GPU CLUSTERS

	AC	**Lincoln**
CPU Host	HP xw9400	Dell PowerEdge 1950 III
CPU	dual-core 2216 AMD Opteron	quad-core Intel 64 (Harpertown)
CPU frequency (GHz)	2.4	2.33
CPU cores per node	4	8
CPU memory host (GB)	8	16
GPU host (NVIDIA Tesla...	S1070-400 Turn-key	S1070 (a-la-carte)
GPU frequency (GHz)	1.30	1.44
GPU chips per node	4	2
GPU memory per host (GB)	16	8
CPU/GPU ratio	1	4
interconnect	IB QDR	IB SDR
# of cluster nodes	32	192
# of CPU cores	128	1536
# of GPU chips	128	384
# of racks	5	19
Total power rating (kW)	<45	<210

B. Power Consumption on AC

We have conducted a study of power consumption on one of the AC nodes to show the impact of GPUs. Such measurements were only conducted on AC due to its availability for experimentation. Table II provides power consumption measurements made on a single node of the AC cluster. The measurements were made during different usage stages, ranging from the node start-up to an application run. A few observations are particular worthy of discussion. When the system is powered on and all software is loaded, a Tesla S1070's power consumption is around 178 Watt. However, after the first use of the GPUs, its power consumption level doubles and never returns below 365 Watt. We have yet to explain this phenomenon.

To evaluate the GPU power consumption correlated with different kernel workloads, we ran two different tests, one measuring power consumption for a memory-access intensive kernel, and another for a kernel completely dominated by floating-point arithmetic. For the memory-intensive test, we used a memory testing utility we developed [9]. This test repeatedly reads and writes GPU main memory locations with bit patterns checking for uncorrected memory errors. For this memory-intensive test, we observed a maximum power consumption by the Tesla GPUs of 745 Watts. For the

floating-point intensive test case, we used a CUDA single-precision multiply-add benchmark built into VMD [14]. This benchmark executes a long back-to-back sequence of several thousand floating point multiply-add instructions operating entirely out of on-chip registers, with no accesses to main memory except at the very beginning and end of the benchmark. For the floating-point intensive benchmark, we typically observed a Tesla power consumption of 600 Watts. These observations point out that global memory access uses more power than on-chip register or shared memory accesses and floating-point arithmetic—something that has been postulated before, but not demonstrated in practice. We can thus conclude that GPU main memory accesses come not only at the potential cost of kernel performance, but that they also represent a directly measurable cost in terms of increased GPU power consumption.

TABLE II
POWER CONSUMPTION OF A SINGLE NODE ON AC CLUSTER

State	Host Peak (Watt)	Tesla Peak (Watt)	Host power factor (pf)	Tesla power factor (pf)
power off	4	10	.19	.31
start-up	310	182		
pre-GPU use idle	173	178	.98	.96
after NVIDIA driver module unload/reload(1)	173	178	.98	.96
after deviceQuery(2)	173	365	.99	.99
memtest # 10 [9]	269	745	.99	.99
VMD Madd	268	598	.99	.99
after memtest kill (GPU left in bad state)	172	367	.99	.99
after NVIDIA module unload/reload(3)	172	367	.99	.99
NAMD GPU ApoA1 [10]	315	458	.99	.95
NAMD GPU STMV [10]	321	521	.97-1.0	.85-1.0(4)
NAMD CPU only ApoA1	322	365	.99	.99
NAMD CPU only STMV	324	365	.99	.99

(1) Kernel module unload/reload does not increase Tesla power
(2) Any access to Tesla (e.g., deviceQuery) results in doubling power consumption after the application exits
(3) Note that second kernel module unload/reload cycle does not return Tesla power to normal, only a complete reboot can
(4) Power factor stays near one except while load transitions. Range varies with consumption swings

C. Host-Device Bandwidth and Latency

Theoretical bi-directional bandwidth between the GPUs and the host on both Lincoln and AC is 4 GB/s. Figure 3 shows (representative) achievable bandwidth measured on Lincoln and AC nodes. The measurements were done sending data between the (pinned) host and device memories using cudaMemcpy API call and varying packet size from 1 byte to 1 Gbyte. On Lincoln, the sustained bandwidth is 1.5 GB/s. On AC, either 2.6 GB/s or 3.2 GB/s bandwidth is achieved depending on the PCIe interface mapping to the CPU cores: when the data is sent from the memory attached to the same CPU as the PCIe interface, a higher bandwidth is achieved. Latency is typically between 13 and 15 microseconds on both systems. Low bandwidth achieved on Lincoln is subject to further investigation.

