Parallel Self-organizing Map Using Shared Virtual Memory Buffers

Noor Elaiza Bt Abd Khalid, Muhammad Firdaus B. Mustapha, Azlan B. Ismail and Mazani B. Manaf

Abstract Parallel implementation of Self-organizing Map (SOM) has been studied since last decade. Graphic Processing Unit (GPU) is one of most promising architecture for executing SOM in parallel. However, there are performances issues are highlighted when imposing larger mapping and dataset size onto parallel SOM that executed on the GPU. Alternatively, heterogeneous systems that soldered GPU together with Central Processing Unit (CPU) are introduced in order to improve communication between CPU and GPU. Shared Virtual Memory (SVM) is one of features in OpenCL 2.0 which allows the host and the device to share a common virtual address range. Thus this research proposes to introduce a parallel SOM architecture that suitable for both GPU and heterogeneous system with the aim to compare the performance in term of computation time. The architecture comprises of three kernels that executed on two different platforms (1) discrete GPU platform and (2) heterogeneous system platform that tested using SVM buffers. The experimental results show the parallel SOM running on heterogeneous platform has significant improvement in computation time.

Keywords Parallel self-organizing map · GPU computing · OpenCL

N.E.B.A. Khalid (✉) · M.F.B. Mustapha (✉) · A.B. Ismail · M.B. Manaf
Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA, Shah Alam, Malaysia
e-mail: elaiza@tmsk.uitm.edu.my

M.F.B. Mustapha
e-mail: firdaus19@gmail.com

A.B. Ismail
e-mail: azlanismail08@gmail.com

M.B. Manaf
e-mail: mazani@tmsk.uitm.edu.my

© Springer International Publishing AG 2017

D. Król et al. (eds.), *Advanced Topics in Intelligent Information and Database Systems*, Studies in Computational Intelligence 710,
DOI 10.1007/978-3-319-56660-3_5

1 Introduction

Graphic Processing Unit (GPU) is a many core processor consisting hundreds or even thousands of compute cores that has been used to process the applications of scientific computing and scientific simulations or also called General Purpose Graphic Processing Unit (GPGPU) [1]. GPU computing has become popular since the introduction of GPU programming framework such as Compute Unified Device Architecture (CUDA) in 2007 and Open Computing Language (OpenCL) in 2009 [2]. Many researchers are trying to take advantages of GPU computing to execute Self-organizing Map (SOM) algorithm in parallel manner. Some researchers agree that GPU variant shows the significant speed up for large data compared to Central Processing Unit (CPU) variant [3, 4]. Both comparisons are proven that GPU computing achieves better performance in terms of computation time. Moreover, GPU computation time will reduce when the increment of input dimension and SOM network size compared to execute on CPU [5]. On the other words, the GPU computation is suitable to handle large dataset with high dimension. However, [6] address the larger mapping size and feature dimensions, the slower the computation time for both CPU and GPU. The major issue is addressed by researchers in executing parallel SOM on GPU is memory utilization increase when processing large mapping size [5–7]. The high memory utilization leads to high rate of memory transfers which will burden the processing time.

For the meantime, some researchers attempt to decompose several steps of the algorithm with the aim to execute SOM in parallel. There are different configurations of task decomposition on SOM algorithm has been applied and observably many researches works are found in the literature perform decomposition on calculate distance and find the Best Matching Unit (BMU) steps. For instance, [6, 8] decompose calculate Euclidean distance and BMU searching process. Some researchers try to decompose calculate distance, find BMU, and update the neurons' weights [5, 7, 9, 10]. Meanwhile, [4, 11] decompose initialize neuron weights. From the literature shows that the major steps have been decomposed include; initialize weights, calculate distance, find the BMU, and update the weights. Figure 1 illustrates steps in SOM algorithm that are decomposed by researchers specifically using GPU computing platform.

