A Dynamic Load-Balancing Algorithm for Heterogeneous GPU Clusters

Luis Sant'Ana, *CMCC-UFABC* Daniel Cordeiro, *IME-USP* and Raphael Camargo, *CMCC-UFABC*

***Abstract*—The use of GPU clusters for scientific applications is becoming more widespread, with applications in areas such as physics, chemistry and bioinformatics. But due to frequent architectural changes, these clusters are often heterogeneous, with different types of GPUs and CPUs. To use these machines in an efficient manner, it is necessary to perform load-balancing among the GPUs and CPUs, minimizing the execution time of the application. We propose an algorithm for dynamic load balancing in heterogeneous GPU cluster. We implemented the algorithm in the StarPU framework and compared with existing load-balancing algorithms.**

***Keywords*—parallel computing, distribuited systems, GPU cluster, GPGPU.**

I. Introduction

The use of GPUs (Graphics Processing Units) has become popular among developers of high-performance applications that can benefit from a large degree of parallelism [1]. Modern GPUs can have thousands of simple floating point units (FPUs) that, combined, can be several times faster than tradicional CPUs. But it has a limited amount of cache memory and control logic, making it more difficult to develop applications that run efficiently on these units.

Many applications already use the processing power of GPUs. For example, applications in fluid mechanics [2], visualization science [3], machine learning [4], bioinformatics [5] and neural networks [6]. However, many complex problems require the usage of multiple GPUs from GPU clusters [7], [8].

Developing applications for GPU clusters is challenging, since the developer need to manage multiple memories spaces, for each GPU and computer in the cluster. This includes transferring data between these spaces of memories and ensure the consistency of data. A combination of CUDA (Compute Unified Device Architecture) and MPI (Message Passing Interface) is a commom choice to develop applications for GPU clusters. There are efforts by the scientific community for the creation of new programming models [9], [10] for the development of efficient applications on clusters GPUs [11], [12], [13].

GPU clusters are typically homogeneous, which facilitates the development of applications, since they must be optimized for a single architecture, and the load distribution among GPUs. But homogeneity is difficult to achieve in an area where a new generation of hardware are launched every couple of years, which caused the appearance of heterogeneous GPUs clusters.

Developing a load-balancing mechanism that works efficiently for any application is difficult. With two or more GPUs, this problem is strictly equivalent to the classic problem of minimizing the maximum completion time of all tasks (makespan), which is known to be NP-hard [14], but we can restrict the mechanism for some applications classes. For example, we can consider only data-parallel applications that can be divided using domain decomposition [15]. In this case, the data can be distributed between GPUs available and the main task of the load-balancing mechanism is determining the data division amont the GPUs. Many scientific applications fit into this group, including many applications in bioinformatics [5], neural networks [6], chemistry, physics and materials science.

However, with heterogeneous groups of GPUs, this distribution is more difficult. There are currently several types of GPUs with different architectures, such as *Tesla*, *Fermi*, *Kepler* and *Maxwell*. These architectures have different organizations of FPUs (Floating-point Unit), cache, shared memory and memory speeds. A division of the load based on simple heuristics, such as a the number of cores in the GPU, is not effective and can be worse than a plain division [7]. Also, different architectures require different optimizations and a code can be better optimized for one architecture than the other. Finally, GPUs clusters normally have high-end CPUs in addition to the GPUs, and these CPUs may be used by the applications, for instance, to execute parts of the code which cannot be coded efficiently using the Single Instruction Multiple Thread (SIMT) model of GPUs.

The usage of execution time profiles for each GPU type and application task can be used to determine the amount of work given to each GPU. This profiling can be done statically [7] or dynamically [16], [17]. Another solution is to use simple algorithm for task dispatching, such as the greedy algorithm from StarPU [13], where tasks are dispatched to the devices as soon as they become available.

