Journal of Applied Sciences

ISSN 1812-5654

OPEN ACCESS Journal of Applied Sciences

ISSN 1812-5654
DOI: 10.3923/jas.2017.204.211

Research Article

Evaluation of Parallel Self-organizing Map Using Heterogeneous System Platform

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

Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA, Shah Alam, 40450 Selangor, Malaysia

Abstract

Background: Self-organizing map (SOM) is a very popular algorithm that has been used as clustering algorithm and data exploration. The SOM consists of complex calculations where the calculation of complexity depending on the circumstances. Many researchers successfully improve SOM processing speed using discrete Graphic Processing Units (GPU) since the introduction of Compute Unified Device Architecture (CUDA) in 2007 and Open Computing Language (OpenCL) in 2009. In spite of excellent performance using GPU, there are performance issues in processing a large mapping size especially dealing with find the Best Matching Unit (BMU) and updating weightage. Additionally, the larger mapping size also could burden the processing through the usage of high memory capacity which leads to high rate memory transfer. Recently, heterogeneous systems, that soldered CPU and GPU together on a single chip are rapidly attractive the design paradigm for today's platform because of their remarkable parallel processing abilities. Therefore, this study evaluates parallel SOM performance on discrete GPU and heterogeneous system in order to improve the algorithm processing. **Materials and Methods:** Accordingly, this study demonstrates parallel SOM that comprises of three kernels. The parallel SOM then executes on two different platforms: (1) Discrete GPU platform and (2) Heterogeneous system platform. This study evaluates the outcomes of the computation experiments based on computation time and SOM quality measurements. **Results:** As a result, parallel SOM that executed on heterogeneous system platform is able to reduce the total processing time compared to discrete GPU platform when processing large mapping sizes and large data sets. **Conclusion:** More important, this study highlights how the proposed parallel SOM can improve the execution performance and maintain the SOM results when running on heterogeneous system.

Key words: Self-organizing map, Kohonen network, clustering, parallel self-organizing map, parallel computing, GPU computing, heterogeneous system, OpenCL

Received: September 08, 2016 Accepted: January 30, 2017 Published: March 15, 2017

Citation: Muhammad Firdaus B. Mustapha, Noor Elaiza Bt Abd Khalid and Azlan B. Ismail, 2017. Evaluation of parallel self-organizing map using heterogeneous system platform. J. Applied Sci., 17: 204-211.

Corresponding Author: Muhammad Firdaus B. Mustapha, Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA, Shah Alam, 40450 Selangor, Malaysia Tel: +6013-2075853

Competing Interest: The authors have declared that no competing interest exists.

Data Availability: All relevant data are within the paper and its supporting information files.

INTRODUCTION

Self-Organizing Map (SOM) is an unsupervised neural network that has been used as data analysis method. It is widely applied to clustering problem and data exploration in various areas of problems with remarkable abilities to remove noise, outliers and deal with missing values[1]. The SOM algorithm is based on nonlinear projection mapping which reduce the dimensions of high-dimensional data to two-dimensional data (2D) by producing a topology map. The SOM is categorized as data reduction technique performs data reduction on two ways: (1) Reducing the number of dimension (Projection) and (2) Number of observations (Quantization), at the same time preserving the structure and useful information in the dataset[2]. There are other techniques in this category such as Multi Dimensional Scaling (MDS) and Principal Component Analysis (PCA). However the SOM is differs from these techniques because it has learning capability and combines the two methods in the algorithm that are projection and quantization. The learning nature of algorithm in SOM is to allow the information constantly updated in order to improve the quality of final results.

On the other notes, Graphic Processing Unit (GPU) is a many core processor consisting hundreds or even thousands of compute cores[3] has been used to process the applications of scientific computing and scientific simulations or also called General Purpose Graphic Processing Unit (GPGPU)[3]. Originally, GPU has designed for graphic applications through programming environments such as DirectX and OpenGL[4]. The GPU computing has proven to be optimized to produce high throughput of floating point operations using large data in graphic applications. Since the introduction of GPU programming frameworks such as of Compute Unified Device Architecture (CUDA) in 2007 and Open Computing Language (OpenCL) in 2009[5], the GPU have become popular in improving the processing of algorithms in various fields.