Recently, heterogeneous systems using GPU has become attractive computing model given the available scale of data-parallel performance and GPU programming framework such as OpenCL. The heterogeneous systems that combine CPU and GPU on a single chip are capable to share the same memory which leads to improve communication between each other [12]. Thus, the aim of this paper is to explore the parallel SOM architecture that suitable for executing discrete GPU and heterogeneous system architecture. The architecture comprises of three kernels that executed on two different platforms; (1) discrete GPU platform and (2) heterogeneous system platform that tested using SVM buffers; coarse-grained and fine-grained buffers. The performance of both platforms are measured based on computing time in seconds.

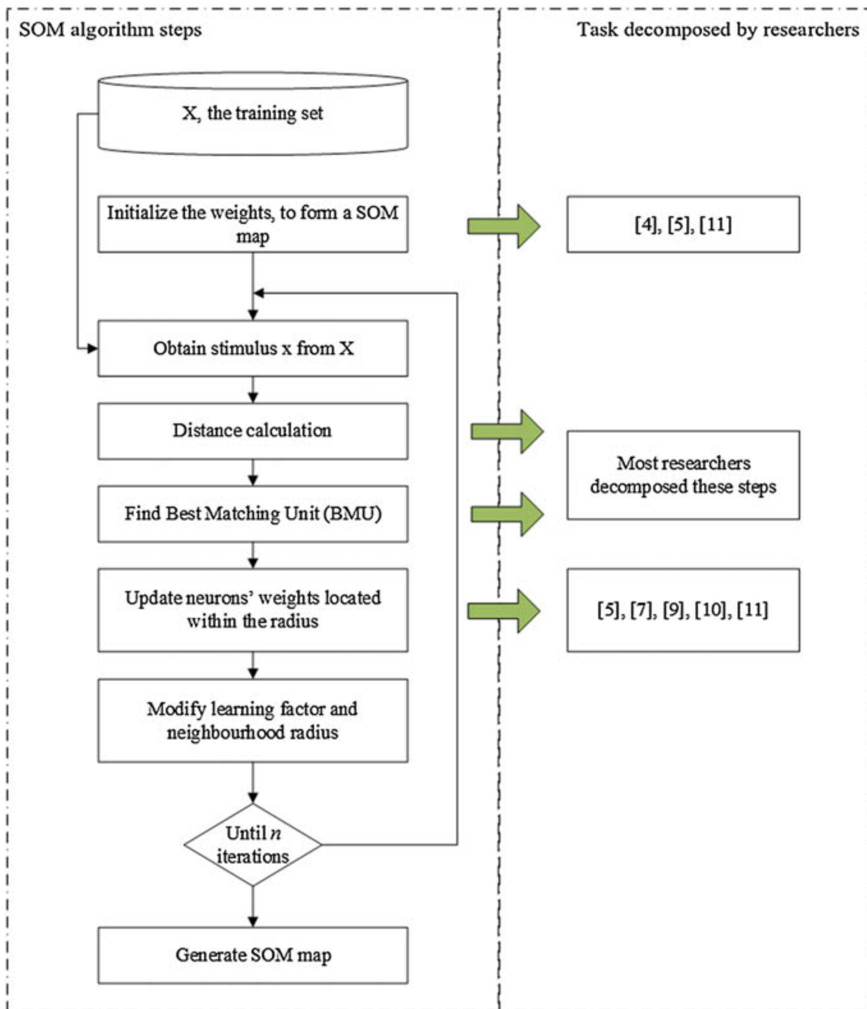

Fig. 1 Task decomposition of SOM algorithm

2 Evolution of Parallelism in Hardware

Nowadays, CPU approximately touches its limit whereas increasing the frequency of CPU will consume large power. As an alternative, modern graphic cards or GPU take role of powerful computation hardware. The performance gap between CPU and GPU becomes wider as GPU achieves seven times for gigaflops and bandwidth metrics compared to CPU [13].

There are another types of accelerator core that gained interest over last decade such as Field-Programmable Gate Arrays (FPGAs) and the Cell Broadband Engine

(Cell BE) [13]. However, the GPUs are most popular among these accelerator cores because large numbers of desktop and laptops computers have a dedicated GPU compared to FPGAs and the Cell BE are only found in specially ordered setup. Additionally, the future of the Cell BE is currently uncertain and FPGAs too hard to program for general-purpose computing [13].

