Dynamic Load Balancing on Multi-GPUs System for Big Data Processing

Chaolong Zhang[1], Yuanping Xu[1], Jiliu Zhou[2], Zhijie Xu[2,3], Li Lu[1], Jun Lu[1]
[1]School of Software Engineering, Chengdu University of Information Technology, Chengdu, China
[2]School of Computer, Chengdu University of Information Technology, Chengdu, China
[3]School of Computing & Engineering, University of Huddersfield, Queensgate, Huddersfield, UK
ypxu@cuit.edu.dn

Abstract: The powerful parallel computing capability of modern GPU (Graphics Processing Unit) processors has attracted increasing attentions of researchers and engineers who had conducted a large number of GPU-based acceleration research projects. However, current single GPU based solutions are still incapable of fulfilling the real-time computational requirements from the latest big data applications. Thus, the multi-GPU solution has become a trend for many real-time application attempts. In those cases, the computational load balancing over the multiple GPU nodes is often the key bottleneck that needs to be further studied to ensure the best possible performance. The existing load balancing approaches are mainly based on the assumption that all GPUs in the same system provide equal computational performance, and had fallen short to address the situations from heterogeneous multi-GPU systems. This paper presents a novel dynamic load balancing model for heterogeneous multi-GPU systems based on the fuzzy neural network (FNN) framework. The devised model has been implemented and demonstrated in a case study for improving the computational performance of a two dimensional (2D) discrete wavelet transform (DWT). Experiment results show that this dynamic load balancing model has enabled a high computational throughput that can satisfy the real-time and accuracy requirements from many big data processing applications.

Keywords: Multi-GPU; Load Balancing; Fuzzy Neural Network; DWT

I. INTRODUCTION

In the last decade, the powerful parallel computing capability of graphics devices and GPUs, originally driven by the market demands for real-time and high-definition game displays, has attracted increasing attention from researchers across the globe in devising hardware-based acceleration solutions for real world engineering and computational challenges [1–3]. Witnessing the trend, in 2007, NVIDIA released the Compute Unified Device Architecture (CUDA) – a software framework and trying to unify the efforts in harnessing the GPU powers for general-purpose usages and "serious applications". It has simplified the GPU programming practices as well as embracing the inherent data parallelism from GPU architecture. The toolkit has greatly assisted some of the most common data/signal processing functions such as Fast Fourier Transform (FFT), Gaussian filtering, and discrete wavelet transform (DWT) that are widely used in applications such as face detection, DNA sequencing, and more recently, machine learning systems such as convolutional neural networks [4–6].

Previous related works on parallelizing processes and data were mainly by using a single GPU that are often struggling to fulfill the real-time requirements from latest big data applications. Thus, the multi-GPU based hardware acceleration solutions have become more popular for applications with huge data throughputs, such as deep learning in the context of big data.

It is a challenging task to fully utilize the parallel computational power of multiple and interconnected GPU nodes. The load balancing model that intelligently distribute tasks to individual GPU node then become a key issue. Chen et al. [7] proposed a task-based dynamic load balancing solution for multi-GPU systems that can achieve a near-linear speedup with the increase number of GPU nodes. Acosta et al. [8] had developed a dynamic load balancing functional library that aims to balancing the load on each node according to the corresponding system runtime. However, these pilot studies are based on the assumptions that all GPU nodes equipped in a multi-GPU platform have equal computational capacity. In addition, the task-based load balancing schedulers these approaches relied upon fall short to support applications with huge data throughputs but limited processing function(s) since there are very few "tasks" to schedule, e.g. DWT. These applications need more attention in refining the task partition in each computational iteration taking into account of the data locality [9].

To optimize the load balancing problem among multi-GPU nodes for big data applications with highly repetitive computational procedures or iterations, this paper presents a novel dynamic load balancing model based on fuzzy neural network (FNN) and data set division method for heterogeneous multi-GPU systems. In this research, five real-time state feedback parameters closely relating to the computational performance of every GPU node are defined. They are capable of predicting the relative computational performance of each GPU node during system runtime. Using the constructed FNN and the advanced data distribution method, a large data set can be adaptively divided to enhance the overall utilization of all hidden computing powers from a heterogeneous multi-GPU system.