The OpenCL is a framework of parallel programming that can be used for programming a heterogeneous collection of Central Processing Units (CPU), GPU and other discrete computing devices organized into a single platform[6]. An OpenCL program is executed on a host and the host is connected to one or more GPU. 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, due to the restrictions imposed by past GPU architectures, most of these frameworks treat the GPU as an accelerator which can only work under close control of the Central Processing Unit (CPU). Further, the communication protocol between a CPU and a GPU is source of high latency and becomes a performance bottleneck. This problem can be noticed when using OpenCL 1.0 where the memory management is relied on the programmer to take care of data movement between the CPU and the GPU. The main problem in performance for OpenCL 1.0 applications is data transfers between the host code and device code[7].

The most recent technology, heterogeneous systems, that soldered CPU and GPU 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[7]. 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 GPU as a first-class computing device which allows the GPU to manage their own resources as well as access some of the CPU resources. OpenCL 2.0 introduced Shared Virtual Memory (SVM) which allows the host and the device to share a common virtual address range[8]. This reduces overhead by eliminating deep copies during host-to-device and device-to-host data transfers. Deep copies involves completely duplicating objects in memory[7].

Many researchers are trying to take advantages of GPU computing to execute SOM algorithm in parallel manner. As a result, Hasan *et al.*[9] addressed the larger mapping size and feature dimensions, the slower the computation time for both CPU and GPU. Some researchers agreed that GPU variant shows the significant speed up for large data compared to CPU variant[4,10,11]. Both comparisons are proven that GPU computing achieves better performance in terms of computation time. Despite its excellent performance, there are performance issues in processing a large map, especially when dealing with winner-search and updating weightage of neurons on the map[1]. Additionally, the larger dataset and mapping size degrades its performance whilst increases its memory usage which leads to high rate memory transfer[9,12]. On the other hand, datasets features also have a great influence to the SOM processing[9].

The SOM algorithm comprises of dependency tasks which make it suitable to decompose. To enable parallelism of SOM, some researchers attempt to decompose several steps of the algorithm. Most of the studies are found in the literature perform decomposition on calculate distance and find the

BMU. This step obviously consists of many computations and it independent each other's. Hasan *et al.*[9] and De *et al.*[10] decomposed calculate Euclidean distance and Best Matching Unit (BMU) searching process. Some researchers tried to decompose calculate distance, find BMU and update the neuron's weights[10,11,13,14]. Meanwhile; Lachmair *et al.*[12] and De *et al.*[10] decomposed initialize neuron weights. From the literature shows that there are three major steps have been decomposed, calculate distance, find the BMU and update the weights. Some researchers are used different configurations of decomposition on these three steps.

This study demonstrates parallel SOM that comprises of three kernels. The parallel SOM then executes on discrete GPU using OpenCL 1.0 and heterogeneous system platform that employs OpenCL 2.0 with the interest to study the performance of the algorithm. The main purpose of this study is to improve processing speed of parallel SOM using heterogeneous system platform. The study evaluates the outcomes of the computation experiments based on computation time and SOM quality measurements. The significance of this study can be summarized as follows:

- This study presents three kernels of parallel SOM specifically calculate distance, find BMU and update weights
- This study describes specific set of performance of parallel SOM appropriate for heterogeneous system using GPU and implemented in OpenCL
- This study evaluates the proposed parallel SOM in terms of execution time and SOM quality measurements on discrete GPU and heterogeneous system

MATERIALS AND METHODS

This study describes about the parallel SOM, parameter setting of GPU programming framework and experimental setup.

Parallel self-organizing map: This study 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 executed on discrete GPU and heterogeneous system. The discrete GPU system will be run on OpenCL 1.0 while the heterogeneous system will be run on OpenCL 2.0.

Calculate distance kernel: Calculate distance kernel includes distance calculation step in SOM algorithm where it calculate the distance between neurons and current input vector. The amount of neurons on the SOM is represented by the same ammount of work-items on the GPU. As such, each work-item of the kernel is responsible for finding the distance between a single neuron and the current input vector. The distance calculation of SOM algorithm can be realized by using algorithm in Fig. 1.

This study applies Manhattan distance calculation, MD to calculate distance using Eq. 1:

$$\text{MD}(x,b) = (x_1-b_1)+(x_2-b_2)+\ldots+(x_n-b_m) \quad (1)$$

where, x is an input vector with n attributes and b is neurons vector with m weights. Afterward the sum of the differences between each of the components of the current input vector and the current neuron's vector is calculated as described in the equation. The result of this calculation is stored in a distance map array. The distance map array will be employed by find BMU kernel on the next step.

