talenta PUBLISHER

DATA SCIENCE
Journal of Computing and Applied Informatics

Enhancing Performance of Parallel Self-Organizing Map on Large Dataset with Dynamic Parallel and Hyper-Q

Alexander F. K. Sibero[1], Opim Salim Sitompul[2*], and Mahyuddin K.M. Nasution[3]

[1]*Graduate Program of Computer Sience, Universitas Sumatera Utara, Medan, Indonesia*
[2,3]*Department of Information Technology, Universitas Sumatera Utara, Medan, Indonesia*

Abstract. Self-Organizing Map (SOM) is an unsupervised artificial neural network algorithm. Even though this algorithm is known to be an appealing clustering method, many efforts to improve its performance are still pursued in various research works. In order to gain faster computation time, for instance, running SOM in parallel had been focused in many previous research works. Utilization of the Graphics Processing Unit (GPU) as a parallel calculation engine is also continuously improved. However, total computation time in parallel SOM is still not optimal on processing large dataset. In this research, we propose a combination of Dynamic Parallel and Hyper-Q to further improve the performance of parallel SOM in terms of faster computing time. Dynamic Parallel and Hyper-Q are utilized on the process of calculating distance and searching best-matching unit (BMU), while updating weight and its neighbors are performed using Hyper-Q only. Result of this study indicates an increase in SOM parallel performance up to two times faster compared to those without using Dynamic Parallel and Hyper-Q.

Keyword: Dynamic Parallel, Hyper-Q, Parallel SOM, Self-Organizing Map

Abstrak. *Self-organizing map (SOM) adalah sebuah algoritma jaringan syaraf tiruan tanpa supervisi. Namun demikian, meskipun algoritma ini dikenal sebagai metode klustering yang menjanjikan, banyak upaya yang telah dilakukan untuk memperbaiki kinerjanya yang dilakukan di berbagai penelitian. Untuk memperoleh waktu komputasi lebih cepat misalnya, para peneliti terdahulu berfokus pada upaya menjalankan SOM secara paralel. Pemanfaatan Graphics Processing Unit (GPU) sebagai mesin perhitungan paralel juga semakin membaik. Akan tetapi, waktu komputasi keseluruhan pada SOM paralel masih saja dirasa belum optimal dalam pemrosesan dataset yang besar. Dalam penelitian ini, penulis mengusulkan satu kombinasi pendekatan Dynamic Parallel dan Hyper-Q untuk lebih meningkatkan kinerja SOM paralel dalam hal kecepatan waktu komputasi. Dynamic Parallel dan Hyper-Q dimanfaatkan pada proses perhitungan jarak dan pencarian best-matching unit (BMU), sementara perbaikan bobot dan tetangga sekelilingnya dilakukan hanya dengan menggunakan Hyper-Q. Hasil studi ini menunjukkan bertambahnya kinerja paralel SOM hingga dua kali lebih cepat dibandingkan dengan tanpa menggunakan Dynamic Parallel dan Hyper-Q.*

Kata Kunci: *Dynamic Parallel, Hyper-Q, Self-Organizing Map, SOM Paralel*

Received 27 December 2017 | Revised 3 January 2018 | Accepted 28 January 2018

*Corresponding author at: Department of Information Technology, Faculty of Computer Science and Information Technology, Universitas Sumatera Utara, Jalan Alumni No. 9 Kampus USU, Medan 20155, Indonesia
E-mail address: opim@usu.ac.id

ISSN: 2580-6769 | e-ISSN: 2580-829X | DOI: 10.32734/jocai.v2.i2-324
Journal Homepage: https://talenta.usu.ac.id/JoCAI

1. Introduction

Self-Organizing Map is popularly used to complete classification and clustering on a dataset. Clustering on the dataset is performed by training the weight of data until its value becomes equal to its input [1]. The first step of SOM is to map the dataset into a set of nodes that are placed on topographic maps of a certain dimension. Training on SOM is intended to find similarity between input nodes and output nodes on a competitive basis. The closest similarity between the input node and the output node is called the best-matching unit (BMU). Furthermore, the node that was selected as the BMU will pull other nodes in the surrounding neighbors to become closer by changing the value of the weight [2]. Finding BMU nodes is an exhaustive process by calculating and comparing distances between the input nodes and the output nodes on the topographic maps. The amount of time spent in finding BMU on large-dimension topographic maps are significantly affect the overall computing time of the clustering process, compared to those in small-dimensional topographic maps [3].