The rest of this paper is organized as follows. Section 2 presents a brief review over the preliminaries and related works in the field. Based on the literatures, the rationales of this research are justified; then, the proposed FNN dynamic load balancing model for multi-GPU is explained and its features discussed in Section 3; Section 4 constructs a case study that demonstrates how to improve the computational performance of the lifting scheme in

DWT by using the devised model; Section 5 provides the test results of the design and evaluations. Finally, the Section 6 concludes the research with future works.

II. PRELIMINARIES

A. The GPU Processors

Modern GPUs are not only powerful graphics engines, but also highly parallel arithmetic and programmable processors. More significantly, NVIDIA introduced CUDA in 2007 that was designed especially for general purpose programming, and it can greatly simplify the GPU programming practices. CUDA adopts a SPMD (Single Program, Multiple Data) programming model and provides a sophisticated memory hierarchy (i.e. register, local memory, shared memory, global memory, texture memory and constant memory, etc.), so a GPU program can achieve high data parallel computation through elaborately design CUDA codes and properly usages of different memories according to their respective features (e.g. access mode, size and format, etc.).

The powerful computational capability of a single GPU can satisfy the computational demands of numerous applications in the areas of image processing and computer vision. However, it is still incapable of processing a massive data set due to the limited memory and computational capability of a single GPU processor. Thus, dealing with large volume data sets require the distributed processing mode on the multi-GPUs. At present, there are two typical categories of commonly used multi-GPU platforms, i.e., the standalone computer (a single CPU node with multiple GPU processors) and the cluster computer (multiple CPU nodes and each CPU node has one or more GPU processors). In general, cluster computer systems require more information communication and data transmission time than a single GPU system due to their relative slow speed of PCI-E (Peripheral Component Interconnect Express) buses and network connections. Thus, standalone computers are preferable to cluster computers in this research.

B. Fuzzy neural network

Fuzzy neural network (FNN) is a machine learning algorithm that combines fuzzy systems and neural networks. Generally, traditional fuzzy systems are based on fuzzy rules which are acquired from experimental knowledge of experts [10]. However, it is very difficult to find experts who can extract and summarize knowledge from their experiences, and fuzzy rules are usually not objective. To solve these shortcomings, neural network has been involved to improve the efficiency and accuracy of fuzzy systems and shown to be a promising model which known as FNN.

C. The traditional implementations on Multi-GPU

The Fig.1 demonstrates a traditional load balancing model based on the pure data set division method [2], and it contains: 1) a large original data set is divided into *n* small chunks (subsets) (*n* is equal to the number of GPU nodes contained in a specific multi-GPU system), and each data chunk is distributed to a GPU node respectively. 2) each GPU node processes the corresponding subset. 3) the final results can be generated after merging the output of each GPU node. This approach is very simple and useful, however it may cause unbalancing load problem when the multi-GPU system contains different GPU nodes with unequal computational performance, known as heterogeneous multi-GPU platforms. As a result, the overall performance of a multi-GPU platform depends on the GPU node which has the lowest computational capability.

III. LOAD BALANCING ON HETEROGENEOUS MULTI-GPU SYSTEMS

A. The Struction of Dynamic Load Balancing Model

To solve the unbalancing load problem in the heterogeneous multi-GPU system, this paper presents a novel dynamic load balancing model for optimizing the overall parallel computational performance of multi-GPU while ensuring the good price/performance ratio based on FNN and the data set division method. In this model, the original data set is divided into several equal-size data units and these data units are organized into *n* groups (*n* is equal to the number of GPU nodes in a specific multi-GPU platform) by using the scheduler, see Fig.2. The number of data units for each GPU node are different, and it is determined by the real-time feedbacks (e.g. real-time computational performance and states of each GPU node) of a single GPU node. Thus, the purpose of dynamic load balancing is to minimize the overall processing time by dynamically adjusting the number of data units in a group for each GPU node during runtime according to the real-time state feedbacks of each GPU node.

