Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability

Atsushi Kawai, Kenji Yasuoka
Department of Mechanical Engineering,
Keio University
Yokohama, Japan
Email: kawai@kfcr.jp, yasuoka@mech.keio.ac.jp

Kazuyuki Yoshikawa, Tetsu Narumi
Department of Informatics and Engineering,
University of Electro-Communications
Tokyo, Japan
Email: kaz82019@jed.uec.ac.jp, narumi@cs.uec.ac.jp

***Abstract*—One of the difficulties for current GPGPU (General-Purpose computing on Graphics Processing Units) users is writing code to use multiple GPUs. One limiting factor is that only a few GPUs can be attached to a PC, which means that MPI (Message Passing Interface) would be a common tool to use tens or more GPUs. However, an MPI-based parallel code is sometimes complicated compared with a serial one. In this paper, we propose DS-CUDA (Distributed-Shared Compute Unified Device Architecture), a middleware to simplify the development of code that uses multiple GPUs distributed on a network. DS-CUDA provides a global view of GPUs at the source-code level. It virtualizes a cluster of GPU equipped PCs to seem like a single PC with many GPUs. Also, it provides automated redundant calculation mechanism to enhance the reliability of GPUs. The performance of Monte Carlo and many-body simulations are measured on 22-node (64-GPU) fraction of the TSUBAME 2.0 supercomputer. The results indicate that DS-CUDA is a practical solution to use tens or more GPUs.**

***Keywords*-GPGPU; CUDA; distributed shared system; virtualization.**

I. INTRODUCTION

Optimization of communication among several hundreds of thousands of CPU cores is one of the main concerns in high performance computing. The largest supercomputer [1] has nearly a million cores, on which the communication tends to be the bottleneck instead of the computation.

On modern massively parallel systems, several (typically 4–16) CPU cores in one processor node share the same memory device. The design of a program should take this memory hierarchy into account. A naive design that assigns one MPI (Message Passing Interface) process to each core causes unnecessary communication among cores in the same node. In order to avoid this inefficiency, a hybrid of MPI and OpenMP is often used, where each OpenMP thread is assigned to one core, and one MPI process is used per node.

Moreover, a significant fraction of recent top machines in the TOP500 list [2] utilize GPUs. For example, the TSUBAME 2.0 supercomputer [3] consists of 1,408 nodes, each containing 3 NVIDIA GPUs. In order to program GPUs, frameworks such as CUDA [4] or OpenCL [5] are necessary. Therefore, a program on massively parallel systems with GPUs needs to be written using at least three frameworks, namely, MPI, OpenMP, and CUDA (or OpenCL).

However, even a complicated program using all three frameworks may fail to take full advantage of all the CPU cores. For example, if one thread is assigned to a core to control each GPU, only a few cores per PC would be in use, since only a few GPUs can be attached to a PC. There are typically more cores than GPUs on a single node, and utilizing these remaining cores is also important.

We propose a Distributed-Shared CUDA (DS-CUDA) framework to solve the major difficulties in programming multi-node heterogeneous computers. DS-CUDA virtualizes all GPUs on a distributed network as if they were attached to a single node. This significantly simplifies the programming of multi-GPU applications.

Another issue that DS-CUDA addresses is reliability of GPUs. Consumer GPUs such as GeForce sometimes are prone to memory errors due to the lack of ECC (Error Check and Correct) functions. Hamada *et al.* [6] reported around a 10% failure rate for one week of execution on GeForce GTX 295 cards. Furthermore, even with server-class GPUs, erroneous code may cause faulty execution of successive parts of the program, which is difficult to debug on a multi-GPU environment. DS-CUDA increases the reliability of GPUs by a built-in redundancy mechanism.

The virtualization of multi-GPUs in DS-CUDA also alleviates the increased burden to manage heterogeneous hardware systems. For example, some nodes in a GPU cluster might have a different number of GPUs than others, or even no GPUs. With DS-CUDA the user no longer needs to worry about the number of GPUs on each node.