In this work we propose an algorithm for dynamic load balancing in heterogeneous GPU clusters. The algorithm adjusts the size of blocks dynamically, depending on the characteristics of the processing units. Based on runtime measures, it creates a execution model using a Gauss-Newton method and determine a weight parameter for each GPU and CPU. We compared the algorithm with a static one, which determine the block sizes before the execution, a greedy algorithm, that delivers tasks to every available device in order of appearance, and the HDSS [17], which is dynamic load-balancing algorithm.

II. Related Work

A common solution for load balancing in distributed systems is to assign blocks according to a weight factor representing the processing speed of each processor. Early approaches used fixed weight factors determined at compile time with limited

success [18]. But these weight factors are difficult to determine in heterogneous GPUs, due to the architectural differences that affect the performance.

The problem of load balancing in heterogeneous GPUs, began to be studied recently. Acosta *et al.* [16] proposed a dynamic load balancing algorithm that iteractively try finds a good distribution of work between GPUs during the execution of the application. A library focuses on differences in times of parallel codes, based on an iterative scheme. It is a decentralized scheme in which synchronizations are done at each iteration to determine whether there is a need to rebalance the load. Each processor performs a redistribution of workload according to the size of the assigned task and the capabilities of the assigned processor. Load balancing is obtained by comparing the execution time of the current task for each processor with the redistribution of the subsequent task, if time reaches a threshold the load redistribution is made. Our algorithm is different because it creates curves for each processing unit, and based on these curves performs the load balancing.

A static algorithm that determines the distribution before starting the execution of the application, using the static profiles from previous executions, was also proposed [7]. The algorithm was evaluated using a large-scale neural network simulation. The algorithm finds the distribution of data that minimizes the execution time of the application, based on profiles from previous executions, ensuring that all GPUs to spend same amount of time performing the processing of kernels. But this approach does not allow dynamic changes in the data distribution.

The Heterogeneous Dynamic Self-Scheduler (HDSS) [17] provides a dynamic load-balancing algorithm, divided in two phases. The first is the adaptive phase, where it determines weights that reflect the speed of each processor. These weights are determined based on fits of logarithmic curves on processing speed graphs. These weights are used during the completion phase, where it divides the remaining iterations among the GPUs based on the relative weights of each GPU. The main differences to our algorithm is that we do not restrict the execution models to logarithm curves, which are appropriate for GPUs but not or CPUs in the range of interest. Also, our algorithm can perform adjustments until the end of the execution.

III. Proposed Algorithm

Overview. In a typical data-parallel application, the application data is divided among the threads in a process called domain decomposition [15]. The threads then simultaneously process their part of the data. After finishing, the threads merge the processed results, and the application terminates or continues to the next phase of computing. The task of our load-balancing algorithm is determining the size of the data block assigned to each GPU and CPU in the system. We will use the term processor to mean a single CPU or GPU.

Our algorithm determines the optimal block size for each processor using an iterative process. We initially determine a function $P_p[x]$, which provides the processing speed for a block of data of size x on processor p. The objective of the algorithm is to minimize the total time of the application, by distributing the input data among the processors. The problem can be described as follows:

Suppose we have n processors and a input data size Z. We consider that the function of processor g has an input of size x^g, corresponding a fraction of input data assigned to processor g. In this case, we have $\sum_{g=1}^{n} x^g = Z$.

We denote as $E^g(x^g)$ the execution time of function in the processor g, for input x^g. The proposed algorithm consists in fiding the set values:

$$X = \{x^g \in \mathbb{R} : [0,1] / \sum_{g=1}^{n} x^g = Z\} \tag{1}$$

where $g = 1, ..., n$, that minimizes E^A while satisfying the constraint:

$$E_A = E_B = ... = E_n \tag{2}$$

which represents that all processors should spend the same amount of time performing the processing, guaranteeing the execution time of the function in all processors is minimized.