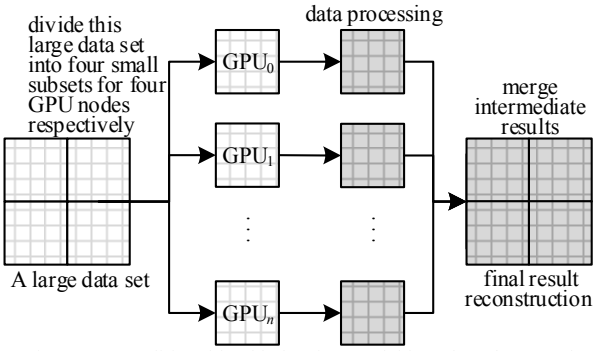

Figure 1. A traditional load balancing model based on the pure data division method

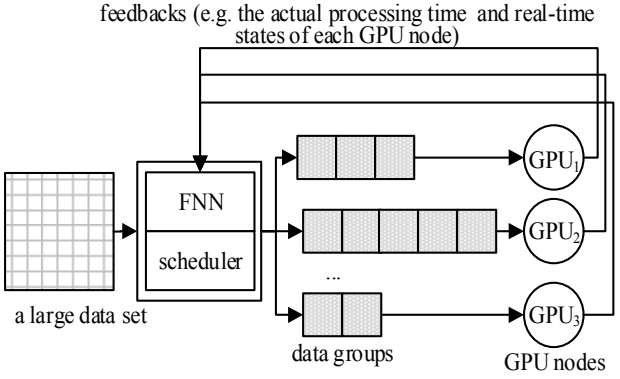

Figure 2. The overall framework of dynamic load balancing model

B. Definition and analysis of Dynamic Load Balancing

To describe the relationship between the real-time state feedback parameters and the number of data units in a group, this model defines the relative computational

ability P_i^n parameter to represent the n^{th} predication of real-time computational performance of i^{th} GPU node, and it means that the scheduler and $P_i^n$is defined as following:

$$P_i^n = f(\frac{D_{unit}}{T_i^{unit}}),\ \ P_i^n \in [0,1],\ n = 0,1,2,... \quad (1)$$

where D_{unit} is a data unit, T_i^{unit} is a feedback parameter denoting the actual processing time of D_{unit} by the i^{th} GPU node, and $f(x)$ is a normalization method.

In the ideal load balancing situation, all GPU nodes in a multi-GPU system would finish their respective work at the same time, and it is satisfying the following equation:

$$\begin{aligned} &T_1=T_2=...=T_m \\ &\Rightarrow T_1^{unit}\times W_1 = T_2^{unit}\times W_2 = ...T_m^{unit}\times W_m \end{aligned} \quad (2)$$

where T_i is the total processing time of i^{th} GPU node in a parallel computational task and W_i is the count of current workload (i.e. the number of data units) for i^{th} GPU node. According to the equations (1) and (2), the number of data units can be calculated. Taking two GPU nodes as an example, $T_1=T_2$, then:

$$\begin{aligned} &T_1^{unit}\times W_1 = T_2^{unit}\times W_2 \\ &\Rightarrow W_1 = \frac{T_2^{unit}\times W_2}{T_1^{unit}} \Rightarrow W_1 = \frac{P_2^n \times W_2}{P_1^n} \end{aligned} \quad (3)$$

The same calculation method can be extended to multiple GPU nodes by using equation (3). Based on equations (1,2,3), the complete procedure for dynamically calculating the number of data units for every GPU node in any multi-GPU platform during runtime can be defined as: 1) This dynamic load balancing model conducts the initial prediction to get P_i^0 for every GPU node by FNN defined in this model after acquiring the original data set (see Fig.2 and Fig.3); 2) The scheduler calculates the number of data units for each data group according to P_i^0 by using equation (3); 3) The multi-GPU platform begins the target parallel computational task when every GPU node gets the corresponding data group organized by the scheduler, and the FNN collects state feedbacks dynamically to prepare the next predication under certain state; 4) Once a GPU node has finished its workload while others are not, the model estimates the remaining time (T_i^r) for each GPU node by using equation (4).