The most recent technology, heterogeneous systems, that incorporated CPUs and GPUs together on a single integrated circuit (IC) chip, is quickly becoming the design paradigm for today's platform because of their impressive parallel processing capabilities [14]. The introduction of heterogeneous programming models such as OpenCL 2.0 in July 2013 is to improve the communication between CPU and GPU. This framework treats the GPUs as a first-class computing device which allows the GPUs to manage their own resources, as well as access some of the CPU's resources.

2.1 GPU Programming Framework

OpenCL 1.2

OpenCL is a framework of parallel programming that can be used for programming a heterogeneous collection of central processing units (CPUs), GPUs and other discrete computing devices are organized into a single platform [12]. An OpenCL device or GPU is divided into one or more compute units (CUs) where each CU has one or more processing elements (PEs).

An OpenCL program is executed on a host and the host is connected to one or more GPUs. The host code portion of an OpenCL program runs on a host processor according to the models native to the host platform. The OpenCL program host code submits various commands to a command queue, to be executed by processing elements within the device. The command can be of different types, such as for execution, memory management, or synchronization.

Meanwhile, the device code or kernel is executed on GPU. Kernels are sets of instructions that are executed in parallel. Each kernel program is stored in a separate file with the extension of. cl. However, the main problem in performance for OpenCL 1.2 applications is data transfers between the host code and device code [14]. Moreover, the memory management in OpenCL 1.2 still relied on the programmer to take care of data movement between the CPU and the GPU.

OpenCL 2.0

OpenCL 2.0 is the next release of OpenCL framework which introduced new features that concentrate on managing the heterogeneous system. This feature is to overcome the data transfers between CPU and GPUs. OpenCL 2.0 introduced Shared Virtual Memory (SVM) which allows the host and the device to share a common virtual address range [12]. This reduces overhead by eliminating deep copies during host-to-device and device-to-host data transfers. Deep copies involve completely duplicating objects in memory [14]. SVM implementations can be described the following below:

- Coarse-grained: includes synchronization during mapping and unmapping of memory objects, along with during kernel launch and completion. Accordingly the memory object updates after the completion of kernel and the unmapping of memory.
- Fine-grained: the synchronization occurs during the implementation of SVM buffers. Therefore the memory objects are updated coherently for both CPU and GPU.

2.2 Proposed Architecture

This paper proposes to parallelize all of the three steps using separate kernels code. The first kernel is to calculate the distance between neurons and a current input vector. The second kernel is to find BMU for each input vector. The BMUs values are then used by the third kernel to update the map appropriately. The parallel SOM will be tested on three different buffers; (1) non-SVM buffers, (2) coarse-grained buffer SVM, and (3) fine-grained buffer SVM. Figure 2 shows the proposed architecture of parallel SOM.

Initially, the input data are retrieved and stored into an array and follows by initialization of SOM parameter such as learning factor and weights. These tasks are performed at host side. In order to execute the kernels three functions are created for providing setting, initializing parameters, and calling the kernels. For example, the calculate distance function uses to call Calculate Distance kernel and it is done the same way with the other two kernels.

All of the kernels are implemented on the device side. The Distance Calculation kernel is to calculate the distance between neurons and current input vector and store the distance values into an array. It is represented by amount of work-items that is equal to the number of neurons in the SOM map. As such, each work-item of the kernel is responsible for finding the distance between a single neuron and the current input vector. This research applies Manhattan distance calculation.

Meanwhile, the Find BMU kernel applies reduction method with the aim of finding BMU in parallel. The kernel utilizes work unit the same number of neurons on SOM map. This process includes two stages where the first stage is to divide the work unit per compute unit (CU). The work unit per CU then is divided by the size of local work group in order to acquire the amount of work units for each processing element must deal with. Each work-item in the work-groups will find the minimum distance among the distance values covered by the work groups. The minimum distance value identified by the kernel is stored in a local array. The second stage is to find minimum distance for each CU from the minimum values in the local array. The minimum values of each CU then stored into global array and the host will determine the winning neurons.