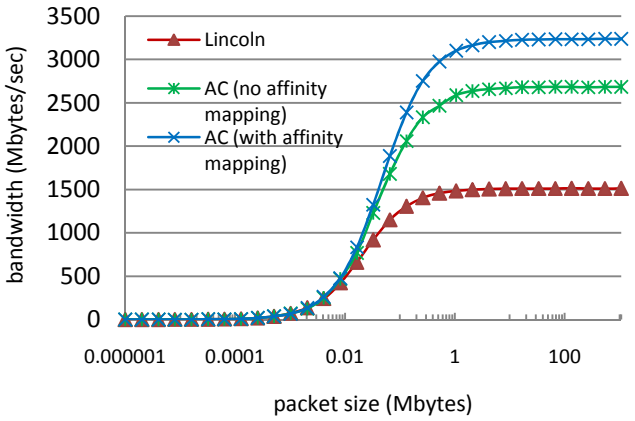

Fig. 3. Host-device bandwidth measurements for AC and Lincoln.

D. HPL Benchmarks for AC and Lincoln

AC's combined peak CPU-GPU single node performance is 349.4 GFLOPS. Lincoln's peak node performance is 247.4 GFLOPS (both numbers are given for double-precision). The actual sustained performance for HPC systems is commonly measured with the help of High-Performance Linpack (HPL) benchmark. Such a benchmark code has recently been ported by NVIDIA to support GPU-based systems [11] using double-precision floating-point data format. Figure 4 shows HPL benchmark values for 1, 2, 4, 8, 16, and 32 nodes for both AC and Lincoln clusters (using pinned memory).

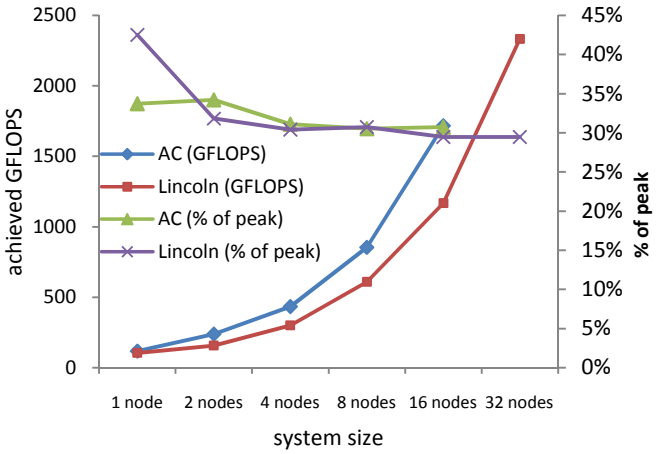

Fig. 4. HPL benchmark values and percentage of the peak system utilization for AC and Lincoln GPU clusters.

On a single node of AC we achieved 117.8 GFLOPS (33.8% of the peak) and on a single node of Lincoln we achieved 105 GFLOPS (42.5% of the peak). This efficiency further drops to ~30% when using multiple nodes. HPL measurements reported for other GPU clusters are in the range of 70-80% of the peak node performance [11]. Results for our clusters are

not surprising since the host nodes used in both AC and Lincoln are not ideal for Tesla S1070 system. Also, further investigation is pending in the libraries used in HPL on AC.

The 16-node AC cluster subset achieves 1.7 TFLOPS and 32-node Lincoln cluster subset achieves 2.3 TFLOPS on the HPL benchmark.

III. GPU Cluster Management Software

The software stack on Lincoln (production) cluster is not different from other Linux clusters. On the other hand, the AC (experimental) cluster has been extensively used as a testbed for developing and evaluating GPU cluster-specific tools that are lacking from NVIDIA and cluster software providers.