A key concept of DS-CUDA is to provide a global view of GPUs for CUDA based programs. Global view of distributed memory is one of the key features of next generation languages, such as Chapel [7] and X10 [8]. These languages greatly reduce the complexity of the program compared to MPI and OpenMP hybrid implementations. However, they do not provide the global view on GPUs, since GPUs can only be accessed through dedicated APIs such as CUDA and OpenCL.

 ISBN: 978-1-61208-217-2

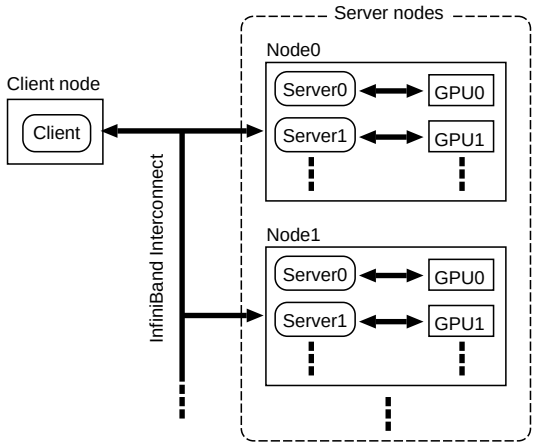

Figure 1. The structure of a typical DS-CUDA system. The client program using CUDA runs on a client node, while DS-CUDA server programs run on each server node for each GPU.

Similar ideas for virtualizing GPUs have been implemeted in rCUDA [9][10], vCUDA [11], gVirtuS [12], GViM [13], and MGP [14]. However, vCUDA, gVirtuS and GViM virtualize the GPUs to enable the access from a virtual machine in the same box, and they do not target remote GPUs. MGP is a middleware to run OpenCL programs on remote GPUs. Their idea is similar to ours, but they only support OpenCL, and not CUDA. rCUDA is also a middleware to virtualize remote GPUs, and it supports CUDA. In this sense, rCUDA is quite similar to DS-CUDA. The primary difference between rCUDA and DS-CUDA is that the former aims to reduce the number of GPUs in a cluster for lowering construction and maintenance cost, while the latter aims to provide a simple and reliable solution to use as many GPUs as possible. To this end, DS-CUDA incorporates a fault tolerant mechanism, that can perform redundant calculations by using multiple GPUs. Errors are detected by comparing the results from multiple GPUs. When an error is detected, it automatically recovers from the error by repeating the previous CUDA API and kernel calls until the results match. This fault tolerant function is hidden from the user.

In this paper, we propose a DS-CUDA middleware. In Section II, the design and implementation are explained. In Section III, the performance measured on up to 64 GPUs is shown. In Section IV, conclusions and future work are described.

II. Implementation

In this section, we describe the design and implementation of DS-CUDA.

A. System Structure

The structure of a typical DS-CUDA system is depicted in Figure 1. It consists of a single client node and multiple server nodes, connected via InfiniBand network.

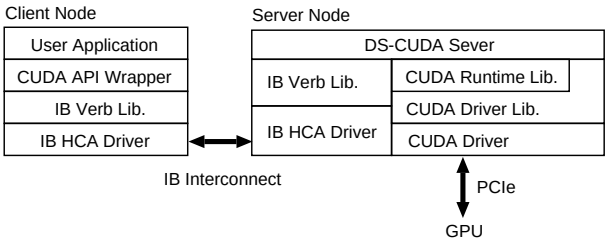

Figure 2. The software-layer stack. On the client node, the user application program calls CUDA API wrappers instead of native CUDA APIs. On server node, DS-CUDA server calls actual cuda APIs. Communication between client and server nodes are performed by InfiniBand Verbs by default.

Each server node has one or more CUDA devices (*i.e.*, GPUs), each of which is handled by a server process. An application running on the client node can utilize these devices by communicating with the server node over the network.

B. Software-Layer Stack

Figure 2 shows the software-layer stack of both the client and server node. On the client node, the application program is linked to the DS-CUDA client library. The library provides CUDA API wrappers, in which the procedure to communicate with the servers are included. Therefore, the CUDA devices on the server nodes can be accessed via usual CUDA APIs, as if they were locally installed.

By default, the client-server communication uses InfiniBand Verbs, but can also use TCP sockets in case the network infrastructure does not support InfiniBand.