$$T_i^r = T_i^{unit} \times (W_i - W_i') \quad (4)$$

(where W_i' is the finished workload of the i^{th} GPU node.); 5) The data group reorganization is required when remaining time of any GPU node exceeds the threshold preset by this model, such that the next predication is required to get P_i^1; 6) The scheduler reorganizes the remaining data groups for all GPU nodes respectively according to P_i^1; 7) The step2-6 maintain a complete iteration that will be repeated until that all GPU nodes finish their workload at the same time or the remaining time for every GPU's is under the threshold (i.e. satisfying the equation (2)).

According to equation (3), it is convenience to divide data units and organize data groups for each GPU node when P_i^n or T_i^{unit} are given. Unfortunately, P_i or T_i^{unit} can be given only when the whole data processing task is finished. Therefore, precise prediction of P_i^n is the key factor of the devised model.

C. FNN for Dynamic Load Balancing

To predict P_i^n for each GPU node, this research explored in depth the fundamentals of fuzzy mathematics theory and defined five real-time state feedback parameters as the fuzzy sets for each GPU node relating closely to the computational performance – the floating-point operations performance (*F*), global memory size (*M*), parallel ability (*P*), the occupancy rate of computing resources of a GPU (*UF*) and the occupancy rate of global memory of a GPU (*UM*), and each fuzzy set defines "high" and "low" two fuzzy subsets. Likewise, the n^{th} relative computational ability P_i^n is also fuzzed as "high" and "low" two fuzzy subsets. All fuzzy sets and subsets are listed in Table 1.

After defining the fuzzy subsets, this research designed a network structure of FNN that combines theories of the fuzzy mathematics and the back propagation to predict P_i^n of each GPU nodes before the scheduler organizes data groups, see Fig.3. The first layer is an input layer while the second layer, third layer and fourth layer are considered to be the fuzzy input layer, hide layer and output layer respectively in the classic structure of back propagation networks. This FNN has ten fuzzy truth values as inputs and two fuzzy truth result values as outputs. The final layer (i.e. fifth layer) decodes the fuzzy truth values to the correct value which is the actual P_i^n of i^{th} GPU of the n^{th} predication. The devised FNN uses I_i^j to denote the input of the i^{th} artificial neuron in j^{th} level layer, O_i^j to denote the output of the i^{th} artificial neuron in j^{th} level layer, w_i to denote weights of connections between the second and third layer and w_i' to denote weights of connections between the third and fourth layer, and w_i'' to denote weights of connections between the fourth and fifth layer (see Fig.3). The workflows of the corresponding inputs and outputs are illustrated in Fig.3.

- **Input layer:** The input layer collects real-time states of a GPU node and generates values of the five state feedback parameters (see Table 1) as inputs when a predication of P_i^n is required. The input layer merely import real-time state feedback parameters into the FNN , and the input-output formula shows as the following:

$$O_i^1 = I_i^1 = x_i \quad (5)$$

where x_i is corresponding to the values of *F*, *M*, *P*, *UF* and *UM* in Table 1 respectively.

Table 1. The defined fuzzy sets and subsets

Sets	Descriptions	Fuzzy subsets	Descriptions of Fuzzy subsets
F	The floating-point operations performance	*FL*	Low
		FH	High
M	Memory size	*ML*	Low
		MH	High
P	Parallel ability (a positive correlation with the count of processor cores of a GPU node)	*PL*	Low
		PH	High
UF	The occupancy rate of computing resources	*UFL*	Low
		UFH	High
UM	The occupancy rate of global memory	*UML*	Low
		UMH	High
CP	The fuzzy relative computational ability	*CPL*	Low
		CPH	High