The figure 1 shows the curves for a GPU and a CPU, the x-axis we vary the block size and obtained the processing time. It is noticed that the curves are both logarithmic functions. These curves are approximated by the method of *least squares*, which gives us an estimate of the function. The function has the form $f(x) = aln(x) - b$. The function is constructed performing the execution of small blocks on each processor.

After generating a certain number of points, normally four points, a curve is fitted to the points, becoming a model for the processor execution times. We fitted curves for all processors and mount a equation system. We solve a equations system and find a point where minimize the *makespan*. The total execution time spent by each processor $g = \{A, B, C, n\}$, is given by:

$$T_g = E_A + E_B + E_C + ... + E_n \tag{3}$$

The 3 is the total execution time, and now we need solve a equation system like this:

$$\begin{cases} x_1 = a_1 ln(T_1) + b_1 \\ x_2 = a_2 ln(T_2) + b_2 \\ x_n = a_3 ln(T_n) + b_3 \end{cases} \tag{4}$$

The equations system is solve by interior point line search filter method [19], . From the moment that are generated the functions by least squares, we continue to measure the time to determine when threads are finishing the job. If the difference of time of threads exceeds a threshold, we restart the process of determining the curves and resolution of the equation system. The synchronization occurs only when the time difference between threads exceeds a threshold set by the user.

Initially the user choose a size block $initialBlockSize$, this size block is send to all processors units. We determine the time for all processors and the processor with smallest time is elected. For the elected processor double the size block and to others units the size block is proportional at time. The

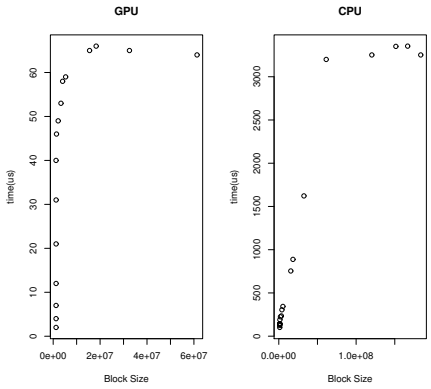

Fig. 1. Curves to GPU x CPU

proportinal time is calculed getting the time of each processor and divide by faster processor. Next pass is get the size block of faster processor and divide this value by each proportinal time, the resulted is the size block in the next iteration. Each value is plotted and with these values is estimated a funtion through least squares. A equation system is generated for each processor. The goal is find a size block for all processor that minimize the time and make the processors terminate at the same time. In 1 is presented the algorithm.

Algorithm 1 Dynamic Algorithm

```
Function dynamic()
B ← initialBlockSize;
while There are data do
  while erroCurve ≥ 0.5 do
    sendPieceDataAllProcessors(B);
    determineFastesProcessor();
    dobleFasterProcessor();
    calculateProportionalTimeEachProcessor();
    synchonize();
    determineCurveProcessor();
  end while
  solveEquationSystem();
  B ← DistributeBlockSize();
  if timeDiference ≥ Threshold then
    synchonize();
    dynamic();
  end if
end while
```

In the algorithm the variable initialBlockSize is initiated by the user. It determines the size of the block in the first iteration for each processor. Then there is a loop that tests whether there are data yet. In the second loop a test is made to determine the error of least squares. We ship the piece of data to all processors, determine the processor that processed the data faster, comparing the processing times. Doubled the time of the faster processor and the other processors sent a piece proportional to the time taken to process the initialBlockSize piece of data, synchronize all processors. And from all values, typically four values of block size and a time equation is determined for each processing unit using the method of least squares. With the equations for each processor the system of equations is solved and distribuited a block size for each processor. If the time difference is increasing with the passage of the distributions is done a synchronization and is rebalanced. The threshold is 0.5 obtained empirically.