Find BMU kernel: This kernel applies two stage parallel reduction[15] with the aims of finding BMU in parallel which performs in stages. Generally, the BMU is a calculation to select a neuron with the smallest distance as a winner neuron[1], C_k where:

$$C_k = \text{argmin (MD)} \quad (2)$$

Algorithm in Fig. 2 describes the find BMU step. Firstly, the kernel acquires the number of work units, chunk_size for each Compute Unit (CU) from the host. chunk_size is calculated using the following Eq. 3[16]:

$$\text{chunk\_size (cs)} = \frac{\text{Amount of work unit}}{\text{Amount of compute unit (CU), (aCU)}} \quad (3)$$

where, the amount of work unit is equivalent with amount of neurons on SOM map and amount of CU on device is depending on the device type. All the CUs should have almost the same number of work units. 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, local_chunk_size as shown in Eq. 4[16]. This kernel then acquires the values of distance map from the host which were stored into distance map array previously. Each work-item in the work-groups will find the minimum distance among the distance values covered by the work groups using Eq. 5. The minimum distance value identified by the kernel is stored in a local array.

At the second stage, each CU will search the minimum distance in the local array using Eq. 6. This local array contains minimum distances from all the work-groups under the same CU. All the CUs in the device will have their BMU. The minimum value identified at this stage will be stored into a global array. Finally, the host read the global array that contains the BMUs from all the CU. Next, the host finds the minimum distance value from the appropriate array as the winning neuron among the winners through Eq. 7.

```
Begin
  for k = 1 to $B_m$ do in parallel

    $MD (x, b) = (x_1-b_1)+(x_2-b_2)+...+ (x_n-b_m)$        (1)

    dist_array[i] = MD(x,b)
  end for
End
```

Fig. 1: Algorithm to calculate distance

```
Begin
    $\text{chunk\_size (cs)} = \frac{\text{Amount of work unit}}{\text{Amount of compute unit (CU), (aCU)}}$        (3)

    $\text{local\_chunk\_size (lcs)} = \frac{\text{chunk\_size}}{\text{Amount of work group (awg)}}$        (4)

    $C_{lcs}$ = argmin (MD)    (5)    do in parallel
      Store the winner into local_winner_array[awg]
    $C_{cs}$ = argmin (local_winner_array [awg])    (6)    do in parallel
      Store the winner into global array, global_winner_array[aCU]

    $C_{aCU}$ = argmin (global_winner_array [aCU])        (7)
End
```

Fig. 2: Algorithm to find BMU

```
Begin
  Determine neighborhood value
  for k=1 to $B_n$ do in parallel
      $W_{update} = w_{current}$+neighborhood_function[$x(i)-w_{current}$]    (8)  do in parallel
End
```

Fig. 3: Algorithm to update weights

Update weights kernel: This kernel deals with updating the weight vectors associated to each neuron using the original update weight. Algorithm in Fig. 3 defines the update weight step. The algorithm begin with determine the neighborhood value which includes learning rate. The learning rate describes how much a neuron's vector is changed during an update according to how far away the 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. The learning rate values can be an inverse time, linear or power function. Equation 8 describes the formulation of update weight[1]. This study applies Gaussian function as a neighborhood function. The weight vectors of units in the neighborhood of the winner are modified according to a spatial temporal neighborhood function.

Parameter setting of GPU programming framework: This study describes the different parameters setting of parallel SOM on OpenCL 1.0 and OpenCL 2.0. The parameter setting can be divided into four parts, pointer declaration, allocating the pointer, accessing the parameters from host and passing the parameters to kernel. Table 1 and 2 show the parameter setting that impose to OpenCL 1.0 and OpenCL 2.0, respectively. Both parameters setting clearly show the different between OpenCL 1.0 and OpenCL 2.0 on every part. The most anticipating part of OpenCL 2.0 is when accessing parameter values from the host, where the parameter can be