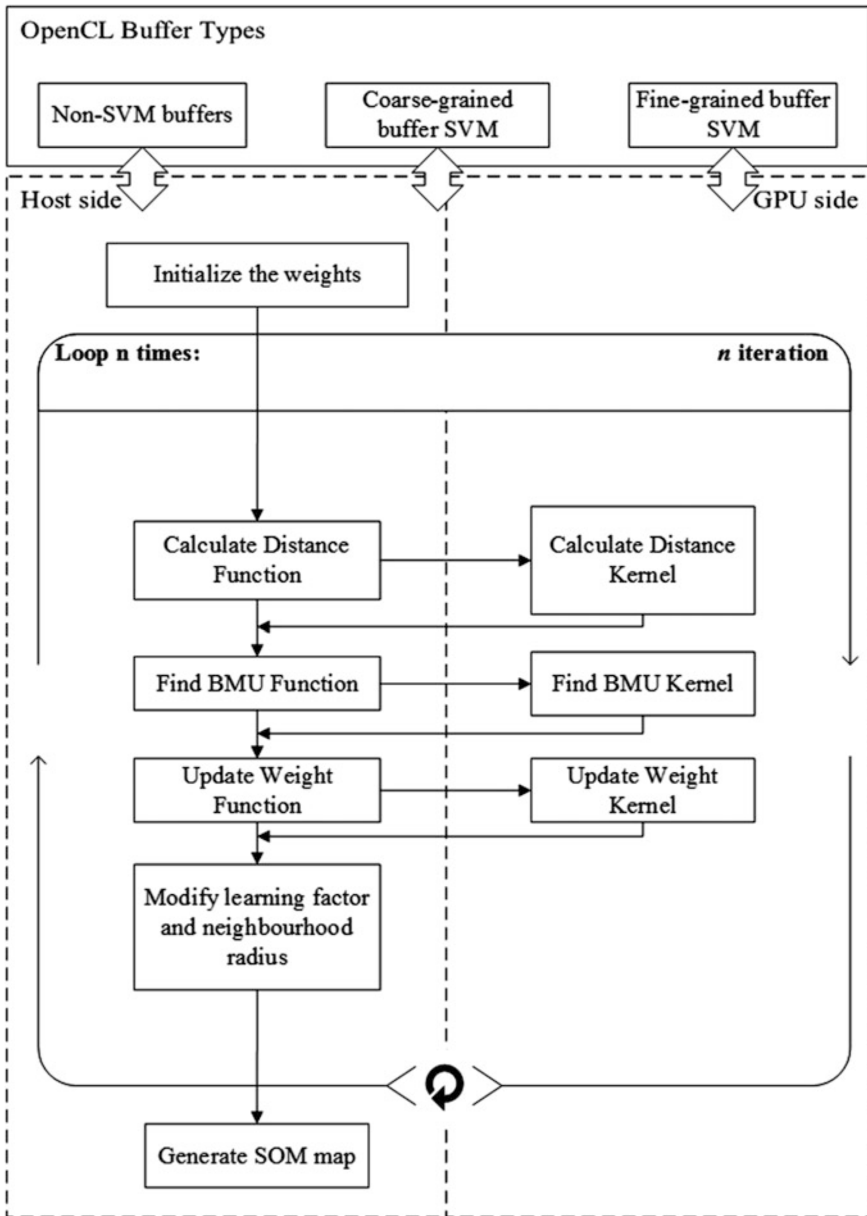

Fig. 2 The proposed architecture of parallel SOM

On the other hand, the third kernel involves updating the weight of neurons based on learning rate and neighborhood function. The learning rate describes how much a neuron's vector is changed during an update according to how far away the

Table 1 The implementation setting of buffers

Type of OpenCL buffers	Parameter declaration	Allocating the parameters
Non-SVM	`cl::Buffer input_buff;`	`input_buff = cl::Buffer (device_context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, sizeof(float)* input_size * input_length, input, &err);`
Coarse-grained SVM	`float* input_buff;`	`input_buff = (float*)clSVMAlloc (oclobjects.context, CL_MEM_READ_WRITE, input_size *input__length *sizeof(float,0);`
Fine-grained SVM	`float* input_buff;`	`input_buff = (float*)clSVMAlloc (oclobjects.context, CL_MEM_READ_WRITE| CL_MEM_SVM_FINE_GRAIN_BUFF, input_size * input_length * sizeof (float,0);`

neuron is from the BMU on the map. The BMU and its close neighbors will be changed the most, while the neurons on the outer edges of the neighborhood are changed the least. Right after executing the three kernels, modify learning factor and neighborhood radius take place. All of the steps include in the loop block will repeat until n iteration or epoch before the SOM map is generated.