IV. Implementation

The implementation was done in the C language with the framework StarPU. StarPU [13] is a tool for parallel programming that supports hybrid architectures like multicore CPUs and accelerators. The StarPU proposes an approach of independent tasks based architecture. Codelets are defined as an abstraction of a task that can be performed on one core of a multicore CPU or subjected to an accelerator. Each codelet may have multiple implementations, one for each architecture in which codelet can be performed using specific languages and libraries for the target architecture. A StarPU application is described as a set of Codelets with data dependencies.

The tool has a set of scheduling policies implemented that the programmer can choose according to the characteristics of the application. The main one is the use of static scheduling algorithm HEFT (Heteregeneous Earliest Finish Time) to schedule tasks based on cost models of task execution.

For each device one codelet has been programmed with the characteristics of the devices. A codelet is a structure that represents a computational kernel. Such a codelet may contain an implementation of the same kernel on different architectures (e.g. CUDA and x86). The applications were implemented by dividing the data set into tasks, implemented as codelets. The tasks are independent, with each task receiving a part of the input set proportional to the processor weight. Two Codelets were implemented one for GPU/CUDA and one for CPU architecture.

To evaluate our load-balancing algorithm, we implemented it by modify the default StarPU balancing algorithm. The modification of the load balancing algorithm is realized by

changing the STARPU_SCHED variable. The STARPU framework has an API that allows modifying the scheduling policies. There are data structures and functions that speed up the process of development. For example the function "double starpu_timing_now (void)" return the current date in micro seconds, which makes it easier for the determination of measures runtime. In StarPU there is a data structure called "starpu_sched_policy" This structure contains all the methods que Implement a scheduling policy.

Three other algorithms were implemented for comparison: the greedy, static and HDSS. The greedy consisted in dividing the input set in pieces and assigning each piece of input to any idle processor, without any priority assignment. The static [7], measures processing speeds before the execution and set static block sizes per processor at the beginning of the execution, with the block size proportional to the processor speed. Finally, The HDSS [17] was implemented using minimum square estimation to estimate the weights and divided into two phases: adaptation phase and completion phase.

The library used for solve the equation system was the IPOPT. IPOPT (Interior Point Optimize) is an open source software package for large-scale nonlinear optimization. It can be used to solve general nonlinear programming problems.

A. Applications

For evaluating our algorithm, we adapted two applications from the CUDA SDK [20] to execute using the StarPU framework, the blackscholes and matrix multiplication applications.

For the matrix multiplication, we assume that each element in the product matrix can be obtained by applying the equation 5. A copy of the matrix A was distributed to all processing units and matrix B was divided according to the load-balancing scheme, the matrix multiplication version use the shared memory. Matrix multiplication has complexity $O(n^3)$.

$$C[i][j] = \sum_{k=1}^{n} A[i][k] * B[k][j] \tag{5}$$

Blackscholes is a popular financial analysis algorithm for calculating prices for European style options. The Black-Scholes equation is a differential equation that describes how, under a certain set of assumptions, the value of an option changes as the price of the underlying asset changes. More precisely, it is a stochastic differential equation that includes a random walk term, which models the random fluctuation of the price of the underlying asset over time. The Black-Scholes equation implies that the value of a European call option,

The cumulative normal distribution function, gives us the probability that a normally distributed random variable will have a value less than x. There is no closed-form expression for this function, and as such it must be evaluated numerically. It is typically approximated using a polynomial function. The idea is to calculate the Black Scholes the greatest amount of possible options. Thus, the input is a vector of data that the options should be calculated by applying the differential equation. The division of the task is given a position of providing input vector to each thread. The complexity of the algorithm is $O(1)$.

