Architecture of Request Distributor for GPU Clusters

Mani Shafaat Doost, S. Masoud Sadjadi
School of Computing and Information Sciences
Florida International University
11200 SW 8th St ECS 212 Miami, FL 33199 - USA
{*mshaf012,sadjadi*}*@cis.fiu.edu*

Jose Ricardo da S. Jr., Marcelo Zamith, Mark Joselli, Esteban Clua
MediaLab
Universidade Federal Fluminense
Rua Passo da Patria, 156 - Niterói, Rio de Janeiro - Brazil
{*jricardo,mzamith, mjoselli, esteban*}*@ic.uff.br*

***Abstract*—The advent of GPU computing has enabled development of many strategies for accelerating different kinds of simulations. Even further, instead of processing an application by just using one GPU, it is a common to use a collection of GPUs as a solution. These GPUs can be located in the same machine, network, or even across a wide area network. Unfortunately, distribution and management of GPUs requires additional efforts by the user such as deal with data transfer, connection and processing among GPUs.**

***Request distributor for GPU clusters (RDGPUC)* is a software architecture which allows companies, institutes and other users to share their GPU resources. By using this architecture, each cluster can have its own software to manage internal resources and they only need to develop small code to interact with RDGPUC. This novel design brings flexibility to the system and allows everyone to share their resources without need to change their GPU cluster tool. Another interesting part of system is to allow users to submit requests from all kind of devices and platforms. Admin of this system is able to specify resource groups and special schedules for using resources. On the other hand, end-users can just use a simple interface to submit their requests on RDGPUC without knowing about internal design and current status of GPU clusters.**

***Keywords*-cluster computing; GPU; load balance; resource sharing.**

I. Introduction

Graphics Processing Unit (GPU) computing has become an important choice for many parallel computational problems. GPUs are potentially more powerful than CPUs for processing massively parallel data. The reason behind this discrepancy is in floating-point capability of GPUs. CPU is specialized for compute-intensive, highly parallel computation, but GPU is typically has used for graphics rendering. Therefore, GPU architecture is designed in such a way that more transistors are devoted to data processing than data caching and flow control. Many different non-graphical computation, simulation and numerical problems, including Protein Structure Prediction [1], Solution of Linear Equation Systems [2], Fluid Simulation [3] and others, have been solved in GPUs.

Nowadays, many GPU Computing systems are starting to have multiple GPU devices to solve problems [4]. In order to distribute the workload across multiple GPUs, developer must manage data exchange between main memory and these devices, guaranteeing consistency between the multiple copies of data and making the development for these architectures is more difficult for the developer.

In order to use GPU resources effectively and avoid of resource wasting it is better to share resources. There are different mechanisms for sharing resources, it is possible to share one GPU among multiuser by installing a hypervisor on the GPU or just share multiple GPUs among multiple users. Since most GPU programs are computation intensive [5] we are going to implement a method which can share multiple resources among multiple users.

In this paper we present a novel architecture to expose GPU computation for anyone who is interested in solving problems or simply study them. By using RDGPUC, user can automatically scale its problem to runs on many GPUs. But before that, user needs to model his/her problem in order to fit into multiples GPUs.

The remainder of this paper is organized as follows. After referring to related works on resource sharing, in Section II, we present the *Request Distributor for GPU Clusters (RDGPUC)* architecture for automatically manages request on the collection of available GPUs, in Section III. In Section IV we present a study case that could benefit from this architecture in simulating 3D acoustic waves. Finally, in Section V we present conclusions of the paper.

II. Related works

In order to decrease the required time to process large data information, researches are laying on using multiples GPUs. Fan et. al. [6] proposed an architecture of a GPU cluster and demonstrate it feasibility by a flow simulation using the Boltzmann model with 30 GPUs node.

Hartley et. al [7] use a cluster of collaborate GPUs and CPUs in order to analyze biomedical images, obtaining an almost linear speed-up using a both of them in this heterogeneous architecture.

Abdelkhalek et. al. [8] designed a parallel simulator in order to solve the acoustic wave equation on a GPU cluster, using a finite difference approach in both 2D and 3D cases. In this simulation, up to 8 GPUs is used to make this simulation.