A. Resource Allocation for Sharing and Efficient Use

The Torque batch system used on AC considers a CPU core as an allocatable resource, but it has no such awareness for GPUs. We can use the node property feature to allow users to acquire nodes with the desired resources, but this by itself does not prevent users from interfering with each other, e.g., accessing the same GPU, when sharing the same node. In order to provide a truly shared multi-user environment, we wrote a library, called CUDA wrapper [8], that works in sync with the batch system and overrides some CUDA device management API calls to ensure that the users see and have access only to the GPUs allocated to them by the batch system. Since AC has a 1:1 ratio of CPUs to GPUs and since Torque does support CPU resource requests, we allow users to allocate as many GPUs as CPUs requested. Up to 4 users may share an AC node and never "see" each other's GPUs.

The CUDA wrapper library provides two additional features that simplify the use of the GPU cluster. One of them is GPU device virtualization. The user's application sees only the virtual GPU devices, where device0 is rotated to a different physical GPU after any GPU device open call. This is implemented by shifting the virtual GPU to physical GPU mapping by one each time a process is launched. The CUDA/MPI section below discusses how this feature is used in MPI-based applications.

Note that assigning unique GPUs to host threads has been partially addressed in the latest CUDA SDK 2.2 via a tool called System Management Interface which is distributed as part of the Linux driver. In compute-*exclusive* mode, a given thread will *own* a GPU at first use. Additionally, this mode includes the capability to fall back to the next device available.

The other feature implemented in the CUDA wrapper library is the support for NUMA (Non-Uniform Memory Architecture) architecture. On a NUMA-like system, such as AC, different GPUs have better memory bandwidth performance to the host CPU cores depending on what CPU socket the process is running on, as shown in Figure 3. To implement proper affinity mapping, we supply a file on each node containing the optimal mapping of GPU to CPU cores, and extend the CUDA wrapper library to set process affinity for the CPU cores "closer" to the GPU being allocated.

CUDA API calls, such as cudaSetDevice and cuDeviceGet, are overridden by pre-loading the CUDA wrapper library on each node. If needed and allowed by the administrator, the overloaded functions can be turned off by setting up an environment variable.

B. Health Monitoring and Data Security

We discovered that applications that use GPUs can frequently leave GPUs in an unusable state due to a bug in the driver. Driver version 185.08-14 has exhibited this problem. Reloading the kernel module fixes the problem. Thus, prior to node de-allocation we run a post-job node health check to detect GPUs left in unusable state. The test for this is just one of many memory test utilities implemented in the GPU memory test suite [9].

The Linux kernel cannot be depended upon to clean up or secure memory that resides on the GPU board. This poses security vulnerabilities from one user's run to the next. To address this issue, we developed a memory scrubbing utility that we run between jobs on each node. The utility allocates all available GPU device memory and fills it in with a user-supplied pattern.

C. Pre/Post Node Allocation Sequence

The CUDA wrapper, memory test utilities, and memory scrubber are used together as a part of the GPU node pre- and post-allocation procedure as follows:

Pre-job allocation

- detect GPU devices on the allocated node and assemble custom device list file, if not available
- *checkout* requested GPU devices from the device file
- initialize the CUDA wrapper shared memory with unique keys for a user to allow him to ssh to the node outside of the job environment and still see only the allocated GPUs

Post-job de-allocation

- run GPU memory test utility against job's allocated GPU device(s) to verify healthy GPU state
  - if bad state is detected, mark the node offline if other jobs present on it
  - if no other jobs present, reload the kernel module to recover the node and mark it on-line again
- run the memory scrubber to clear GPU device memory
- notify on any failure events with job details
- clear CUDA wrapper shared memory segment
- *check-in* GPUs back to the device file

IV. GPU Cluster Programming

A. GPU Code Development Tools

In order to develop cluster applications that take advantage of GPU accelerators, one must select from one of a number of different GPU programming languages and toolkits. The currently available GPU programming tools can be roughly assigned into three categories:

1) High abstraction subroutine libraries or template libraries that provide commonly used algorithms with auto-

generated or self-contained GPU kernels, e.g., CUBLAS, CUFFT, and CUDPP.