V. Results

A. System Configuration

We used three different machines to evaluate our algorithm. Machine A has a quad-core: Core i7 CPU, 8 GB of RAM and 2 nVidia GTX 295 board, each GTX 295 has 280 cores per GPU, 999 MHz memory clock and memory bandwidth 223.8 GB/sec with 2 GPUs on each. Machine B has a machine a quad-core Core i7 CPU, 32 GB of RAM and 2 nVidia GTX 680 GPUs, GTX 680 has 1546 cores, 6 Gbps memory clock and memory bandwidth 192.2 MHz . Machine C has a machine a quad-core: Core i7 CPU, 32 GB of RAM and 1 nVidia GTX Titan GPU, GTX Titan has 2688 cores, 6 Gbps memory speed and memory bandwidth 223.8 GB/sec. We consider the scenarios with only machine A, with machine A and B, and with the 3 machines (A, B and C). The computers are connected by a Gigabit Ethernet network. We used Ubuntu 12.04 and CUDA 5.5.

To use all the n multiprocessors from a GPU, it is necessary to create at least n blocks. Moreover, each multiprocessor simultaneously executes groups (called warps) of m threads from a single block, and several warps should be present on each GPU for efficient usage of its processors.

In all tests, we used all the multiprocessor of the GPUs, launching kernels with k blocks per with 1024 threads for each block, where k is the number of processor in the GPU. For the used GPUs, k is **192, 8 and 30** in GTX Titan, GTX 680 and GTX 295 respectively. For the CPUs, we used all the CPU cores, launchinhg one task per core.

B. Matrix Multiplication

We evaluated the execution time of the matrix multiplication application using one, two and three machines and four different scheduling algorithms: (1) our algorithm, (2) static, (3) HDSS, and (4) StarPU (greedy).

Figure 2 shows the results for matrices with sizes from 10000 x 10000 to 50000 x 50000. In all scenarios, our scheduling algorithms obtained the best results, with the HDSS algorithn in second. The static and StarPU default greedy algorithm were clearly inferior.

With one machine the difference was smaller because there are few types of devices for the scheduler to select. With two machines our algorithm starts to perform better, especially for larger matrices, which is explained by the fact that the execution time of the matrix mutiplication increases quickly as we increase the matrix size. With three machines we have the most heterogeneous environment and the performance gain using our algorithm is the largest. As in the other scenarios, increasing the matrix size also increases the performance gain with our algorithm.

The obtained results shows that the more heterogeneous the environment is, the largest is the perform advantage of using our proposed algorithm. This advantage is due to a better distribution of the data along the execution. Figure 3 shows the time difference between the earliest and latest finishing threads, in the scenario with three machines.

The StarPU greedy algorithm performed well considering that it do not directly use information on the processing

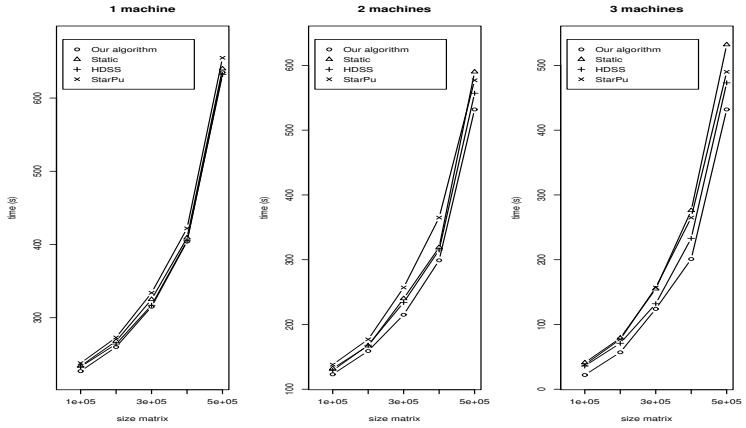

Fig. 2. Difference in runtime with different sizes of matrices for matrices multiplication

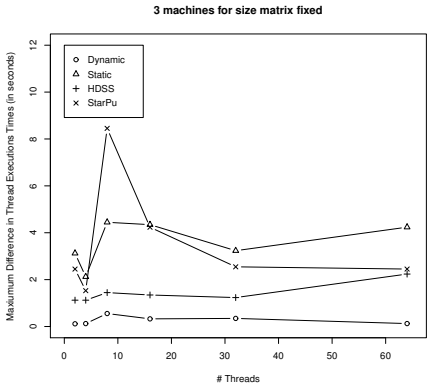

Fig. 3. Time differences between the earliest and latest finishing threads

speeds of the devices. But it uses this information indirectly, since faster devices finish their tasks earlier and, consequently, receive more tasks. The static algorithm (...)