C. CUDA C/C++ Extensions

Access to CUDA devices are usually done through CUDA API calls, such as `cudaMalloc()` and `cudaMemcpy()`. As mentioned above, these are replaced with calls to CUDA API wrappers, which communicate with the devices on the server nodes.

There are, however, several exceptions. Some CUDA C/C++ extensions, including calls to CUDA kernels using triple angle brackets, *e.g.*, `myKernel<<<g,b>>>(val,...)`, access the CUDA devices without explicit calls to CUDA APIs.

The DS-CUDA preprocessor, `dscudacpp` handles CUDA C/C++ extensions. Figure 3 summarizes the procedure: In order to build an executable for the application program, the source codes are fed to `dscudacpp`, instead of usual `nvcc`. The sources are scanned by `dscudacpp`, and the CUDA C/C++ extensions are replaced with remote calls which load and launch the kernel on the server side. Then, it passes the modified sources on to the C compiler `cc`. Meanwhile, `dscudacpp` retrieves the definitions of kernel functions and passes them on to `nvcc`. The kernel functions are compiled by `nvcc`, and kernel modules are

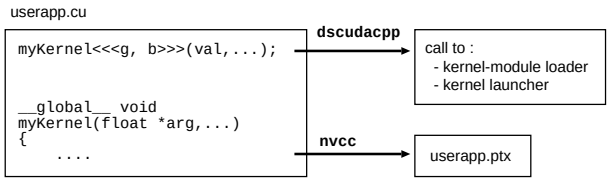

Figure 3. CUDA C/C++ Extensions are preprocessed by `dscudacpp`. GPU kernel `myKernel` is compiled by native `nvcc` compiler and also converted by `dscudacpp` to calls to the kernel-module loader and kernel launcher.

generated in `.ptx` format. During the execution of the application program, the kernel modules in the client node are transferred to the server nodes upon request.

D. Virtual Device Configuration

The application program sees virtual devices that represent real devices on the server nodes via the CUDA API wrapper. The mapping of the real to virtual devices is given by an environment variable `DSCUDA_SERVER`. Table I gives an example of the mapping.

```
sh> export DSCUDA_SERVER= \
    "node0:0 node0:1,node1:0 node1:1"
```

Device0 on Server Node0 is mapped to the virtual Device0, Device1 of Node0 and Device0 of Node1 are mapped to virtual Device1, and so on. Note that two devices, Device1 of Node0 and Device0 of Node1, are mapped to a single virtual Device1, that represents a 2-way *redundant device*. The mechanism of redundant devices will be described in the next section.

E. Fault-Tolerant Mechanism

A virtual device can have redundancy, in order to improve reliability of the calculations performed on the device. That is, multiple CUDA devices on the server nodes can be assigned to a single virtual device on the client node. Identical calculations are performed on the redundant devices, and the results are compared between the redundant calculations. If any of the results do not match, the client library invokes an error handler.

By default, the handler tries to recover from the error. It reproduces all CUDA API calls after the latest successful call to `cudaMemcpy()` of transfer type `cudaMemcpyDeviceToHost`. The application program may override this behavior, if it is not desirable.

F. Functional Restrictions

Although DS-CUDA provides transparent access to CUDA devices over the network, its function is not fully compatible with the native CUDA framework. The current version has the following restrictions:

(a) The graphics relevant APIs, such as OpenGL and Direct3D interoperability, are not supported.

Table I
AN EXAMPLE OF A MAPPING OF REAL DEVICES ON THE SERVER NODES TO VIRTUAL DEVICES ON THE CLIENT NODE.

Client-side virtual device	Server-side real device
Device 0	Device 0 of node0
Device 1	Device 1 of node0 and device 0 of node1 (2-way redundancy)
Device 2	Device 1 of node1

(b) Only CUDA Toolkit 4.0 is supported.
(c) Some capabilities to set memory attributes, including page lock and write combining, are not supported.
(d) Asynchronous APIs are implemented as aliases to their synchronous counterparts.
(e) Only the CUDA Runtime APIs, a part of CUFFT and a part of CUBLAS are supported. The CUDA Driver APIs are not supported.