Table 1: Parameter setting on OpenCL 1.0

Pointers declaration	cl::Buffer winner_distance_array_buffer

Allocating the pointers and initializing the buffer	winner_distance_array = (float *)malloc(sizeof(float)*compute_units);
	winner_index_array = (int *)malloc(sizeof(int)*compute_units);
	for (int i = 0; i < compute_units; I++)
	{winner_distance_array[i] = FLT_MAX;
	winner_index_array[i] = -1}
	
	winner_distance_array_buffer = cl::Buffer(device_context, CL_MEM_READ_WRITE CL_MEM_USE_HOST_PTR, sizeof(float)*compute_units, winner_distance_array, &err);

Accessing from host	Read from host:
	err = command_queue.enqueueReadBuffer(winner_distance_array_buffer, CL_TRUE, 0, sizeof(float)*compute_units, winner_distance_array)
	Write from host:
	err = command_queue.enqueueWriteBuffer(winner_distance_array_buffer, CL_TRUE, 0, sizeof(float)*compute_units, winner_distance_array)

Passing to kernel	manhattan_distance_kernel.setArg(0, winner_distance_array_buffer)

Table 2: Parameter setting on OpenCL 2.0

Pointers declaration	float* map_buff;
	float* distance_map_buff;
	float* input_buff

Allocating the pointers	input_buff = (float*)clSVMAlloc(oclobjects.context, CL_MEM_READ_WRITE|CL_MEM_SVM_FINE_GRAIN_BUFFER, input_size*input_vector_length*sizeof(float), 0)

Accessing from host	for (int i = 0; i < input_size*input_vector_length; i++)
	{input_buff[i] = input[i]}

Passing to kernel	err = clSetKernelArgSVMPointer(executable.kernel, 0, input_buff)

Table 3: Experimental setting

Dataset parameters		SOM parameters		
No. of samples	No. of parameters	Iterations	Mapping size	Quality measurements
5000	3	30	10×10	Time
10000			20×20	Quantization error
15000			30×30	Topographic error
			40×40	

SOM: Self-organization map, QE: Quantization error, TE: Topographic error

Table 4: Intel i7-6700HQ details

Model	Intel i7-6700HQ
No. of cores/threads	4/8
CPU clock speed	2.6-3.5 GHz
CPU micro-architecture	14 nm
GPU clock speed	350-1050 MHz
No. of execution units	24

accessed as normal as C^{++} variables. This feature is enabled by SVM through clsMAlloc at allocating the pointers. The pointers can be shared between host code and device code without calling any wrapping function[8,17]. The improvement version shows the codes are more similar to normal C^{++} coding rather than depending on specific function. Furthermore, the parameter setting of OpenCL 2.0 is simpler which could help to improve programmability of programmer.

Experimental setup: This study applies Bank Marketing dataset taken from UCI Machine Learning Repository. The dataset is divided into three different number of samples; 5000, 10000 and 15000. All the samples have three parameters. Each sample is tested using the same number of iterations and mapping size as shown in Table 3. The number of iterations is set to 30 due to processor constraint. The objective of these experiments is to examine the performance of parallel SOM that tested on OpenCL 1.0 and OpenCL 2.0. These experiments will be evaluated based on processing time and quality of SOM results. The processing time has taken from executing three kernels and total processing time. The quality measurement SOM results is measured using Quantization Error (QE) and Topographic Error (TE). The experiments were conducted on a laptop equipped with Intel i7-6700HQ processor, 16GB 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 four CPU cores and 24 number of execution units placed at GPU. Table 4 shows detailed information of the processor.

RESULTS AND DISCUSSION

Firstly, the results are highlighted in term of processing time performance. Three series of experiments have been conducted with three different data sets sizes and mapping sizes. In order to facilitate the reader understanding, this study labels the experiment of parallel SOM using OpenCL 1.0 with SOMonCL1 and parallel SOM using OpenCL2.0 with SOMonCL2. The performances of time processing are depicted in Fig. 4-6.