Many efforts to obtain optimal computing time had been pursued in many research works, especially in the process of looking for BMU on large datasets. Faster method has been achieved by performing the process of finding BMU and updating the weights in parallel. While early parallel computing uses parallel technology on the CPU, further development of parallel computing is currently focused on the use of GPUs that have more parallel computing units[4]. Nowadays, the emergence of the OpenCL and CUDA frameworks bring many parallel computing functions to improve GPU utilization as a general-purpose computing engine known as the General Purpose Graphics Processor Unit (GPGPU) [5].

The development of nVidia technology invented the Kepler generation GPU with the ability of Dynamic Parallel [6] and Hyper-Q [7]. Both of these capabilities yield a more concurrent computing and efficiency in utilizing GPU. The work in this study tries to get parallel SOM performance improvement using Dynamic Parallel (DP) and Hyper-Q (HQ). The combination of DP and HQ is used in finding BMU and updating weight.

This paper is structured as follows. In section 2 we present a brief overview of some previous research works. Section 3 briefly introduced the Dynamic Parallel and Hyper-Q approaches used in this research. Section 4 presents the proposed methodology, continued by result and discussion in section 5. Section 6 as the last section provides the conclusion and future work of this study.

2. Previous Research

Previous research works on enhancing SOM have successfully utilized CPU parallelization. The development of computational technology is now directing parallel SOM research by utilizing GPU usage. The utilization of GPU as architecture in parallel SOM had been presented in various researches works (*see* for examples [3, 8, 9]), whereby the process of finding BMU and updating weights was executed in three kernels. The three kernels perform the task of calculating the Euclidean distance, finding the BMU using parallel reduction and updating neighbor weights. Another parallel SOM architecture [10, 11] were implemented by dividing the calculation into two kernels, the first kernel calculates the Euclidean distance and finds the BMU and the second kernel updates the neighbor weights. Yet another architecture works were using one kernel to calculate Euclidean distance and find BMU, while neighbor weight update was performed on the CPU [12].

Experiments to reduce computational time are also conducted with a combination of common methods used in SOM, such as network partition and data partition methods that are run in parallel [11]. Another method that is also utilized is parallel reduction as a solution to reduce computational time [13]. Other methods such as *coarse-grain buffer* and *fine-grain buffer* are also used to reduce computing time of distance calculation and finding BMU [8].

Other improvements are also made to the weight update. How to find neighboring nodes around BMU using the technique performed by [10] can also reduce computational time. Utilizing streams and shared memory usage has also managed to improve the parallel performance of SOM [8, 12].

Utilization of Dynamic Parallel (DP) and Hyper-Q (HQ) technology on Kepler generation has also been conducted by [6] in some experiments which perform several algorithms recursively, resulting varying results. The effect of DP performance depends on the number of thread divergences contained in the algorithm [13]. Hyper-Q utilization is deployed to improve performance and efficiency by running processes concurrently on the GPU. Improved performance with Hyper-Q utilization that runs those processes simultaneously adds a record of success in its application [14].

3. Dynamic Parallel and Hyper-Q Approaches

The ability to perform parallelization dynamically (Dynamic Parallelism) developed by nVidia has allowed GPU to freely build a work mechanism within one of its processes. Configuration as well as usage of threads created is performed by GPU alone with no CPU intervention (*see* Figure 1a).

Dynamic parallelization of the nVidia GPU allowing one kernel process calls another different kernel process. This recursive call could not be performed on previous GPU generation. Using Dynamic Parallel, CPU usage becomes lower, allowing the CPU to be utilized by another process.