HDSS uses the beginning of the execution to estimate the best block size distribution and uses this distribution through the complete execution. Moreover, larger blocks are used in the beginning of the execution, causing a larger delay due to disbalance when use different types of devices, such as GPUs and CPUs. Finally, HDSS uses a crude approxiamtion to the device capabilities.

Our algorithm estimate fairly accurately the amount of data that should be provided to each processing unit, solving an optimization problem with the restriction that all units should finish the execution of the tasks at the same time. When the difference between finishing the execution of threads for certain data partition exceeds a certain threshold, the algorithm rebalances the data distribution.

C. Blackscholes

We evaluate the execution time of the Blackscholes application using different machine configurations. Figure 4 shows the execution times, where we varied the number of options on each execution and obtained the runtimes. Similarly to the matrix multiplication experiments, the performance gains were the largest with bigger problem sizes (number of options) and more heterogeneous environments, and the results can be explained using the same arguments. Interestingly, the Blackscholes application has linear complexity with the number of options, showing that our scheduling algorithm is also useful with this class of application.

Also, considering that the application finishes in less than 4 seconds, for the scenario with 3 machines, we also see that the cost of solving the optimization problem to determine the best data distribution is small and the obtained gains outweighs the cost of the calculations. For applications tested in a few iterations around 6, it was possible to obtain the solution of the system in a few milisseconds.

Figure 5 confirms this result, showing that the time difference between the earliest and latest finishing threads, in the scenario with three machines is always smaller for our proposed algorithm.

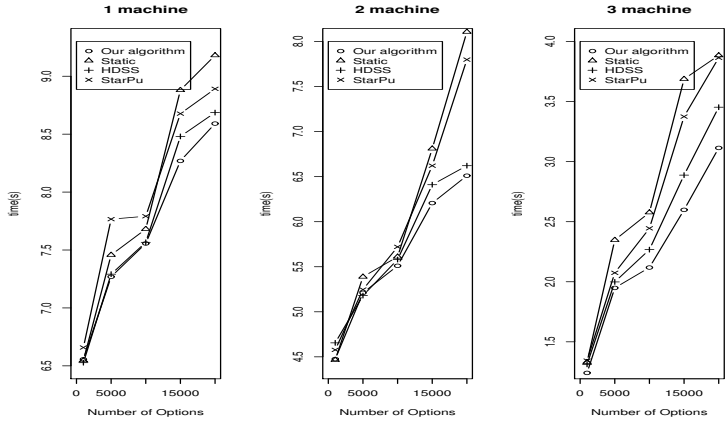

Fig. 4. Difference in runtime with different number options

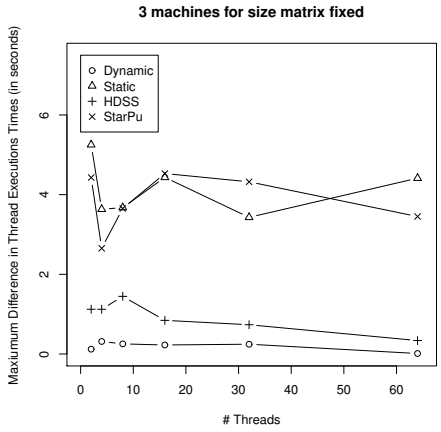

Fig. 5. Time differences between the earliest and latest finishing threads

VI. Conclusions and future work

In this paper we propose an algorithm for scheduling tasks in domain decomposition problems executing on clusters of heterogeneous CPUs and GPUs. It outperforms other similar existing algorithms, due to the online estimation of the performance curve for each processing unit and the selection of the best data distribution for the devices. We showed for two applications that our algorithm provides the highest gains for more heterogeneous and larger problems sizes.