From the graphs, the calculate distance kernel and update weight kernel of SOMonCL2 are outperforming the time of SOMonCL1. This trend can be seen on the graph for each mapping size. The time is increasing when the experiments use larger mapping size for both SOMonCL1 and SOMonCL2. Both of kernels uses the same work-item of GPU representation which amount of work-items are equal to the number of neurons in the SOM map. The performance of SOMonCL2 is triggered by parameter setting in OpenCL 2.0. The SOMonCL2 utilizes SVM in OpenCL 2.0 which resulting both kernels capable to reduce the processing time compared

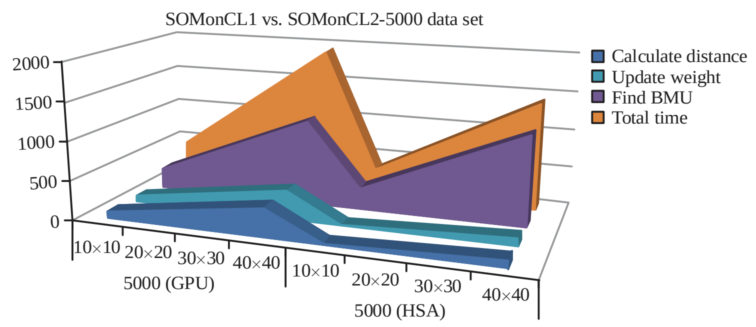

Fig. 4: Performance of execution time on 5000 data sets

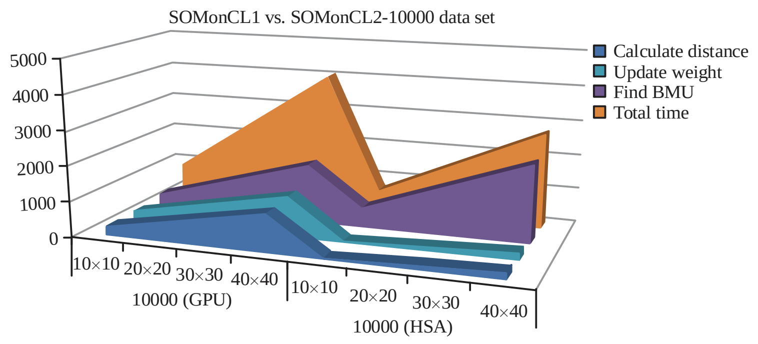

Fig. 5: Performance of execution time on 10000 data sets

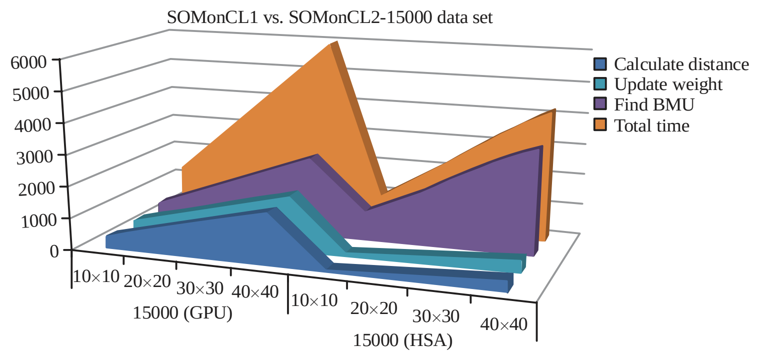

Fig. 6: Performance of execution time on 15000 data sets

to SOMonCL1. The SVM feature manages the 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[7].

Meanwhile, the find BMU kernel achieves in opposite way because the SOMonCL2 consumes longer time compared to SOMonCL1. The find BMU kernel is more complex algorithm because it comprises of two level parallelisms that involve work group processing in GPU. Moreover, this kernel contains read operation from GPU to host which could consume higher time compared to other kernels. Even though the SOMonCL2 is drove by SVM in OpenCL 2.0, it still unmanageable to reduce the time from SOMonCL1 at find BMU kernel. One of the reasons, the two level of parallelism requires synchronization point in order to harmonize the SVM memory. The SOMonCL2 use clFinish function for synchronization point[7]. It has been tested in the experiments that this function is useful for SOMonCL2 where the final result could not be accomplished if the function is removed. Furthermore, it can be seen clearly on the graph, the find BMU kernel consumes the highest time among of three kernels which resulting the total time increases for both SOMonCL1 and SOMonCL2. However, the SOMonCL2 attains better in total processing time compared to SOMonCL1. This performance can be realized in Fig. 7 that indicates the performance of SOMonCL2 better in total time about 30-40%.