2) Low abstraction lightweight GPU programming toolkits, where developers write GPU kernels entirely by themselves with no automatic code generation, e.g., CUDA and OpenCL.
3) High abstraction compiler-based approaches where GPU kernels are automatically generated by compilers or language runtime systems, through the use of directives, algorithm templates, and sophisticated program analysis techniques, e.g., Portland Group compilers, RapidMind, PyCUDA, Jacket, and HMPP.

For applications that spend the dominant portion of their runtime in standard subroutines, GPU-accelerated versions of popular subroutine libraries are an easy route to increased performance. The key requirement for obtaining effective acceleration from GPU subroutine libraries is minimization of I/O between the host and the GPU. The balance between host-GPU I/O and GPU computation is frequently determined by the size of the problem and the specific sequence of algorithms to be executed on the host and GPU. In addition to this, many GPU subroutine libraries provide both a slower API that performs host-GPU after every call, but is completely backward-compatible with existing CPU subroutine libraries, and a faster incompatible API that maximizes GPU data residency at the cost of requiring changes to the application.

Applications that spend the dominant fraction of runtime in a small number of domain-specific algorithms not found in standard subroutine libraries are often best served by low-abstraction programming toolkits described in category 2. In these cases, the fact that performance-critical code is concentrated into a handful of subroutines makes it feasible to write GPU kernels entirely by hand, potentially using GPU-specific data structures and other optimizations aimed at maximizing acceleration.

Some applications spend their execution time over a large number of domain-specific subroutines, limiting the utility of GPU accelerated versions of standard subroutine libraries, and often making it impractical for developers to develop custom GPU kernels for such a large amount of the application code. In cases like this, compiler-based approaches that use program annotations and compiler directives, and sophisticated source code analysis, may be the only feasible option. This category of tools is in its infancy at present, but could be viewed as similar to the approach taken by current auto-vectorizing compilers, OpenMP, and similar language extensions.

B. CUDA C

Currently, NVIDIA's CUDA toolkit is the most widely used GPU programming toolkit available. It includes a compiler for development of GPU kernels in an extended dialect of C that supports a limited set of features from C++, and eliminates other language features (such as recursive functions) that do not map to GPU hardware capabilities.

The CUDA programming model is focused entirely on data parallelism, and provides convenient lightweight programming abstractions that allow programmers to express kernels in terms of a single thread of execution, which is expanded at runtime to a collection of blocks of tens of threads that cooperate with each other and share resources, which expands further into an aggregate of tens of thousands of such threads running on the entire GPU device. Since CUDA uses language extensions, the work of packing and unpacking GPU kernel parameters and specifying various runtime kernel launch parameters is largely taken care of by the CUDA compiler. This makes the host side of CUDA code relatively uncluttered and easy to read.

The CUDA toolkit provides a variety of synchronous and asynchronous APIs for performing host-GPU I/O, launching kernels, recording events, and overlapping GPU computation and I/O from independent execution streams. When used properly, these APIs allow developers to completely overlap CPU and GPU execution and I/O. Most recently, CUDA has been enhanced with new I/O capabilities that allow host-side memory buffers to be directly shared by multiple GPUs, and allowing zero-copy I/O semantics for GPU kernels that read through host-provided buffers only once during a given GPU kernel execution. These enhancements reduce or eliminate the need for per-GPU host-side I/O buffers.

C. OpenCL

OpenCL is a newly developed industry standard computing library that targets not only GPUs, but also CPUs and potentially other types of accelerator hardware. The OpenCL standard is managed by the Khronos group, who also maintain the OpenGL graphics standard, in cooperation with all of the major CPU, GPU, and accelerator vendors. Unlike CUDA, OpenCL is implemented solely as a library. In practice, this places the burden of packing and unpacking GPU kernel parameters and similar bookkeeping into the hands of the application developer, who must write explicit code for these operations. Since OpenCL is a library, developers do not need to modify their software compilation process or incorporate multiple compilation tools. Another upshot of this approach is that GPU kernels are not compiled in batch mode along with the rest of the application, rather, they are compiled at runtime by the OpenCL library itself. Runtime compilation of OpenCL kernels is accomplished by sending the OpenCL kernel source code as a sequence of strings to an appropriate OpenCL API. Once the OpenCL kernels are compiled, the calling application must manage these kernels through various handles provided by the API. In practice, this involves much more code than in a comparable CUDA-based application, though these operations are fairly simple to manage.