2.3 OpenCL Buffers Types

With the interest to evaluate the performance of SVM buffers, the proposed architecture is tested with three different type of OpenCL buffer. The following Table 1 shows the implementation of three buffers.

The non-SVM buffer is following the OpenCL 1.x specification, meanwhile the SVM buffers that comprise of coarse-grained and fine-grained buffers are using OpenCL 2.0 specification. The fine-grained SVM applies CL_MEM_SVM_FINE_GRAIN_BUFF in order to activate fine-grained compared to coarse-grained SVM. Additionally, the synchronization point of coarse-grained SVM occurs during mapping or unmapping of memory objects and kernel launch and completion. While the synchronization point of fine-grained SVM happens during the executing of the memory objects.

3 Computation Experiment

3.1 Experimental Setup

In this study, Bank Marketing dataset from UCI Machine Learning Repository is applied for the computation experiments. Three sizes of dataset has selected; 5000, 10000 and 15000. Table 2 depicts the description of the dataset and experimental design for this paper. The experiments are conducted in order to examine the performance of parallel SOM using three different buffers; non-SVM (NSVM), coarse-grained SVM (CG), and fine-grained SVM (FG). The evaluations are based on computation time that divided into three: total kernel time, total setup time, and total time. The experiments were conducted on a laptop equipped with Intel i7-6700HQ processor, 16 GB of RAM and built in Intel® HD Graphics 530. This processor belongs to the Skylake family which supports the OpenCL 2.0. It is equipped with 4 CPU cores and 24 number of execution units placed at GPUs.

3.2 Computation Results and Analysis

The results of the computation experiments are presented in Fig. 3. Performance of three different buffers are included into the figure using the following label; NSVM, FG, and CG. Each buffer has tested on three different sizes of dataset and four SOM mapping size. The NSVM is executed on OpenCL 1.x platform meanwhile CG and FG are performed on OpenCL 2.0.

From the results, FG outperforms NSVM and its sibling for every dataset size. The parallel SOM triggered by FG well utilize SVM features in OpenCL 2.0 due to CPU and GPU efficiently share a common virtual address space where it is removing the need to explicitly copy buffers back and forth between the two devices [14]. Meanwhile the CG performs the worst among of three buffers where it suffers from consuming the most total setup time. The CG buffers which also utilize SVM feature but apply `clEnqueueSVMMap` and `clEnqueueSVMUnmap` for the synchronization point likely to burden the processing. Overall, all of the buffers types share the same trends as the bar chart raise higher when executing larger dataset size and mapping size.

Table 2 The experimental setting

Dataset parameters		SOM parameters		Performance measurement
No. of samples	No. of parameters	Iterations	Mapping size	Time, s
5000	3	30	10×10 20×20 30×30 40×40	Total kernel time
10000				Total setup time
15000				Total time