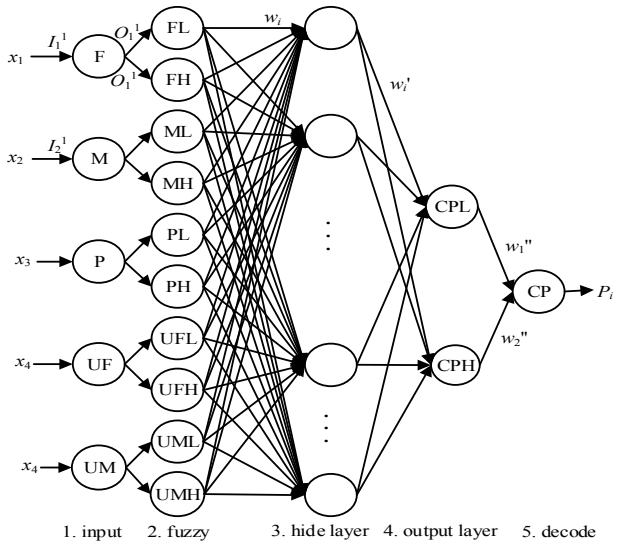

Figure 3. The structure of FNN in the dynamic load balancing model

- **Fuzzy layer:** The fuzzy layer transforms the correct values into fuzzy truth values by using an membership function. The input and output formulas are illustrated as the following:

$$I_i^2 = O_i^1$$
$$O_i^2 = u_A\left(I_i^2\right),\ O_i^2 \subset [0,1] \tag{6}$$

where $u_A(x)$ is membership [11]. There are a lot of membership functions available, but this research chose the sigmoid function due to its "S" shaped curve can gracefully reflect the fluctuations of computational performance of GPU nodes [12]. The equation of sigmoid membership function is defined as the following:

$$f(x) = \frac{1}{1+e^{-a(x-c)}} \tag{7}$$

where a and c are constants having different values for different fuzzy subsets. Taking the occupancy rate of computing resources of a GPU node (UF, and $UF \in [0, 1]$) as an example, this model takes a=-15 and c=0.5, and a=15 and c=0.5 to transform a correct value of UF into its fuzzy truth values of UFL and UFH respectively, so the membership functions of UFL and UFH can be defined as:

$$\begin{cases} UFL: u_{UFL}(UF) = \dfrac{1}{1+e^{15\times(UF-0.5)}} \\ UFH: u_{UFH}(UF) = \dfrac{1}{1+e^{-15\times(UF-0.5)}} \end{cases} \tag{8}$$

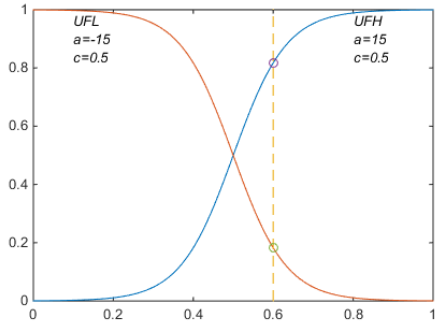

Figure 4. The Membership Functions of UFL and UFH

According to equation (8), for instance, when a GPU's UF=0.6 at some point, then the membership value of UFL is 0.18, and the membership value of UFH is 0.82, see Figure 4.

- **Hide layer:** In principle, the more the number of hide layers, the more complex functions can be fitted. However, it also may cause the disadvantages of a mass of computation and overfitting. Generally, one hide layer can meet requires of common prediction purposes in this research [12]. Therefore, this load balancing model has only one hide layer. The input and output formulas is defined as the following:

$$I_i^3 = \sum_{j=1}^{n} w_i O_j^2 - \theta_i$$
$$O_i^3 = \varphi(I_i^3) \tag{9}$$

where O_j^2 (n=10) denotes outputs of 10 artificial neurons on the 2^{nd} level layer, and Θ_i is a threshold value while $\varphi(x)$ is the activation function used by the artificial neurons. This research chose a sigmoid function as the activation function:

$$\varphi(x) = \frac{1}{1+\exp(-ax)} \tag{10}$$

- **Output layer:** The output layer outputs fuzzy truth values of the "high" and "low" fuzzy subsets of P_i^n. The input and output formulas are defined as the following:

$$I_i^4 = \sum_{j=1}^{m} w_j' O_j^3 - \theta_i'$$
$$O_i^4 = \varphi\left(I_i^4\right) \tag{11}$$

where m is the number of artificial neurons of on the hide layer (i.e. 3rd level layer), Θ_i' is a threshold value while the definition of $\varphi(x)$ is the same as equation (10).