978-0-7695-4916-3/12 $26.00 © 2012 IEEE
DOI 10.1109/WAMCA.2012.15

Virtual Computing Lab (VCL) is an application which used for sharing computing resources among multiple users [9]. VCL allow users to reserve one or more images (operation system with specific software) and use these resources during the time of reservation. VCL architecture consists of simple user interface, database and management node. However, VCL shares physical resource among multiple virtual machines, but VCL does not divide an application among multiple servers.

Method of sharing is the main difference between VCL and RDGPUC. VCL shares each physical machine among multiple virtual machines at the same time and it doe not know anything about the context of each virtual machine. But RDGPUC shares multiple physical machines for running one specific application on them. RDGPUC does not know about the context of the application, but it needs to know about run-time environment and number of required devices for distribute this application among multiple GPUs.

III. Architecture of Request Distributor for GPU cluster

RDGPUC has been designed to solve problems related to collaboration among GPU clusters. It is responsible to receive requests from user interface and distribute requests among one or more GPU clusters. When GPU clusters finish processing one request, RDGPUC will notify user of the result. Request in RDGPUC can be in variant forms. But in general, one request consists of one or more kernel codes which need to run on GPUs. User can submit kernel codes individually or add a CPU/GPU code together.

In our design, we tried to make a generic design for managing GPU clusters. Since there are many companies, universities and other institutes who are using GPU clusters and each of them have their own regulation and their own software to manage GPU cluster, in our design we make a standard interface for communication with RDGPUC.

This application consists of the modules which are shown in Figure 1. Each module is responsible for one general task. But it is possible for individual users to add customized module to the system. In the following sections we are going to explain each component independently and then we will describe functionalities of this application.

A. User Interface

In order to make a generic solution, it is better to use a web-based interface for this module. But since this project consists of modules which have predefined protocols to communicate to the other modules, we can have more than one User Interfaces. For example, it is possible to add a module which handles requests from Smartphone.

In general, User Interface is responsible to accept requests from user and put it in the database. Also, it needs to have other components which allow user to customize his/her information, review submitted request and view results.

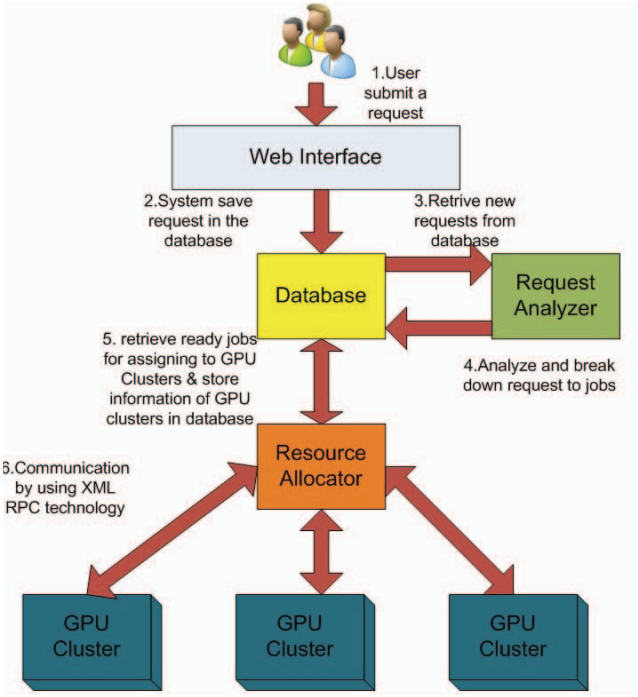

Figure 1. RDGPUC data workflow.

Submitting a request can be done in many forms. It is possible for user to submit the whole code, kernel code or executable file. If user decides to submit executable file, then he/she must specify the required platform for running the file. But if user submit the kernel code or whole source code, then Analyzer can review the code an analyze it (described in section 4.3) which helps to find appropriate resource for processing this request.

In addition to the code or executable file, user needs to submit input and output files. Usually in GPU programming, application works with massive datasets [10], [11], allowing at least two general options for input/output files. First is to upload and download files from web interface which consumes a lot of spaces of database server. And also, this solution makes system much slower and consumes network bandwidth for uploading and downloading files and it suffers from single point of failure.

The second solution is submitting access link and authentication method to the website. In this solution, user does not need to upload and download files to/from the server and he/she just submits addresses of files and authentication to access those files. In this case, GPU clusters are responsible to download files from the source and upload results back to the specified address.