Another features belongs to nVidia GPU is the ability to perform CPU instruction in parallel. Simultaneously, Hyper-Q performs CPU commands to be executed in GPU. Theoretically, Hyper-Q reduces delayed computation time by exchanging utilization of resources in GPU. Using Hyper-Q, usage of each stream could be optimized [15] (*see* Figure 1b).

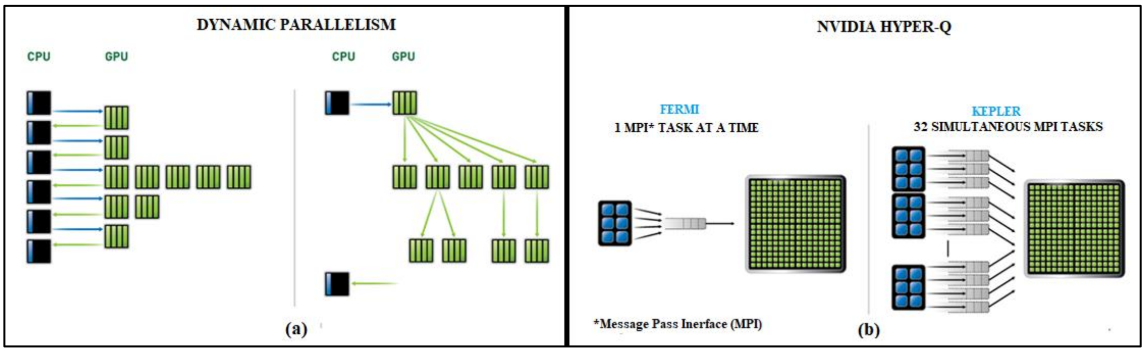


Figure 1. nVidia Dynamic Parallel and Hyper-Q

4. Methodology

The architecture proposed in this study is to find BMU and to update weight using Dynamic Parallel (DP) and Hyper-Q (HQ). Finding BMU is performed on the GPU consists of 3 kernels running on the device wrapped in DP and HQ. An update of the neighboring weights is performed on a kernel that runs concurrently using HQ (Figure 2).

4.1. Finding BMU

Dynamic Parallel (DP) and Hyper-Q (HQ) allow *kernel* to run concurrently on the GPU. The DP structure allows the kernel to run in other kernels [6]. The use of HQ utilizes parallel paths on the host (CPU) in order to run concurrently on the device (GPU) kernel [7]. Finding BMU in the parallel SOM in this study is divided into 3 kernels.

The first kernel calculates the distance between the input nodes and the output nodes using the Euclidean distance. In these kernel weights of the input and output nodes are kept into two arrays. The sum resulted from the first kernel process is stored in a new array. In order for a kernel execution to run properly, the use of blocks and threads should be appropriate. In this kernel the block used is one, with the number of threads as many as the maximum number of threads per block device.

The result of calculating Euclidean distance is continued in the second kernel using *Parallel Reduce Interleaved Address* method. This method is selected because it can complete the summation in the array [16]. In this second kernel the number of threads used is the same as the number of data features. The numbers of blocks used are determined by finding the largest multiplier value factor of the number of threads used, which is divisible by the number of array elements from the Euclidean distance calculation results.