D. PGI x64+GPU Fortran & C99 Compilers

The PGI x86+GPU complier is based on ideas used in OpenMP; that is, a set of newly introduced directives, or pragmas, is used to indicate which sections of the code, most likely data-parallel loops, should be targeted for GPU execution. The directives define kernel regions and describe loop structures and their mapping to GPU thread blocks and threads. The directives are also used to indicate which data need to be copied between the host and GPU memory. The GPU directives are similar in nature to OpenMP pragmas and their use roughly introduces the same level of complexity in

the code and relies on a similar way of thinking for parallelizing the code. The compiler supports C and Fortran.

The PGI x64+GPU compiler analyses program structure and data and splits parts of the application between the CPU and GPU guided by the user-supplied directives. It then generates a unified executable that embeds GPU code and all data handling necessary to orchestrate data movement between various memories. While this approach eliminates the mechanical part of GPU code implementation, e.g., the need for explicit memory copy and writing code in CUDA C, it does little to hide the complexity of GPU kernel parallelization. The user is still left with making all the major decisions as to how the nested loop structure is to be mapped onto the underlying streaming multiprocessors.

E. Combining MPI and CUDA C

Many of the HPC applications have been implemented using MPI for parallelizing the application. The simplest way to start building an MPI application that uses GPU-accelerated kernels is to use NVIDIA's nvcc compiler for compiling everything. The nvcc compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, so it is easier to make MPI code into .cu (since CUDA is a proper superset of C) and compile with nvcc than the other way around. A sample makefile might resemble the one shown in Figure 5. The important point is to resolve the INCLUDE and LIB paths for MPI since by default nvcc only finds the system and CUDA libs and includes.

```
MPICC := nvcc -Xptxas -v
MPI_INCLUDES :=/usr/mpi/intel/mvapich2-1.2p1/include
MPI_LIBS := /usr/mpi/intel/mvapich2-1.2p1/lib

%.o: %.cu
        $(MPICC) -I$(MPI_INCLUDES) -o $@ -c $<

mpi_hello_gpu: vecadd.o mpi_hello_gpu.o
        $(MPICC) -L$(MPI_LIBS) -lmpich -o $@ *.o

clean:
        rm vecadd.o mpi_hello_gpu.o

all: mpi_hello_gpu
```

Fig. 5. MPI/CUDA makefile example.

In one scenario, one could run one MPI thread per GPU, thus ensuring that each MPI thread has access to a unique GPU and does not share it with other threads. On Lincoln this will result in unused CPU cores. In another scenario, one could run one MPI thread per CPU. In this case, on Lincoln multiple MPI threads will end up sharing the same GPUs, potentially oversubscribing the available GPUs. On AC the outcome from both scenarios is the same.

Assigning GPUs to MPI ranks is somewhat less straightforward. On the AC cluster with the CUDA wrapper, cudaSetDevice is intercepted and handled by the wrapper to ensure that the assigned GPU, CPU, PCI bus, and NUMA memory are co-located for optimal performance. This is especially important since the multiple PCI buses on AC can result in CUDA memory bandwidth performance variations. Therefore, users should always call cudaSetDevice(**0**) as this will ensure the proper and unique GPU assignment. On the Lincoln cluster with only one PCI bus, cudaSetDevice is not intercepted and GPUs are assigned by it directly. Thus, it is user's responsibility to keep track of which GPU is assigned to which MPI thread. Since memory bandwidth is uniform on Lincoln with respect to CPU core and the single PCI bus, this does not impact performance as it might on AC. MPI rank modulo number of GPUs per node is useful in determining a unique GPU device id if ranks are packed into nodes and not assigned in round robin fashion. Otherwise there is no simple way to ensure that all MPI threads will not end up using the same GPU.