The graphics APIs listed in (a) are meaningless for remote devices, whose video outputs are not used. Rest of the restricted capabilities, (b)–(e), are planned to be supported in the near future.

III. MEASURED PERFORMANCE

In this section, we show performances of the DS-CUDA system measured on a fraction of the TSUBAME 2.0 [3] GPU cluster. The fractional system consists of 22 nodes, each houses two Intel Xeon processors (X5670) and three NVIDIA Tesla GPUs (M2050, x16 Gen2 PCI Express, 64Gbps).

In Section III-A, some measurements on our prototype GPU cluster are also shown. The cluster consists of 8 nodes, each houses an Intel Core i5 processor (i5-2500) and an NVIDIA GeForce GPU (GTX 580, x16 Gen1 PCI Express, 32Gbps).

In the both systems, the nodes are connected via Infini-Band (X4 QDR, 40Gbps) network.

A. Data Transfer

Here, we show data transfer speeds of the `cudaMemcpy()` wrapper function. Curves labeled "IB Verb" in Figure 4 show the results. The left and right panels are for our prototype system and the TSUBAME 2.0, respectively. For comparison, the speeds of native `cudaMemcpy()` are also shown as "local" curves.

On the both systems, the effective bandwidth is around 1GB/s for large enough data size ($\geq$ 100kB). This speed is marginal for some productive runs as shown in later sections and [9]. Also, we should note that we still have room for improvement in the effective bandwidth. In [10], the effective bandwidth of nearly 3GB/s is reported. The high data bandwidth is achieved by eliminating the main memory to main memory copy using *GPU direct* [15] technology. Furthermore, they overlapped the network transfer and memory copy.

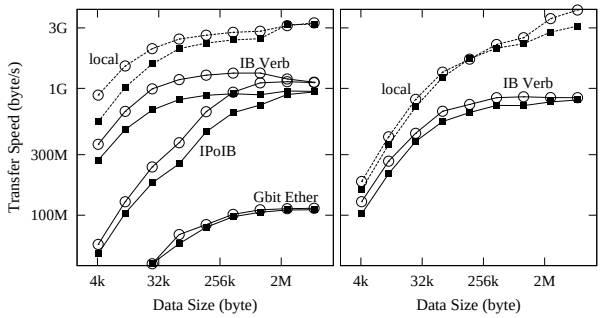

Figure 4. Performance of the `cudaMemcpy()` wrapper function. Data transfer speeds are plotted against data payload size. The left panel shows the results on the prototype system, and the right shows that on the TSUBAME 2.0. Curves with open circles are results for transfer from the client to the server (transfer type `cudaMemcpyHostToDevice`), those with filled squares are from the server to the client (`cudaMemcpyDeviceToHost`).

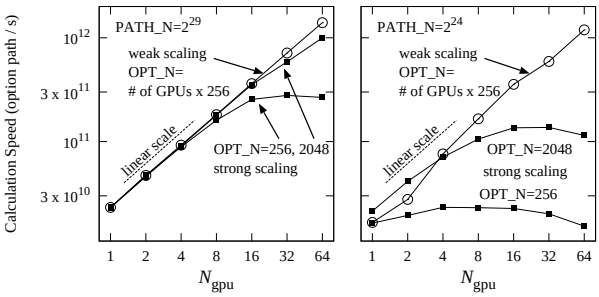

Figure 5. Performance of `MonteCarloMultiGPU`, an implementation of the Monte Carlo approach to option pricing, included in the CUDA SDK. Calculation speeds are plotted against the number of GPU devices, $N_{\rm gpu}$. The number of options calculated for is denoted by `OPT_N`. The number of integration paths is denoted by `PATH_N`. Weak scaling graphs show good scalability for both cases, but for strong scaling with smaller calculation cost shown in right panel is worse.

Poor performance at smaller data size ($\leq$ 100kB) is observed on the TSUBAME 2.0. We observe performance degradation for "local" curves, too. This is likely to be caused not by the network latency, but by the memory-access latency inside the server node.

Curves labeled "IPoIB" and "Gbit Ether" in the left panel are results with data transfer using the TCP socket over InfiniBand and over Gigabit Ethernet, respectively. These are shown just for comparison.