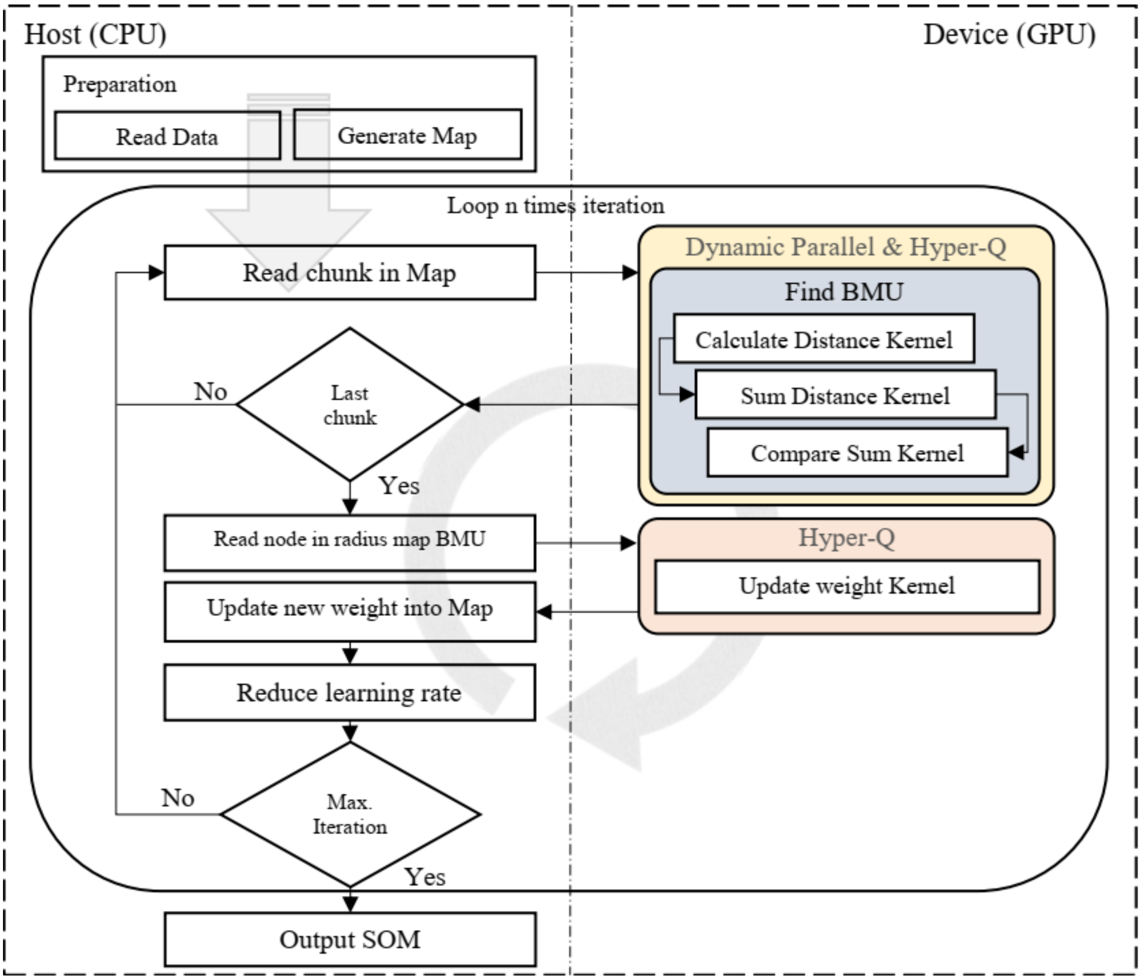
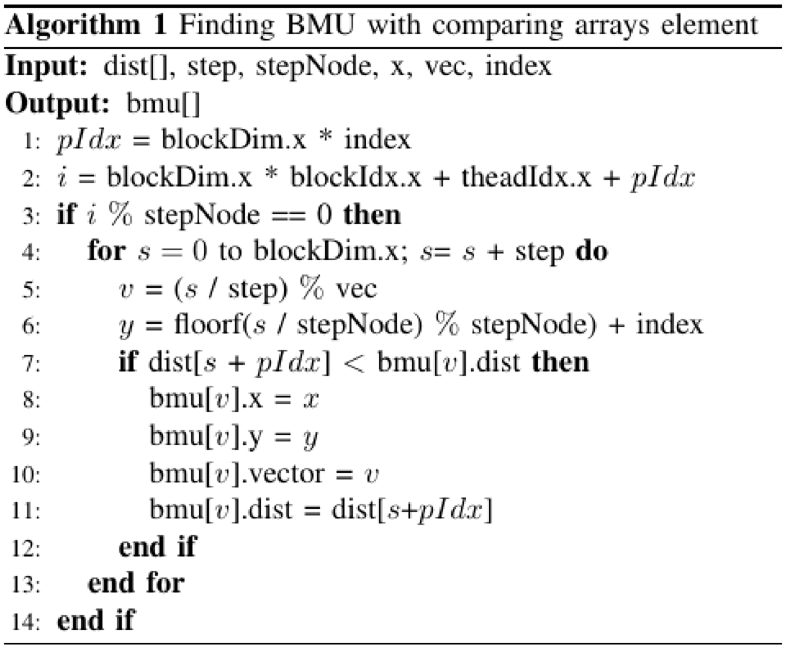