Some MPI implementations will use locked memory along with CUDA. There is no good convention currently in place to deal with potential resource contention for locked memory between MPI and CUDA. It may make sense to avoid cudaMallocHost and cudaMemcpy*Async in cases where MPI also needs locked memory for buffers. This essentially means one would program for CUDA shared memory instead of pinned/locked memory in scenarios where MPI needs locked memory. Mvapich (for Infiniband clusters) requires some locked memory.

F. Combining Charm++ and CUDA C

Charm++ [15] is a C++-based machine-independent parallel programming system that supports prioritized message-driven execution. In Charm++, the programmer is responsible for over-decomposing the problem into arrays of large numbers of medium-grained objects, which are then mapped to the underlying hardware by the Charm++ runtime's measurement-based load balancer. This model bears some similarity to CUDA, in which medium-grained thread blocks are dynamically mapped to available multiprocessors, except that in CUDA work is performed as a sequence of grids of independent thread blocks executing the same kernel where Charm++ is more flexible.

The preferred mapping of a Charm++ program, such as the molecular dynamics program NAMD, to CUDA, depends on the granularity into which the program has been decomposed. If a single Charm++ object represents enough work to efficiently utilize the GPU on its own, then the serial CPU-based code can be simply replaced with a CUDA kernel invocation. If, as in NAMD [10], the granularity of Charm++ objects maps more closely to a thread block, then additional programmer effort is required to aggregate similar Charm++ object invocations, execute the work on the GPU in bulk, and distribute the results. Charm++ provides periodic callbacks that can be used to poll the GPU for completion, allowing the CPU to be used for other computation or message transfer.

V. Application Experiences

A. TPACF

The two-point angular correlation function (TPACF) application is used in cosmology to study the distribution of objects on the celestial sphere. It is an example of a trivially

parallelizable code that can be easily scaled to run on a cluster using MPI. Its ability to scale is only limited by the size of the dataset used in the analysis. Its computations are confined to a single kernel which we were able to map into an efficient high-performance GPU implementation. We ported TPACF to run on AC's predecessor, QP cluster [5], resulting in nearly linear scaling for up to 50 GPUs [12]. Since there is a 1:1 CPU core to GPU ratio on each cluster node, running the MPI version of the code was straightforward since each MPI process can have an exclusive access to a single CPU core and a single GPU. MPI process ID modulo the number of GPUs per node identifies a unique GPU for each MPI process running on a given node. Speedups achieved on a 12-node QP cluster subset when using 48 Quadro FX 5600 GPUs were ~30× as compared to the performance on the same number of nodes using 48 CPU cores only.

B. NAMD

NAMD is a Charm++-based parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. In adapting NAMD to GPU-accelerated clusters [10], work to be done on the GPU was divided into two stages per timestep. Work that produced results to be sent to other processes was done in the first stage, allowing the resulting communication to overlap with the remaining work. A similar strategy could be adopted in a message-passing model.

The AC cluster, to which NAMD was ported originally, has a one-to-one ratio of CPU cores to GPUs (four each per node). The newer Lincoln cluster has a four-to-one ratio (eight cores and two GPUs per node), which is likely to be typical as CPU core counts increase. Since NAMD does significant work on the CPU and does not employ loop-level multithreading such as OpenMP, one process is used per CPU core, and hence four processes share each GPU.

CUDA does not time-slice or space-share the GPU between clients, but runs each grid of blocks to completion before moving on to the next. The scheme for selecting which client to service next is not specified, but appears to follow a round-robin or fair-share strategy. A weakness of the current system is that if each client must execute multiple grids before obtaining useful results, then it would be more efficient to service a single client until a data transfer to the host is scheduled before moving to the next client. This could be incorporated into the CUDA programming model by supporting grid "convoys" to be executed consecutively or through explicit yield points in asynchronous CUDA streams.