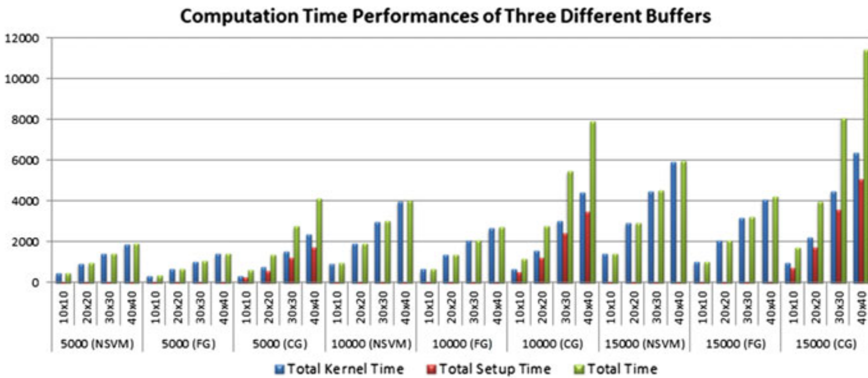

Fig. 3 Computation time performances of three different buffers

4 Conclusion

This paper proposes a parallel SOM architecture that comprise of three kernels. There is a possible way to parallelize the SOM algorithm through decomposing three major steps; calculate distance, find the BMU, and updating the weights. On the other note, GPU computing offers a great solution for SOM parallelism where the large amount of calculation could be catered by massive parallelism. The aim of this architecture is to accelerate SOM training. From the results show that the proposed parallel SOM architecture is capable to execute parallel SOM especially for FG buffers.

Acknowledgements This work was funded by Ministry of Higher Education (MOHE) of Malaysia, under the Fundamental Research Grant Scheme (FRGS), grant no. FRGS/81/2015 and Academic Staff Bumiputera Training Scheme (SLAB). The authors also would like to thank the Universiti Teknologi MARA for supporting this study.

References

1. Perelygin, K., Lam, S., Wu, X.: Graphics Processing Units and Open Computing Language for parallel computing. Comput. Electr. Eng. **40**(1), 241–251 (2014)
2. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors. Elsevier (2013)
3. Wittek, P., Darányi, S.: Accelerating text mining workloads in a MapReduce-based distributed GPU environment. J. Parallel Distrib. Comput. **73**(2), 198–206 (2013)
4. Lachmair, J., Merényi, E., Porrmann, M., Rückert, U.: A reconfigurable neuroprocessor for self-organizing feature maps. Neurocomputing **112**, 189–199 (2013)
5. Gajdos, P., Platos, J.: GPU based parallelism for self-organizing map. In: Advances in Intelligent Systems and Computing, IHCI 2011, vol. 179, pp. 3–12 (2013)
6. Hasan, S., Shamsuddin, S.M., Lopes, N.: Machine learning big data framework and analytics for big data problems. Int. J. Adv. Soft Comput. Appl. **6**(2), 1–17 (2014)

7. McConnell, S., Sturgeon, R., Henry, G., Mayne, A., Hurley, R.: Scalability of self-organizing maps on a GPU cluster using OpenCL and CUDA. J. Phys. Conf. Ser. **341**, 12018 (2012)
8. Moraes, F.C., Botelho, S.C., Filho, N.D., Gaya, J.F.O.: Parallel high dimensional self organizing maps using CUDA. In: 2012 Brazilian Robotics Symposium Latin American Robotics Symposium, pp. 302–306 (Oct. 2012)
9. Khan, S.Q., Ismail, M.A.: Design and implementation of parallel SOM model on GPGPU. In: 2013 5th International Conference Computer Science Information Technology, pp. 233–237 (Mar. 2013)
10. Wang, H., Zhang, N., Créput, J.-C.: A Massive Parallel Cellular GPU Implementation of Neural Network to Large Scale Euclidean TSP. In: Castro, F., Gelbukh, A., González, M. (eds.) Advances in Soft Computing and Its Applications: 12th Mexican International Conference on Artificial Intelligence, MICAI 2013, Mexico City, Mexico, 24–30 November 2013, Proceedings, Part II, pp. 118–129. Springer, Berlin, Heidelberg (2013)
11. Faro, A., Giordano, D., Palazzo, S.: Integrating unsupervised and supervised clustering methods on a GPU platform for fast image segmentation. In: 2012 3rd International Conference Image Processing Theory, Tools Applications IPTA 2012, pp. 85–90 (2012)
12. Khronos OpenCL: OpenCL Specification (2014)
13. Brodtkorb, A.R., Hagen, T.R., Sætra, M.L.: Graphics processing unit (GPU) programming strategies and trends in GPU computing. J. Parallel Distrib. Comput. **73**(1), 4–13 (2013)
14. Mukherjee, S., Sun, Y., Blinzer, P., Ziabari, A.K., Kaeli, D.: A comprehensive performance analysis of HSA and OpenCL 2.0. In: 2016 IEEE International Symposium Performance Analysis System Software (April, 2016)