Figure 2. Parallel SOM Architecture

Algorithm 1 illustrates how to find BMU by comparing elements of arrays. The sum of the arrays is calculated using the *Parallel Reduce Interleaved Address* method. This method stores the result in the array element of index 0 and the next index value is multiple of the number of data features. The next step is to get the BMU value in the array that has been summed from the previous kernel. The approach used is to compare the array element of index 0 with the next index specified. Arrangement of array elements is performed by modifying the *Parallel Reduce Interleaved Address* method. Modifications are made by adding 2 step parameters. The first step parameter is to find the vector position of the array element; the second step parameter looks for the *y* coordinate position of the array element. The lowest value of the benchmark result is used as a temporary BMU value to compare with the BMU value in the next process.



Algorithm 1 Finding BMU with comparing arrays element

```
Input: dist[], step, stepNode, x, vec, index
Output: bmu[]
$pIdx$ = blockDim.x * index
$i$ = blockDim.x * blockIdx.x + theadIdx.x + $pIdx$
if $i$ % stepNode == 0 then
    for $s = 0$ to blockDim.x; $s$= $s$ + step do
        $v$ = ($s$ / step) % vec
        $y$ = floorf($s$ / stepNode) % stepNode) + index
        if dist[$s$ + $pIdx$] < bmu[$v$].dist then
            bmu[$v$].x = $x$
            bmu[$v$].y = $y$
            bmu[$v$].vector = $v$
            bmu[$v$].dist = dist[$s$+$pIdx$]
        end if
    end for
end if
```

The problem with large array calculations is the limited number of threads that can be used; the array calculation can not be performing entirely simultaneously. This problem can be solved by setting the array index numbering during kernel execution. Determination of array index numbering is performed by dividing the number of array elements, which is a multiplication of number of blocks and thread per block used in the kernel.

Another problem arising from simultaneous kernel calling is that each of the executed kernels precedes each other which are called race condition [17]. In the comparison process that occurs in the third kernel, the race condition event can affect the results of finding BMU. Solution to this problem is by synchronizing the kernel and thread. Synchronization in the kernel and the thread is required to ensure the threads are completing the entire process before the next command.

4.2. Updating Neighborhoods

BMU obtained from the previous process is continued by neighbor updates. Searching neighbor nodes of BMU from previous method are performed by comparing distance coordinates between BMU nodes and another nodes contained in topographic maps, one by one. The result of calculating both coordinate distances which is less than the radius map value (rm) is the neighboring node to be updated. This research proposes another way to shorten the searching for neighbor nodes. The proposed way to determine minimum and maximum coordinates of x and y could be seen in Equations (1)-(2) and Equations (3)-(4), respectively.

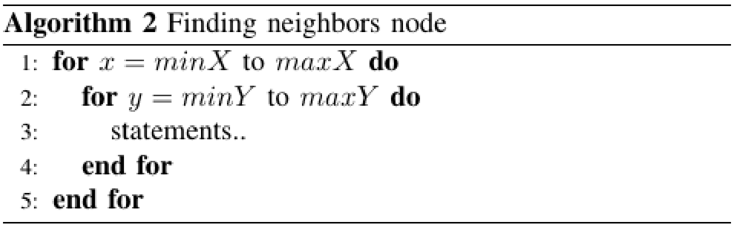
$$\min X = \begin{cases} bmu.X - rm, & otherwise \\ 0, & bmu.X - rm < 0 \end{cases} \tag{1}$$