The two stages of GPU work in NAMD create further difficulties when sharing GPUs, since all processes sharing a GPU should complete the first stage (work resulting in communication) before any second-stage work (results only used locally) starts. This is currently enforced by code in NAMD that infers which processes share a GPU and uses explicit messages to coordinate exclusive access to the GPU in turns. If CUDA used simple first-in first-out scheduling it would be possible to simply submit work to the GPU for one stage and yield to the next process immediately, but each process must in fact wait for its own work to complete, which results in unnecessary GPU idle time and is less efficient than uncoordinated GPU access when running on few nodes. An alternative strategy would be to have a single process access each GPU while other processes perform only CPU-bound work, or to use multithreaded Charm++ and funnel all CUDA calls through a single thread per GPU.

GPU-accelerated NAMD runs 7.1 times faster on an AC node when compared a CPU-only quad-core version running on the same node. Based on the power consumption measurements shown in Table II, we can calculate the performance/watt ratio, normalized relative to the CPU-only run as follows: (CPU-only power)/(CPU+GPU power)*(GPU speedup). Thus, for NAMD GPU STMV run, the performance/watt ratio is 324 / (321+521) * 7.1 = 2.73×. In other words, NAMD GPU implementation is 2.73 times more power-efficient than the CPU-only version.

C. DSCF

Code that implements the Direct Self-Consistent Field (DSCF) method for energy calculations has been recently implemented to run on a GPU-based system [13]. We have parallelized the code to execute on a GPU cluster.

There are two main components in DSCF: computation of J and K matrices and linear algebra for post-processing. The computation of J and K matrices is implemented on the GPU whereas the linear algebra is implemented on the CPU using ScaLapack from Intel MKL. Our cluster parallelization of the code is as follows: For each cluster node, we start N processes, where N is the number of CPU cores in the node. This is required for an efficient ScaLapack use. All of the MPI processes are initially put into sleep except one. The active MPI process spawns M pthreads, where M is the number of GPUs in the node. All the GPUs in the cluster will compute their contribution to the J and K matrices. The computed contributions are communicated and summed in a binary tree fashion. Once J and K are computed, all sleeping MPI processes are wakened and the completed matrices are distributed among them in block-cyclic way as required by the ScaLapack. All the MPI processes are then used to compute linear algebra functions to finish post-processing. This process is repeated until the density matrix converges.

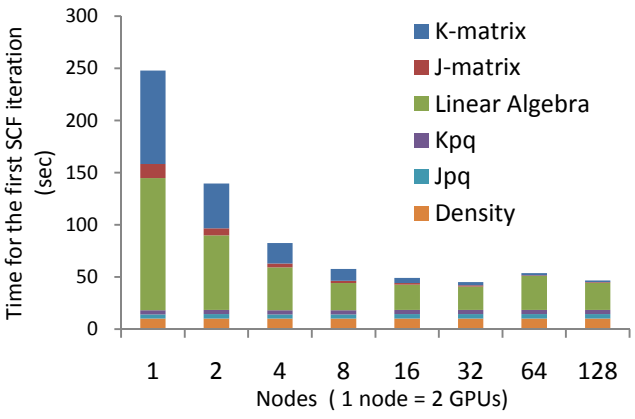

Fig. 6. DSCF scalability study on Lincoln.

Our results on Lincoln show the J and K matrices computed on the GPUs are scaling very well while the linear algebra part executed on the CPU cores is scaling not as well (Figure 6). This is not surprising since the GPUs do not require communication when computing their contributions to the matrices, but ScaLapack requires a lot of data exchanges among all the nodes.

VI. Discussion and Conclusions

Our long-term interest is to see utilities that we developed to be pulled in or re-implemented by NVIDIA as part of the CUDA SDK. One example of this is the virtual device mapping rotation, implemented to allow common device targeting parameters to be used with multiple processes on a single host, while running kernels on different GPU devices behind the scenes. This feature was first implemented in the CUDA wrapper library. Starting from CUDA 2.2, NVIDIA provides the compute-exclusive mode which includes device fall back capability that effectively implements the device mapping rotation feature of the CUDA wrapper library. We hope that other features we implement (optimal NUMA affinity, controlled device allocation on multi-user systems, and memory security) will eventually be re-implemented in NVIDIA's tool set as well.