B. MonteCarloMultiGPU

Next, we show the performance of `MonteCarloMultiGPU`, an implementation of the Monte Carlo approach to option pricing, which is included in the CUDA SDK. In the source code, the number of options calculated for is given by a parameter `OPT_N`, and the number of integration paths is given by `PATH_N`. We measured the calculation speed for various combinations of `OPT_N`, `PATH_N` and the number of CUDA devices, $N_{\rm gpu}$, involved in the simulation.

Figure 5 shows the results. Calculation speeds (defined as the total number of paths processed per second) are plotted against $N_{\rm gpu}$. The left and right panels are for `PATH_N` $= 2^{29}$ and `PATH_N` $= 2^{24}$, respectively. Three curves in each panel are for runs with `OPT_N` fixed to 256 and 2048 in order to see strong scaling, and those with `OPT_N` scaled by $N_{\rm gpu}$ to see weak scaling.

In the left panel, the result shows 95% weak scaling. Although strong scaling goes down to 68% (`OPT_N`=2048) and 18% (`OPT_N`=256) at runs with 64 devices, they show better scalability for runs with smaller $N_{\rm gpu}$.

The results in the right panel also show ideal weak scalability. Strong scalings are, however, lower than 10% even with `OPT_N`=2048. In this case, runs with more than a few devices are not practical.

C. Many-Body Simulation

Now we show the performance of the simplest gravitational many-body simulation. In the simulation, gravitational forces from all point masses (hereafter we denote them "j-particle") are evaluated at the location of all point masses ("i-particle") by a naive $O(N^2)$ algorithm. In parallel runs, i-particles are split into fractions, and each of them are sent to one CUDA device, while all j-particles are sent to all devices.

Figure 6 shows the results. Calculation speeds (defined as the number of pairwise interactions between two particles calculated per second) are plotted against the number of devices, $N_{\rm gpu}$. The results with $N = 128$k shows fairly good scalability for up to 8 devices. Those with $N \leq 32$k scales only up to a few devices, in which case the locally installed 1–4 devices would be a better choice than DS-CUDA.

We should note that in production runs, fast algorithms such as the Barnes-Hut treecode [16] of $O(N \log N)$ and the FMM [17] of $O(N)$ are often used, whose performance exceed that of $O(N^2)$ algorithm at large N, say $\geq 10^5$. According to our preliminary result with a serial treecode, reasonable balance is achieved when one CUDA device is assigned to one CPU core. In that case the workload on the device is roughly equivalent to that of the $O(N^2)$ algorithm with $N/N_{\rm gpu}$ i-particles and 10k j-particles. Using the $O(N^2)$ parallel code, we measured the performance at that workload and found it scales well at least up to $N_{\rm gpu} = 8$.

D. Molecular Dynamics Simulation with Redundancy

We performed a molecular dynamics simulation of a NaCl system. Figure 7 shows the temperature against time. We used 512 atoms, and Tosi-Fumi potential [18] is used for the interaction between atoms. The initial temperature is set to 300 K and atom positions are integrated with a Leap-Frog method with a time-step of 0.5 fs for 2 ps (40,000

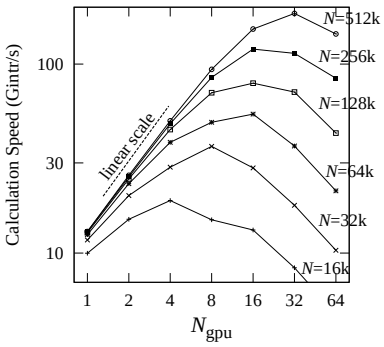

Figure 6. Performance of gravitational many-body simulations. The number of pairwise interactions between two particles calculated per second is plotted against the number of CUDA devices, N_{gpu}, for various number of particles in the system, N.

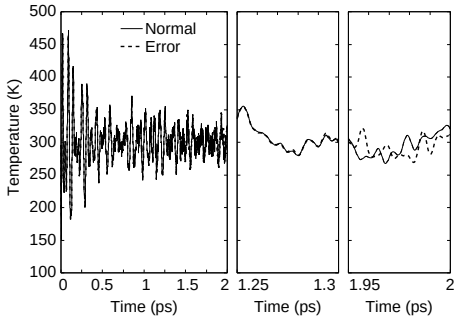

Figure 7. Temperature of a system from a molecular dynamics simulation is plotted against time (pico second). Middle and right panels are a close-up of the left one. Solid curves (`Normal`) show the correct result, while dashed curves (`Error`) show the result including errors.