$$\max X = \begin{cases} bmu.X + rm, & otherwise \\ max.X, & bmu.X + rm > max.X \end{cases} \tag{2}$$

$$\min Y = \begin{cases} bmu.Y - rm, & otherwise \\ 0, & bmu.Y - rm < 0 \end{cases} \tag{3}$$

$$\min Y = \begin{cases} bmu.Y + rm, & otherwise \\ max.Y, & bmu.Y + rm > max.Y \end{cases} \tag{4}$$

The following algorithm 2 illustrates how to find the neighboring node.



Algorithm 2 Finding neighbors node

```
for $x = minX$ to $maxX$ do
    for $y = minY$ to $maxY$ do
        statements..
    end for
end for
```

Nodes found from search results are then copied into an array. The SOM parameters used in calculations such as input nodes, learning rate and gaussian neighbor function are copied into each array. Each of these arrays is then passed on the kernel update weights to be calculated. The results obtained from the calculation of the kernel are then stored into topographic maps using the same algorithm in searching neighbor nodes around the BMU above.

4.3. Experimental Setup

Data used in this experiment were taken from Kaggle data provider. The amount of data and features varies. Further information from the dataset are shown in Table 1.

Table 1. Dataset Experiment

Dataset	Data Rows	Attributes
Credit Card Fraud Detection	250,000	16
Individual Tax Income Statistics	160,000	32
30 Years of European Solar Generation	200,000	64

SOM parameters and experiments evaluation carried out are shown in Table 2. SOM parameters testing are using topographic maps sizes of 100x100, 200x200 and 300x300 with the number of iterations as many as 100. The number of inputs on each test consisting of 3, 4 and 5 input vectors. Measurement of this experimental performance are calculated by capturing total computational time using Dynamic Parallel (DP) and Hyper-Q (HQ) and without using DP and HQ. From the total computation time then the calculated speed up is obtained.

Table 2. SOM Experiment Parameters

Map Size	Input Vector	Iteration	Learn Rate	Measurement

100x100	3, 4, 5	100	0.4	Total time and Speed up
200x200				
300x300				

The experiments were performed on a PC equipped with an Intel i3-1460 processor, 8 GB of RAM. GPU used in this experiment using Kepler Nvidia GT 720, 2 GB of RAM.

5. Result and Discussion

The experiments were performed on three datasets divided into two parts, the first part was an experiment without using Dynamic Parallel (DP) and Hyper-Q (HQ) and the second part was an experiment using DP and HQ. Experiments without the use of DP and HQ are conducted by calculating the average computation time of the three kernels that consist of the distance calculation kernel, the addition kernel and the comparative kernel, and a weight update kernel. The total amount of computational time in finding BMU and updating weight is the overall computational time of the experiment without the use of DP and HQ. Experiment using DP and HQ is using two kernels. The first kernel finds the BMU and the second kernel update weights. The overall computational time of this experiment is the sum of the computation time for the two kernels. This experiment runs as many as 100 iterations with an initial learning rate of 0.4.

The first experiment on the Credit Card Fraud Detection dataset uses the SOM experimental parameters as in Table 2. The results of recording computational time of the three kernels in the experiment without using DP and HQ with 100x100 map dimensions using input vectors 3, 4 and 5 have slightly different time results, similar to the 200x200 topographic map using input vectors 3, 4 and 5. While 300x300 topographic map with input vectors 3, 4 and 5 also get a time result that is not much different. This means that the three input vectors do not significantly affect the computation time. The weight update kernel gets relatively low computation time on each size of the topographic map using the three input vectors. This shows the computation time of the low-load kernel computing reduces the computational time of the three previous kernels above. Experiments using DP and HQ on topographic maps with dimensions of 100x100 using input vectors 3, 4 and 5 recorded lower time than without using DP and HQ. The same thing happens on topographic maps with dimensions of 200x200 and 300x300 using three input vectors. It means that by using DP and HQ, computing time is slightly better than without using DP and HQ. In the update weights kernel using DP and HQ get lower computational times than without using DP and HQ. The visible improvement of the experiments on the Credit Card Fraud Detection dataset achieves the first improvement assumption.