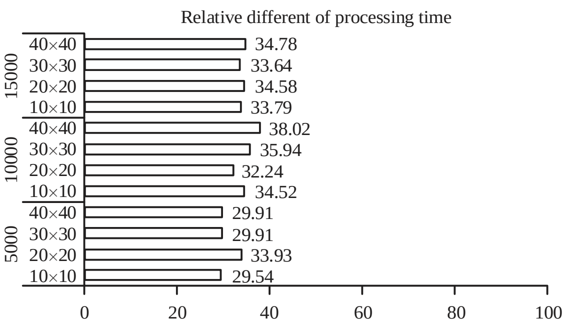

Fig. 7: Relative different of processing time between SOMonCL1 and SOMonCL2

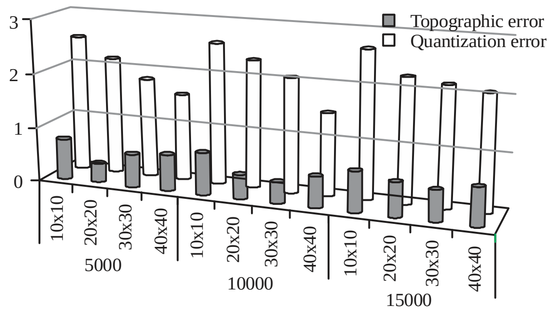

Fig. 8: Performance of SOM quality measurement on SOMonCL1

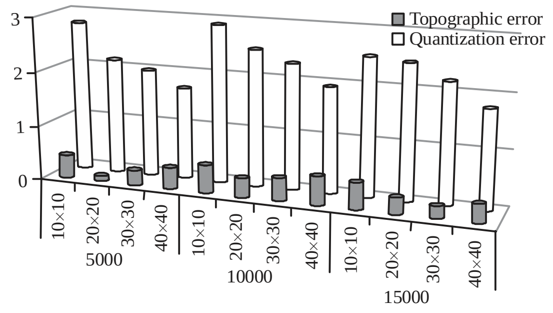

Fig. 9: Performance of SOM quality measurement on SOMonCL2

Even the SOMonCL1 surpasses the performance of SOMonCL2 in find BMU kernel, the total time of execution of SOMonCL2 is substantially reduced by calculate distance kernel and update weight kernel.

Overall, all of the three graphs of time processing performance indicate that the increasing of the data sets size will lead to rise the computation time across mapping sizes. As a result, the SOM complexity increases when dealing with larger data set sizes and mapping sizes for both SOMonCL1 and SOMonCL2. This issues has been highlighted by researchers[9,12]. However, the SOMonCL2 is capable to reduce processing time when dealing with larger dataset and mapping size compared to SOMonCL1. As pointed by Kohonen[1], the find BMU and update weight are the main issues in SOM processing. Hence, SOMonCL2 is able to perform better at calculate distance kernel and update weight kernel but still incontrollable to reduce time at find BMU kernel.

On the other note, Fig. 8 and 9 describe the analysis on SOM quality measurement based on TE and QE for both SOMonCL1 and SOMonCL2 respectively. The graphs show that the trend of QE values is decreasing when the mapping size growth for the same data set. However, the QE value rises when tested on larger data set that uses the same mapping size. Meanwhile, both graphs depict quite the same trend in terms of TE measurements. Overall, the values of TE from the both graphs are below one which indicates the data are sufficiently spreading to whole map of the SOM map. Kohonen[1] states that, mapping size could influent the SOM quality measurement. The results show that larger mapping size will reduce QE meanwhile TE values depending on data spreading on the map.

However, in order to find the best mapping sizes, both of TE and QE values should be designated with minimum values as pointed in[18]. The best mapping sizes of SOMonCL1 are 20×20 for 5000 data sets and 30×30 for both 10000 and 15000 data sets. Meanwhile, the ideal mapping size of SOMonCL2 are 20×20 for 5000 and 10000 data sets and 30×30 for 15000 data sets. Generally, the ideal mapping size can be seen increasing in term of the mapping size when larger dataset is imposed. For example, the ideal map of dataset 5000 is 20×20, 10000 and 15000 are 30×30. Consequently, both SOMonCL1 and SOMonCL2 show fairly the same trends of SOM quality measurement. These could indicate the quality of results from SOMonCL1 and SOMonCL2 are on the par.