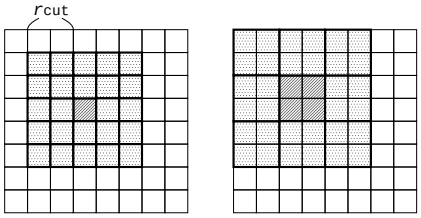

Figure 8. Communication region in the cell-index method. The left panel shows the communication without DS-CUDA, while the right panel shows the communication with DS-CUDA. The thick hatched region is the part a node has to manage, and thin hatched region is the region that it has to communicate among surrounding nodes. The granularity of communication and calculation becomes larger when DS-CUDA is used. Therefore, both the communication and calculation would be accelerated.

steps). Solid curves show the correct result, while dashed curves show the result including artificial errors. The error is artificially generated with a special DS-CUDA server. It randomly injects a bit error every 6 Mbyte in the data transferred from the server to the client. Using this technique we are able to emulate a faulty GPU. As shown in the right panel in Figure 7, the error causes different behavior of the temperature. Note that a single bit error may cause different results after a long simulation. If a bit error is critical, the simulation may stop immediately.

In our prototype system, we constructed a 2-way redundant device, that consists of a normal DS-CUDA server and an error-prone one. When we performed the simulation using our redundant device, we were able to obtain the correct results. The point is that the application program is not changed at all, and reliable calculation with redundant operation is achieved with the DS-CUDA system automatically.

IV. Conclusion and Future Work

We proposed DS-CUDA, a middleware to virtualize GPU clusters as a distributed shared GPU system. It simplifies the development of code on multiple GPUs on a distributed memory system.

Performances of two applications were measured on a 64-GPU system, where good scalability is confirmed. Also, the usefulness of the redundant calculation mechanism is shown, which distinguishes this work from other related work.

In the following part, we will discuss some issues under investigation.

A. Hybrid Programs with DS-CUDA

In some applications, time spent on the CPU cores, or communication between the cluster nodes and GPUs, can be the bottleneck of the overall calculation time. In such cases, DS-CUDA alone cannot offer much improvement in speed. However, combining with MPI parallelization, it may offer better performance than what can be achieved by MPI only. In the following, cell-index method [19] is discussed as example of such an application.

The cell-index method is a method to reduce the calculation cost of many-body interactions, such as gravity, Coulomb, and van der Waals forces. In this method, a cutoff length of interaction between two particles, r_{cut} is defined, and particles do not interact beyond this length. When interactions are calculated on a cluster of nodes, spatial domain decomposition is used and communication is needed only among neighboring nodes.

Consider a simulation space that is composed of $8 \times 8 \times 8 = 512$ cells and the cutoff length r_{cut} is double the cell size. The simulation is performed using a cluster of 512 nodes, each equipped with one GPU, and one MPI process is running on each cluster node.

A node is responsible for calculation of forces on the particles inside one cell. Each node requires data from the surrounding 124 (=$5^3 - 1$) cells to perform the calculation (recall that r_{cut} is double of the cell size). The left panel of Figure 8 illustrates a two dimensional slice of the simulation space. In order to calculate the forces on particles inside the thick-hatched cell, data must be transferred from the thin-

hatched cells.

For example, if 8 nodes are handled by one DS-CUDA node, the corresponding 8 cells, instead of a cell, are taken care by one MPI process with 8 GPUs. In this case, the number of MPI processes in total is reduced to 64, and each has to communicate with 26 ($= 3^3 - 1$) nodes, as shown in the right panel of Figure 8.