B. Database

The database stores users, resources and requests temporary information. Administrator is able to add users and

resources to the database. When user submits a request from User Interface, it will go directly to the database, then Request Analyzer retrieve information of each request and after analyzing, it will save the result as one or more jobs to the database. Resource Allocator receives information from resources and also retrieves information of resources and jobs from database, then decide to assign one or many jobs to each resource. When GPU resources finish processing one job, they will store the results in the database.

C. Request Analyzer

Sometimes, one request consists of many sub-requests and Analyzer is responsible for analyzing each request to determine sub-requests, estimate execution time of each request and put this information into the database. We save sub-requests as a job which is ready to send to GPU clusters. Also, analyzer can make history of resources, requests and users. By making history of each user, it is easier to estimate execution time of requests and with information from resources; it is easier to assign each request to appropriate resource.

D. Resource allocator

The Resource allocator is responsible for making communication between GPU clusters and RDGPUC. It provides standard interface for GPU clusters which makes it easier to send and receive messages to/from RDGPUC. Depend on different kind of GPU clusters, Resource Allocator can send request to GPU cluster for processing or receive a message from GPU cluster of availability of resources. Also, this component is responsible for receiving result of process from GPU clusters. Also, it is possible to implement other component which can analyze the current situation of resources and develop different algorithms for resource allocation.

E. GPU clusters

GPU clusters are responsible for executing jobs and this is the endpoint of our design. As shown in Figure 2, each GPU cluster has its own management node which is responsible for communication between Resource Allocator and GPU resources and also managing GPU resources. For communication between GPU clusters and Resource allocator, we use XML-RPC which is easy to implement and it is reusable in all system. Each GPU cluster can have its own management system which allows GPU clusters to hide their sensitive information from our application. But GPU clusters need to implement one interface for their software to make a communication to XML-RPC server in Resource Allocator.

Another responsibility of Management node is managing internal resources on each GPU cluster. As shown in Figure 2, one easy way to manage Resources is to add an agent on each GPU resource and then management node manages

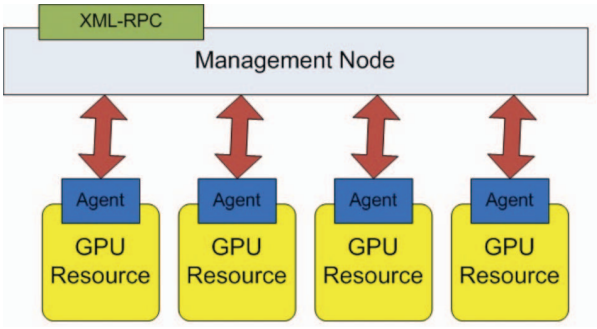

Figure 2. GPU cluster's node communication.

resources by sending messages through to these agents. Agents are responsible for compiling the code, retrieving input/output file, running the application, returning results and also monitoring resources. Depend on requirements of individual GPU cluster, it is possible to define more functionalities.

F. Functionality

There are many functionality which has been defined for each part. In the following paragraphs, we briefly describe these functionalities.

- **Add a Request:** User login and add a request to the system. Request can be submitted in many forms. The User must upload source code, executable file or kernel code plus input and output files. Another method is to allow user to just give a link of input, output and credentials to access to these files (this may reduce network overhead). After that the system stores the request at the database.
- **Show Request List:** In this part, system shows information of all submitted request of specific user to him/her. This information consists of request date, name, status and etc. By entering each of these requests, the system will show more details of request or the result of the request as an output file.
- **Show Result:** The system shows detailed information of the request and if the result of request or some part of it is ready, then system allow user to have access to the output.
- **Edit Profile:** The system allows user to change the information of him/her. In this part the user can change name, password, email and etc.
- **Manage Resources:** In this part, administrator can change the status of resources, add a new resource or remove current resources. In order to add a new resource, the administrator needs to insert all the information of the new GPU resource.
- **Manage Users:** In this part, the administrator can add new users, remove current users and change user privileges.