- **Decode layer:** Decode layer is added in this network to transform the fuzzy truth values of the CPL and CPH into the correct value of P_i^n by using the fuzzy weighted average method. The input and output formulas are defined as the following:

$$I^5 = \sum_{i}^{2} w_i'' O_i^4$$
$$P_i = O^5 = \frac{I^5}{\sum_{i=1}^{2} O_i^4} \tag{12}$$

Based on FNN developed in Fig.3, the proposed load balancing model can be learned by training data using back propagation algorithm that is collected from historical data about real-time state feedbacks (e.g. data processing time and a GPU' states at some point). After the model is trained, it can be used to predict P_i^n, and then scheduler can organize the data groups dynamically according to equation (3), so as to this model balances the load of large-volume-data based applications dynamically and flexibly.

IV. A CASE STUDY

Discrete wavelet transform (DWT) is one of the most widely used algorithms in the areas of signal processing, image processing, biomedicine, machine vision, etc. The

widespread usage of DWT has motivated the development of fast DWT approaches. Among them, the lifting scheme, known as the second generation wavelet or lifting wavelet transform (LWT), is the most popular fast DWT algorithm. However, the pure software accelerated DWT still failed to cope with the demands of real-time processing when facing very large data sizes. Big data applications with highly repetitive computational procedures or iterations is the prominent feature of LWT, so this research applied parallel computation of 2D LWT as a case study to verify the feasibility and effectiveness of the dynamic load balancing model.

The 1D forward LWT contains four operation steps: split, predict, update, and scale [13].

- **Split:** This step splits the original signal into two subsets of coefficients, i.e. even and odd, and the former is corresponding to the even index values while the latter is corresponding to the odd index values. The split method is expressed as equation(13), and it is also called the lazy wavelet transform.

$$\begin{cases} even[i] = X[2i] \\ odd[i] = X[2i] \end{cases} \tag{13}$$

- **Predict:** The odd coefficients can be predicted from the even by using prediction operator P, and then replace the older odd values by the prediction result as the next new odd coefficients, recursively. This step can be expressed as equation(14).

$$odd = odd - P(even) \tag{14}$$

- **Update:** Likewise, even coefficients can be updated from the update operator U, and then replace the older even values by the updated result as the next new even coefficients, recursively. This step shows as equation(15).

$$even = even + U(odd) \tag{15}$$

- **Scale:** Normalize even and odd coefficients with factor K respectively by using equation (16) to get the results of *evenApp* and *oddDet*, which are the final approximation coefficients and detail coefficients of forward LWT respectively.

$$\begin{cases} evenApp = even \times (1/K) \\ oddDet = odd \times K \end{cases} \tag{16}$$

The inverse LWT with lifting scheme is achieved by inverting the complete sequence of steps of forward LWT and switching the corresponding addition and subtraction operators. For a multi-level DWT, the process is repeatedly applied to the approximation coefficients until a desired number of decomposition levels is reached. In the case of a 2D DWT, it simply needs to perform horizontal 1D LWT for each row of a 2D input data set and vertical 1D LWT for each corresponding column in sequence separately due to 2D LWT can be realized through the 1D wavelet transform along its x- and y-axes.

Table 2 illustrates equations for a single level forward LWT based on the CDF (9, 7) wavelet, and its scheduling software routine on a CPU is illustrated in Table 3. The basic idea is that every step of the lifting scheme is performed by different functions, and the CPU program schedules and launches these functions with respect to all data dependencies.