Although we have not fully evaluated OpenCL for use in HPC cluster applications, we expect that our experiences will be similar to our findings for CUDA. Since the OpenCL standard does not currently require implementation to provide special device allocation modes, such as CUDA "exclusive" mode, nor automatic fall-back or other features previously described for CUDA, it is likely that there will be a need to create an OpenCL wrapper library similar to the one we developed for CUDA, implementing exactly the same device virtualization, rotation, and NUMA optimizations. Since the OpenCL specification does not specify whether GPU device memory is cleared between accesses by different processes, it is also likely that the same inter-job memory clearing tools will be required for OpenCL-based applications.

While we are glad to see NVIDIA's two new compute modes, normal and compute-exclusive, we do not think that NVIDIA's implementation is complete. Normal mode allows devices to be shared serially between threads, with no fall back feature should multiple GPU devices be available. Compute-exclusive mode allows for fall back to another device, but only if another thread does not already own it, whether it is actively using it or not. What is needed for optimal utilization is an additional mode, combining both the device fall back and shared use aspects, where a new thread will be assigned to the least subscribed GPU available. This would allow for the possibility to oversubscribe the GPU devices in a balanced manner, removing that burden from each application.

With the continuing increase in the number of cores with each CPU generation, there will be a significant need for efficient mechanisms for sharing GPUs among multiple cores, particularly for legacy MPI applications that do not use a hybrid of shared-memory and message passing techniques within a node. This capability could be provided by a more sophisticated grid or block scheduler within the GPU hardware itself, allowing truly concurrent kernel execution from multiple clients, or alternately with software approaches such as the "convoy" coordination scheme mentioned previously, that give the application more control over scheduling granularity and fairness.

Acknowledgment

The AC and Lincoln clusters were built with support from the NSF HEC Core Facilities (SCI 05-25308) program along with generous donations of hardware from NVIDIA. NAMD work was supported in part by the National Institutes of Health under grant P41-RR05969. The DSCF work was sponsored by the National Science Foundation grant CHE-06-26354.

References

[1] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stove, "GPU Cluster for High Performance Computing,", *in Proc. ACM/IEEE conference on Supercomputing*, 2004.

[2] H, Takizawa and H. Kobayashi, "Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing," J. Supercomput., vol. 36, pp. 219--234, 2006.

[3] (2009) GraphStream, Inc. website. [Online]. Available: http://www.graphstream.com/.

[4] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Tureka, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," *Parallel Computing*, vol. 33, pp. 685-699, Nov 2007.

[5] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W. Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster," *in Proc. 10th LCI International Conference on High-Performance Clustered Computing*, 2009. [Online]. Available: http://www.ncsa.illinois.edu/~kindr/papers/lci09_paper.pdf

[6] (2008) Intel 64 Tesla Linux Cluster Lincoln webpage. [Online] Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/

[7] (2009) Accelerator Cluster webpage. [Online]. Available: http://iacat.illinois.edu/resources/cluster/

[8] (2009) cuda_wrapper project at SourceForge website. [Online]. Available: http://sourceforge.net/projects/cudawrapper/

[9] G. Shi, J. Enos, M. Showerman, V. Kindratenko, "On Testing GPU Memory for Hard and Soft Errors," *in Proc. Symposium on Application Accelerators in High-Performance Computing*, 2009.

[10] J. Phillips, J. Stone, and K. Schulten,. "Adapting a message-driven parallel application to GPU-accelerated clusters," *in Proc. 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[11] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," *in Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[12] D. Roeh, V. Kindratenko, R. Brunner, "Accelerating Cosmological Data Analysis with Graphics Processors," *in Proc. 2nd Workshop on General-Purpose Computation on Graphics Processing Units*, 2009.

[13] I. Ufimtsev and T. Martinez, "Quantum Chemistry on Graphical Processing Units. 2. Direct Self-Consistent-Field Implementation," *J. Chem. Theory Comput.*, vol. 5, pp 1004–1015, 2009.

[14] W. Humphrey, A. Dalke, and K. Schulten, "VMD - Visual Molecular Dynamics," *Journal of Molecular Graphics*, vo. 14, pp. 33-38, 1996.

[15] (2009) Charm++, webpage. [Online]. Available: http://charm.cs.illinois.edu/