CONCLUSION

This study is carried out with the aims to evaluate the performance of both parallel SOM on discrete GPU (SOMonCL1) and heterogeneous system (SOMonCL2). As a result, the main finding of this study is SOMonCL2 capable to reduce the total processing time from discrete GPU platform when processing large mapping sizes and large data sets. The processing of SOMonCL2 is drastically improved at calculate distance kernel and update weight kernel. Overall, the SOMonCL2 shows improvements compared to SOMonCL1 in terms of reducing the total processing time about 30-40%. On the other hand, SOMonCL2 is capable to generate equivalent quality of SOM map with SOMonCL1. The main issue arise from this study is the processing time increases at Find BMU kernel compared to SOMonCL1 due to synchronization point. The future improvement may focus on this issue in order to reduce processing time.

ACKNOWLEDGMENTS

This study was funded by Ministry of Higher Education (MOHE) of Malaysia, under the FRGS, grant No. FRGS/1/2015/ICT02/UITM/02/6 and academic staff Bumiputera Training Scheme (SLAB). The authors also would like to thank the Universiti Teknologi MARA for supporting this study.

REFERENCES

1. Kohonen, T., 2013. Essentials of the self-organizing map. Neural Netw., 37: 52-65.
2. Arribas-Bel, D., K. Kourtit and P. Nijkamp, 2013. Benchmarking of world cities through self-organizing maps. Cities, 31: 248-257.
3. Perelygin, K., S. Lam and X. Wu, 2014. Graphics processing units and open computing language for parallel computing. Comput. Electr. Eng., 40: 241-251.
4. Wittek, P. and S. Daranyi, 2013. Accelerating text mining workloads in a map reduce-based distributed GPU environment. J. Parallel Distrib. Comput., 73: 198-206.
5. Kirk, D.B. and W.W. Hwu, 2013. Programming Massively Parallel Processors: A Hands-on Approach. 2nd Edn., Elsevier, New York, ISBN: 978-0-12-381472-2, Pages: 514.
6. Gaster, B.R. L. Howes, D. Kaeli, P. Mistry and D. Schaa, 2012. Heterogeneous Computing with OpenCL. 2nd Edn., Morgan Kaufmann Publisher Inc., San Francisco, CA, USA., ISBN: 978-0-12-387766-6.
7. Mukherjee, S., Y. Sun, P. Blinzer, A.K. Ziabari and D. Kaeli, 2016. A comprehensive performance analysis of HSA and OpenCL 2.0. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, April 17-19, 2016, IEEE.
8. Khronos OpenCL Working Group, 2014. The OpenCL specification. Version 2.0. The Khronos Group Inc. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.
9. Hasan, S., S.M. Shamsuddin and N. Lopes, 2014. Machine learning big data framework and analytics for big data problems. Int. J. Adv. Soft Comput. Appl., 6: 1-14.
10. De, A., Y. Zhang and C. Guo, 2016. A parallel adaptive segmentation method based on SOM and GPU with application to MRI image processing. Neurocomputing, 198: 180-189.
11. Richardson, T. and E. Winer, 2015. Extending parallelization of the self-organizing map by combining data and network partitioned methods. Adv. Eng. Softw., 88: 1-7.
12. Lachmair, J., E. Merenyi, M. Porrmann and U. Ruckert, 2013. A reconfigurable neuroprocessor for self-organizing feature maps. Neurocomputing, 112: 189-199.
13. Mustapha, M.F.B., N.E.B. Khalid and A.B. Ismail, 2015. Time consuming factors for self-organizing map algorithm. Proceedings of the International Conference on Information Technology and Society, June 8-9, 2015, Kuala Lumpur, Malaysia, pp: 84-92.
14. Khan, S.Q. and M.A. Ismail, 2013. Design and implementation of parallel SOM model on GPGPU. Proceedings of the 5th International Conference on Computer Science and Information Technology, March 27-28, 2015, IEEE., pp: 233-237.
15. Catanzaro, B., 2010. OpenCL^{TM} optimization case study: Simple reductions. Khronos, 2010. http://developer.amd.com/resources/articles-whitepapers/opencl-optimization-case-study-simple-reductions/.
16. Davidson, G., 2015. A parallel implementation of the self organising map using OpenCL. University of Glasgow, March 27, 2015.
17. Kyriazis, G., 2012. Heterogeneous system architecture: A technical review. pp: 1-18, http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/ 10/hsa10.pdf
18. Chattopadhyay, M., P.K. Dan and S. Mazumdar, 2012. Application of visual clustering properties of self organizing map in machine-part cell formation. Applied Soft Comput., 12: 600-610.