The second experiment on the Individual Income Tax Statistics dataset also uses SOM experimental parameters presented in Table 2. The results of recording computational time of the three kernels in the experiment without using DP and HQ with 100x100 map dimensions using input vectors 3, 4 and 5, as well as on the 200x200 dimension map and 300x300 dimensioned map with three input vectors are not obtained much time different compared to what was done in the first experiment. This also happens to the kernel update weights that get a computational time lower than the total time of the three kernels above. In the second experiment without using the DP and HQ it gets the same pattern as the first experiment. The second experiment using DP and HQ on topographic maps of 100x100, 200x200, 300x300 dimension using input vectors 3, 4 and 5 recorded better computation time from experimental results without using DP and HQ. In the update weights kernel using DP and HQ are also getting lower computational times compared to those without using DP and HQ. The visible improvement of the experiments on the Individual Income Tax Statistics dataset achieves a second improvement assumption.

The third experiment on the 30 Years of European Solar Generation dataset also uses SOM experimental parameters in Table 2. The results of recording computational time of the three kernels in the experiment without using DP and HQ with 100x100 map dimensions using input vectors 3, 4 and 5, as well as on the 200x200 dimension map and 300x300 dimensioned map with three input vectors not obtained much different computation time as was the case in the second experiment. Likewise in the update weights kernel, calculated computation time is lower than the total time of the three experimental kernels without using DP and HQ. The third experiment using DP and HQ on topographic maps of 100x100, 200x200, 300x300 dimension using input vectors 3, 4 and 5 recorded better computation time results compared to those experimental results without using DP and HQ. In the contrary, in the updating weights kernel using DP and HQ, a lower computation time is also calculated compared to those without using DP and HQ. This third experiment is also seen the same increases as the first and second experiments before.

The three experimental results obtained are shown in Figure 3. Bar graph shows experiments without using DP and HQ and the line graph shows experiments using DP and HQ. The trend of increase in computational time without and using DP and HQ tends to go up linearly in terms of map size and number of vector inputs. The larger the amount of data, the more computation time is required. In the case of topographic map of 100 x 100, the computational time difference based on the number of vector inputs did not change significantly. While on the topographic map measuring 200x200 and 300x300 the number of vector inputs indicates the influence.