Although we used dedicated clusters, we can also consider the usage of public clouds, where the user can request a number of resources allocated in virtual machines from shared machines. In this case, the quality of service may change during execution, and the addition of the execution time difference threshold permits readjustments in data distributions. We can also consider the scenario with added fault-tolerance, where machines may become unavailable during execution. In this scenario, a simple redistribution of the data among the remaining devices would permit the application to readapt to this scenario.

As ongoing work, we are including the cost of communication in the scheduling algorithm, which is essential for applications where the time spent with information exchange among the tasks cannot be ignored.

Acknowledgment

The authors would like to thank FAPESP (Proc. n. 2012/03778-0) for the financial support.

References

[1] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: ACM, 2014, pp. 1:1–1:8.

[2] G. Oyarzun, R. Borrell, A. Gorobets, O. Lehmkuhl, and A. Oliva, "Direct numerical simulation of incompressible flows on unstructured meshes using hybrid cpu/gpu supercomputers," *Procedia Engineering*, vol. 61, no. 0, pp. 87 – 93, 2013, 25th International Conference on Parallel Computational Fluid Dynamics.

[3] H. K. Raghavan and S. S. Vadhiyar, "Efficient asynchronous executions of amr computations and visualization on a gpu system," *Journal of Parallel and Distributed Computing*, vol. 73, no. 6, pp. 866 – 875, 2013.

[4] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "Parallel multitask cross validation for support vector machine using gpu," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 293 – 302, 2013, models and Algorithms for High-Performance Distributed Data Mining.

[5] Y. Jeon, E. Jung, H. Min, E.-Y. Chung, and S. Yoon, "Gpu-based acceleration of an rna tertiary structure prediction algorithm," *Computers in Biology and Medicine*, vol. 43, no. 8, pp. 1011 – 1022, 2013.

[6] J. Liu and L. Guo, "Implementation of neural network backpropagation in cuda," in *Intelligence Computation and Evolutionary Computation*, ser. Advances in Intelligent Systems and Computing, Z. Du, Ed. Springer Berlin Heidelberg, 2013, vol. 180, pp. 1021–1027.

[7] R. de Camargo, "A load distribution algorithm based on profiling for heterogeneous gpu clusters," in *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, 2012, pp. 1–6.

[8] 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–.

[9] L. Wang, M. Huang, V. K. Narayana, and T. El-Ghazawi, "Scaling scientific applications on clusters of hybrid multicore/gpu nodes," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10.

[10] L. Wang, W. Jia, X. Chi, Y. Wu, W. Gao, and L.-W. Wang, "Large scale plane wave pseudopotential density functional theory calculations on gpu clusters," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–10.

[11] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, pp. 341–352.

[12] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, "Flat: a gpu programming framework to provide embedded mpi," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 20–29.

[13] C. Augonnet, S. Thibault, and R. Namyst, "Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines," Laboratoire Bordelais de Recherche en Informatique - LaBRI, RUNTIME - INRIA Bordeaux - Sud-Ouest, Rapport de recherche RR-7240, Mar 2010.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[15] W. D. Gropp, "Parallel computing and domain decomposition," in *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[16] A. Acosta, V. Blanco, and F. Almeida, "Towards the dynamic load balancing on heterogeneous multi-gpu systems." in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, 2012, pp. 646–653.

[17] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 57:1–57:20, 2013.

[18] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 318–328.

[19] J. Nocedal, A. Wächter, and R. Waltz, "Adaptive barrier update strategies for nonlinear interior methods," *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1674–1693, 2009. [Online]. Available: http://dx.doi.org/10.1137/060649513

[20] CUDA Development Core Team, "CUDA 4.1 Programming Guide," Santa Clara CA, EUA, 2012. [Online]. Available: http://www.nvidia.com/cuda