TABLE 2 A SINGLE-LEVEL FORWARD LWT BASED ON CDF (9, 7) WAVELET

Split:	$\begin{cases} even[i] = X[2i] \\ odd[i] = X[2i+1] \end{cases}$
1th Predict:	$odd[i] -= -\alpha \times (even[i] + even[i+1])$
1th Update	$even[i] -= -\beta \times (odd[i] + odd[i-1])$
2th Predict	$odd[i] -= -\gamma \times (even[i] + even[i+1])$
2th Update:	$even[i] -= -\delta \times (odd[i] + odd[i-1])$
Scale:	$\begin{cases} even = even \times \varepsilon \\ odd = odd \times (1/\varepsilon) \end{cases}$

TABLE 3 THE SCHEDULING SOFTWARE ROUTINE ON A CPU

```
for(i=0;i<COLS/2;++i){  // Split
   X[i] = X[2*i+i]; X[i+COLS/2]=X[2*i+1]; }
for(i=0;i<COLS/2;++i){  // 1th Predict
   X[i] += alpha * (X[i-COLS/2] + X[i-COLS/2+1]); }
for(i=0;i<COLS/2;++i){  // 1th Update
      X[i]+=beta*(X[i+COLS/2-1]+X[i+COLS/2]); }
for(i=0;i<COLS/2;++i){  // 2th Predict
   X[i] += gamma * (X[i-COLS/2] + X[i-COLS/2+1]); }
for(i=0;i<COLS/2;++i){  // 2th Update
   [i] += deta * (X[i+COLS/2-1] + X[i+COLS/2]);}
for(i=0;i<COLS/2;++i){  // Scale
   X[i] = (1/phi)*X[2*i+i]; X[i+COLS/2]=phi*X[2*i+1]; }
```

V. EXPERIMENT RESULTS AND EVALUATION

This section tests and evaluates experimental results gathered from the dynamic load balancing model based implementation of LWT case. Table 4 specifies the test computer of this research which contains two different GPU nodes — a middle low GPU (GTX 750 Ti) and a high-end GPU (GTX 1080).

To begin with, this study tested and compared the processing time of LWT between a single GPU (using only GPU1 and GPU2 respectively) and two GPUs (using both GPU1 and GPU2) environments without using the devised load balancing model. This test performs 4 levels of forward 2D LWT with CDF (9, 7) wavelet on three test environments having data size from 10240 ×10240 to 12288×12288. Table 4 shows that the GPU2 version needs less processing time than the GPU1 version because the computational performance of GPU2 is higher than GPU1. The two GPUs (GPU1 & GPU2) version merely gains limited speedup of about 1.6 times compared with GPU1 and of around 1.3 times compared with GPU2. It can be clearly seen from Table 5 that the overall processing time of the two GPUs version in context of the unbalancing situation is equal to the GPU1 version because the overall computational performance of a multi-GPU platform is mainly determined by the GPU node with the lowest power, in this case, it is GPU1.

Then, this study compared the time performance between unbalancing implementation (i.e., each GPU node processes a half of a large data set without consideration of its computational performance) and dynamic load balancing implementation by using two GPUs version see Fig.5 which shows that the dynamic load balancing implementation can keep the high computational performance steadily, i.e., it can process very large data sets (e.g. 16384×16384) less than one second. Compared with the unbalancing implementation, the speedup of dynamical load balancing implementation can reach the high point at about 12 times. The

experimental results show that the proposed model achieves higher computational performance than the unbalancing implementation, and it can satisfy the real-time and big data applications.

VI. Conclusions and Future Work

To fully utilize the parallel computation power of modern GPUs, this paper presents a novel dynamic load balancing model for the multi-GPU platform based on FNN and the dataset division method. Tests show that the proposed model can achieve superior computational performance than conventional data-only division method. The innovative model and its corresponding techniques have addressed the key challenges from big data applications that are often accompanied by extremely large input volume and highly repetitive operational procedures or iterations. One avenue opened up during the research for future exploration is how to bridge the FNN idealism across the GPU and CPU boundary, especially when facing multi-CPU or Cell CPUs, for a hybrid and efficient task distribution scheme.