- **Manage User group:** The administrator can add each user to specific user group and also map each user group to one or more resource group. By using this method he can manage users to have access to specific kind of resources.
- **Manage Resource group:** The administrator can add each resource to one or more resource group and manage a group of resources together. He can specify a schedule for availability of each resource group. Also, it is possible to add special features for each resource group.
- **Manage Schedules:** In this part, the administrator can make a schedule of availability of a resource group.
- **Analyze request:** The Request Analyzer retrieves requests from the database and analyze them by using different methods. It may break down one request to many smaller requests or make a profile from each request and then store the result of analyzing at the database as multiple jobs.
- **Fetch Request:** Management Nodes in each GPU cluster may send a message to the resource allocator and ask for a request, then the Resource Allocator will send a request to the management node for processing.
- **Allocate Request:** there are certain resources which give access to resource allocator to send requests to them. The Resource Allocator analyzes the situation of resources and sends the request to one resource for execution.
- **Collect Result:** When one GPU cluster finishes a request, it should send result to the resource allocator and resource allocator is responsible to receive the result and put it in to the database.

IV. Study case: acoustic wave simulation

In order to illustrate the importance of this architecture, we present a study case that demonstrates a parallel idea for scattering of 2-D acoustic waves in semi-infinite non-homogeneous medium on heterogeneous cluster based GPUs. For more information about Physics aspects of acoustic simulation, please refer to [12].

Hybrid-cluster is composed of two parts. The first is responsible for simulating the numerical method, i.e., calculating the value of each domain point. This is done by GPUs using the CUDA language. The second is responsible for guaranteeing the correctness of communication among the nodes and requires synchronization. Thus, for each time instant, GPUs calculate unknown in values int the simulation domain and then the communication phase takes place. After that, each node sends required values to other nodes for the next instant time, as the domain where the simulation is performed is divided among the available GPUs.

In order to calculate each value in the domain through finite difference method on each GPU, it is required the border data copy among its neighborhood [13], as can be seen in Figure 3. This way, each GPU kernel is responsible to process its border at each time step and send them to its neighborhood. Without using any architecture, additionally to fit the model in a parallel manner, the management of data transfer among GPUs also is the user's responsibility.

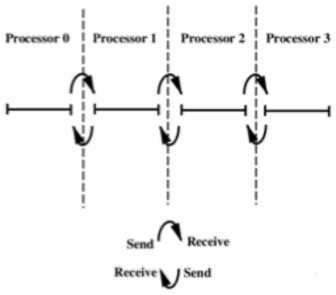

Figure 3. Information sharing among GPUs.

Using our proposed architecture, the user only needs to model its problem to fit among the requested GPUs, being the architecture responsible to make data transfer and maintain the connection among these requested GPUs. Figure 4 illustrate the steps necessary to perform the simulation using our novel architecture.

In the first step, after logging in the system, the user must submit its code by using the **Web Interface** component. Before submitting its code, the user also must specify how many GPUs it wants to allocate to process its application.

After this step the user does not need to do anything, only to wait the result of its program execution. Depending on the demand of processes to be executed, the **Resource Allocator** is able to assign its process to be executed at the same time of its submission, or schedule it to be processed when resources are available.

Internally, the **Resource Allocator** dispatches the application to be executed by one or more **GPU cluster**, through a message to its **Management Node**. The **Management Node** is responsible to allocate GPU resources in this cluster and guarantee communication between its own resources and others **Management Node**, which frees the user to manages it by itself.

As is possible to see in Figure 4, each **GPU Resource** is managed by an **Agent**, which is responsible to process GPU kernels and store the state of each GPU in the cluster. In this architecture, the **Management Node** only communicates with a GPU resource by its **Agent** through a defined protocol. As can be observed, for this specific problem, border data transfer is made by **Agent** communication, leaving the **Management Node** frees for managing other applications that may be allocated in its cluster.

V. Conclusions

In this paper, we have discussed about an architecture which can be used to manage many GPU clusters. Each