By using DS-CUDA, the performance and the programmability are improved in the following sense: (1) the number of MPI process is reduced to 1/8; (2) the total amount of communication among cluster nodes are reduced. In the right panel of Figure 8, the thick-hatched 8 cells communicate with 208 ($= 6^3 - 2^3$) cells. These 8 cells need to communicate with 992 ($= 8 \times (5^3 - 1)$) cells, if DS-CUDA is not used; (3) load imbalance among MPI processes becomes smaller, since the number of particles handled by one node increases 8 times on average.

B. DS-CUDA as a Cloud

A GPU cluster virtualized using DS-CUDA can be seen as a cloud, that offers flexibility, power saving, and reliability. The flexibility means that an arbitrary amount of GPU resource can be derived on request. The power saving means that some part of the cluster nodes can be suspended while there are no running jobs on the GPU. The reliability means that calculation errors can be recovered by the automated redundancy mechanism described in Section II-E.

If we could implement a function to dynamically migrate GPU resources between different nodes, and combining it with the redundant mechanism, a fault-tolerant system can be constructed. For example, if an unrecoverable error occurred on a server node, the malfunctioning node could automatically be retired and a new one could be joined, without stopping the simulation.

ACKNOWLEDGMENT

This research was supported in part by the Japan Science and Technology Agency (JST) Core Research of Evolutional Science and Technology (CREST) research programs "Highly Productive, High Performance Application Frameworks for Post Petascale Computing", and in part by Venture Business Promotion program of University of Electro-Communications. Authors thank Dr. Rio Yokota for his support on refining the manuscript.

REFERENCES

[1] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe, "The K Computer: Japanese Next-Generation Supercomputer Development Project " in *International Symposium on Low Power Electronics and Design (ISLPED) 2011*, 2011, pp. 371–372.

[2] TOP500 Supercomputer Sites. Available: http://www.top500.org/ [retrieved: June, 2012]

[3] TSUBAME 2.0 Supercomputer. Available: http://www.gsic.titech.ac.jp/en/tsubame2 [retrieved: June, 2012]

[4] The NVIDIA website. Available: http://www.nvidia.com/object/cuda_home_new.html [retrieved: June, 2012]

[5] The KHRONOS website. Available: http://www.khronos.org/opencl/ [retrieved: June, 2012]

[6] T. Hamada, R. Yokota, K. Nitadori, T. Narumi, K. Yasuoka, M. Taiji, and K. Oguri, "42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence" in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2009*, 2009, USB memory.

[7] Cray Inc. The Chapel Parallel Programming Language. Available: http://chapel.cray.com/ [retrieved: June, 2012]

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an Object-Oriented Approach to Non-Uniform Cluster Computing" in *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, 2005, pp. 519–538.

[9] J. Duato, A. J. Pẽna, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters" in *2011 IEEE International Conference on Parallel Processing*, 2011, pp. 365–374.

[10] R. M. Gual, "rCUDA:an approach to provide remote access to GPU computational power" in *HPC Advisory Council European Workshop 2011*, Available: http://www.hpcadvisorycouncil.com/events/2011/european_workshop/pdf/3_jaume.pdf [retrieved: June, 2012]

[11] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines" *IEEE Transactions on Computers*, vol. PP, 2011, pp. 1–13.

[12] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds", in *Euro-Par 2010 - Parallel Processing, ser. LNCS*, vol. 6271, 2010, pp. 379–391.

[13] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "GViM: GPU-accelerated virtual machines", in *3rd Workshop on System-level Virtualization for High Performance Computing*, 2009, pp. 17–24.

[14] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices" in *Workshop on Parallel Programming and Applications on Accelerator Clusters*, 2010.

[15] The NVIDIA website. Available: http://developer.nvidia.com/gpudirect [retrieved: June, 2012]

[16] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm" *Nature*, vol. 324, 1986, pp.446–449.

[17] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations" *J. Comput. Phys.*, vol 73, 1987, pp.325–348.

[18] M. P. Tosi, and F. G. Fumi, "Ionic sizes and born repulsive parameters in the NaCl-type alkali halides–II: The generalized Huggins-Mayer form" *J. Phys. Chem. Solids*, vol. 25, 1964, pp 45–52.

[19] R. W. Hockney, J. W. Eastwood, Computer Simulation Using Particles. New York, McGraw-Hill, 1981.

 ISBN: 978-1-61208-217-2