TABLE 4. The Specification of a Test Computer System

	Description
CPU	Intel Core i7-4790 3.6GHZ
GPU1	GeForce GTX 750 Ti, 2G
GPU2	GeForce GTX 1080, 8G
OS	Windows 10 64 bit
CUDA	Version 8.0

TABLE 5 Time Performance of Three Test Environments (ms)

data size	GPU1	GPU2	GPU1 & GPU2
10240×10240	4500	3312	2758
11264×11264	7058	5564	3876
12288×12288	8546	7148	4500

TABLE 6 The time Performance of Two GPUs Version With Unbalancing Implementation (ms)

data size	GPU1	GPU2	overall
10240×10240	2758	1500	2758
11264×11264	3876	2806	3876
12288×12288	4500	3645	4500

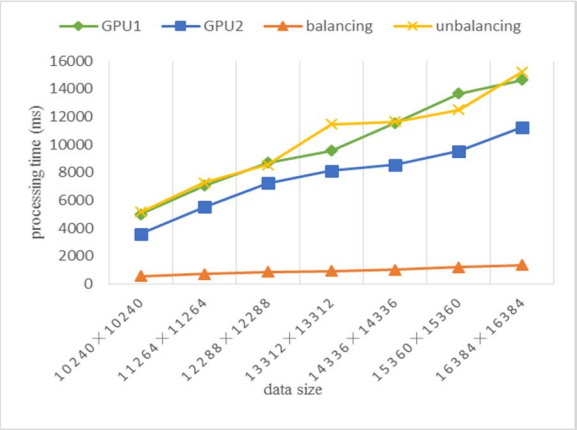

Figure 5. The time performance of unbalanced and balanced implementations

Acknowledgment

This work is supported by the NSFC (61203172), the STD of Sichuan (2017JY0011 and 2014GZ0007), and Shenzhen STPP (GJHZ20160301164521358).

References

[1] D. B. Kirk and W. H. Wen-mei, Programming massively parallel processors: a hands-on approach. Newnes, 2012.

[2] R. Couturier, Designing Scientific Applications on GPUs. CRC Press, 2013.

[3] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," IEEE Micro, vol. 31, no. 5, pp. 7–17, 2011.

[4] C. W. Lee, J. Ko, and T.-Y. Choe, "Two-way partitioning of a recursive Gaussian filter in CUDA," EURASIP J. Image Video Process., vol. 2014, no. 1, pp. 1–12, 2014.

[5] J. A. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, and A. M. Vidal, "Real-time massive convolution for audio applications on GPU," J. Supercomput., vol. 58, no. 3, pp. 449–457, Dec. 2011.

[6] F. Nasse, C. Thurau, and G. A. Fink, "Face detection using gpu-based convolutional neural networks," in International Conference on Computer Analysis of Images and Patterns, 2009, pp. 83–90.

[7] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," Ipdps, 2010.

[8] A. Acosta, R. Corujo, V. Blanco, F. Almeida, H. P. C. Group, and E. T. S. De Ingenier, "Dynamic Load Balancing on Heterogeneous Multicore / MultiGPU Systems," 2010.

[9] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load balancing in a changing world: dealing with heterogeneity and performance variability," in Proceedings of the ACM International Conference on Computing Frontiers, 2013, p. 21.

[10] R. J. Kuo, S. Y. Hong, and Y. C. Huang, "Integration of particle swarm optimization-based fuzzy neural network and artificial neural network for supplier selection," Appl. Math. Model., vol. 34, no. 12, pp. 3976–3990, 2010.

[11] C. L. P. Chen, Y.-J. Liu, and G.-X. Wen, "Fuzzy neural network-based adaptive control for a class of uncertain nonlinear stochastic systems," IEEE Trans. Cybern., vol. 44, no. 5, pp. 583–593, 2014.

[12] S.-H. Lee and J. S. Lim, "Forecasting KOSPI based on a neural network with weighted fuzzy membership functions," Expert Syst. Appl., vol. 38, no. 4, pp. 4259–4263, 2011.

[13] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," SIAM J. Math. Anal., vol. 29, no. 2, pp. 511–546, 1998.