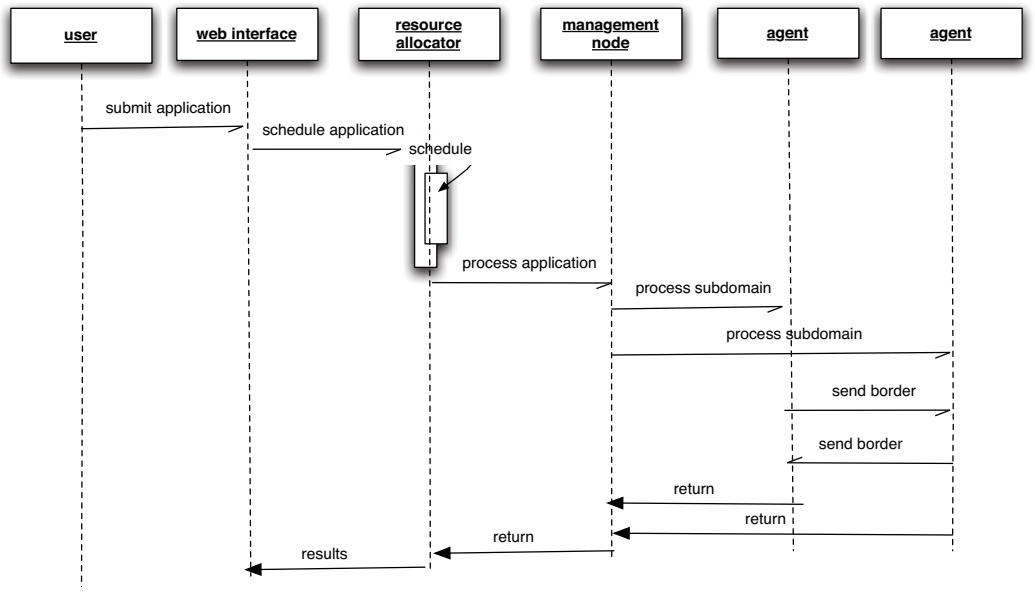

Figure 4. Sequence diagram for the acoustic wave simulation using the proposed architecture.

GPU cluster can have its own software for managing GPU resources and communicate with RDGPUC by using common XML-RPC interface. So, it is easier for institutes to collaborate in this project without sharing information of their resources. Administrator needs to add resources, users, resource groups and user groups to the system. User groups and resource groups bring control over resources. Each resource group associated to one schedule which shows availability of resources during the time. Each user groups can be assigned to one or many resource group and it means that requests from this user group will go directly to particular resource group.

As shown in the paper, this software consists of many modules and for individuals it is easy to add a new module to extend the software for special requirements of themselves. Each module is responsible for one special task but it is possible to extend modules or add new modules, in order to satisfy special requirements. Different kind of user-interfaces can be developed for special devices. Artificial intelligent algorithms can be used in analyzer to estimate execute time more accurately. And also it is possible to add more features on management node of each GPU cluster.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. OISE-0730065.

REFERENCES

[1] W. B. Langdon and W. Banzhaf, "A simd interpreter for genetic programming on gpu graphics cards," in *Proceedings of the 11th European conference on Genetic programming*, ser. EuroGP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 73–85. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792694.1792702

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schrder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," *ACM Trans. Graph*, vol. 22, pp. 917–924, 2003.

[3] J. Ricardo da Silva Junior, E. W. Gonzalez Clua, A. Montenegro, M. Lage, M. d. A. Dreux, M. Joselli, P. A. Pagliosa, and C. L. Kuryla, "A heterogeneous system based on GPU and multi-core CPU for real-time fluid and rigid body simulation," *International Journal of Computational Fluid Dynamics*, vol. 26, no. 3, pp. 193–204, 2012.

[4] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941591

[5] N. Corporation, "Nvidia cuda programming guide," 2011.

[6] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 47–. [Online]. Available: http://dx.doi.org/10.1109/SC.2004.26

[7] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of gpus and multicores," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 15–25. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375533

[8] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a gpu cluster," in *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, june 2009, pp. 36 –43.

[9] S. Averitt, M. Bugaev, A. Peeler, H. Shaffer, E. Sills, S. Stein, J. Thompson, and M. Vouk, "Virtual computing laboratory (vcl)," in *Proceedings of International Conference on Virtual Computing Initiative*, may 2007, pp. 1 –16.

[10] D. Cederman and P. Tsigas, "On sorting and load balancing on gpus," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 11–18, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1556444.1556447

[11] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 325–336. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142511

[12] D. Micha and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010. [Online]. Available: http://dx.doi.org/10.1111/j.1365-246X.2010.04616.x

[13] D. Brandao, M. Zamith, E. Clua, A. Montenegro, A. Bulcao, D. Madeira, M. Kischinhevsky, and R. C. Leal-Toledo, "Performance evaluation of optimized implementations of finite difference method for wave propagation problems on gpu architecture," *Computer Architecture and High Performance Computing Workshops, International Symposium on*, vol. 0, pp. 7–12, 2010.