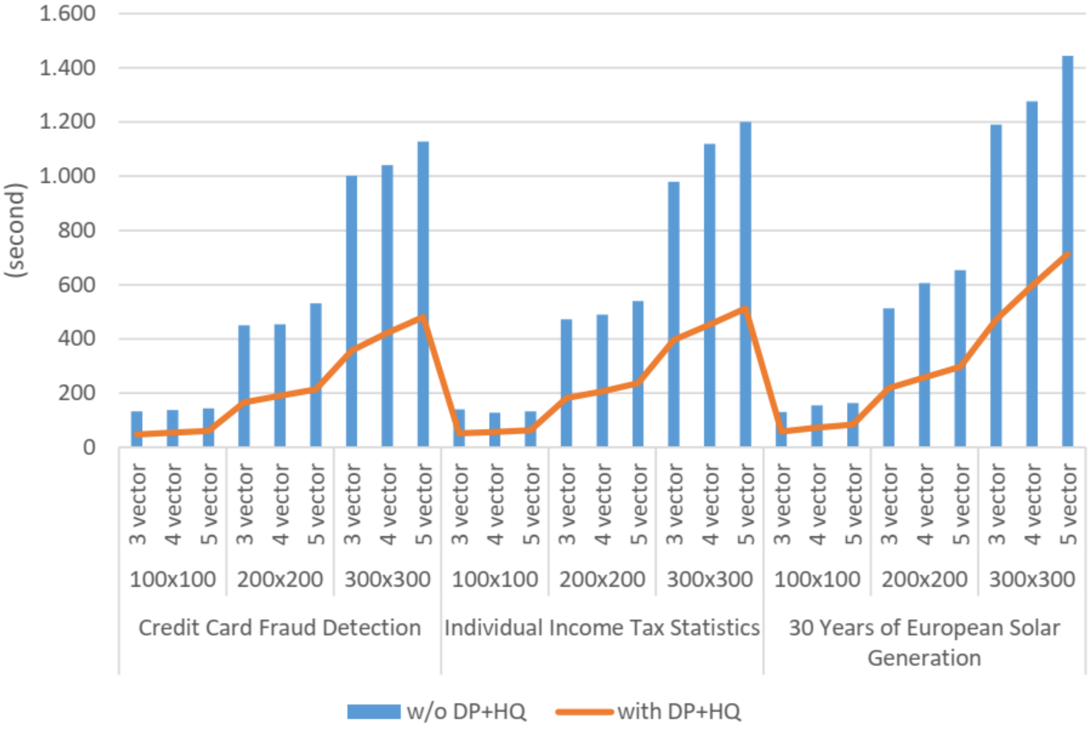


Figure 3. Overall Computation Time and Speed Up

The result of the parallel SOM computing time of the three experiments using different datasets as shown in Figure 3 shows that the overall computing time of the experiment using and without using Dynamic Parallel (DP) and Hyper-Q (HQ) increases linearly based on the map dimension and the number of input vectors used. On a 100x100 topographic map the effect of the input vector does not create a significant time difference. The same also happens on topographic map dimension of 200x200 where the input vector has little influence on the overall computation time. While on topographic map of dimension of 300x300 the amount of input vectors are affecting the whole computation time.

The line graph shown in the Figure 3 is the overall computational time of SOM parallel using DP and HQ, while the bar graph shows the overall computational time of SOM parallel without using DP and HQ. The line graph trajectory shows the amount of time that a parallel SOM generates using DP and HQ. The intersections of the line graph and bar graph show the center of the bar graph. This means that the parallel SOM using DP and HQ gets half the computation time faster than the parallel SOM without using DP and HQ.

6. Conclusion and Future Work

The proposed parallel SOM architecture using Dynamic Parallel and Hyper-Q in this paper was conducted by running the kernel to find the BMU that wraps the three child kernels into a parent kernel. The combination of Dynamic Parallel and Hyper-Q on finding the BMU and weights update shows a good performance improvement. The results of this experiment add some contributions to the improvement of parallel SOM performance, whereby experiments with the combination of Dynamic Parallel and Hyper-Q could lower overall computational time on large dimensional datasets.

Further work of this research could be directed to combining other parallel patterns such as stencil, map, shuffle with new architecture and big data problems.

REFERENCES

[1] T. Kohonen. Essentials of the self-organizing map. *Neural Networks*, volume 37, pp. 52-65, Jan. 2013.

[2] D. Miljkovic. Brief review of self-organizing maps. 2017 *40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2017.

[3] F. C. Moraes, S. C. Botelho, N. D. Filho, and J. F. O. Gaya. Parallel High Dimensional Self Organizing Maps Using CUDA. *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*, Oct. 2012.

[4] D. Mukunoki, T. Imamura, and D. Takahashi. Automatic Thread-Block Size Adjustment for Memory-Bound BLAS Kernels on GPUs. *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sep. 2016.

[5] B. Wang, C. Zhang, F. Wang, and J. Feng. A Synchronization Mechanism between CUDA Blocks for GPU. *Proceedings of the 2017 2nd International Conference on Control, Automation and Artificial Intelligence (CAAI 2017)*, 2017.

[6] L. Jarząbek and P. Czarnul. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *The Journal of Supercomputing*, volume 73, no. 12, pp. 5378-5401, Jun. 2017.

[7] R. S. Luley and Q. Qiu. Effective Utilization of CUDA Hyper-Q for Improved Power and Performance Efficiency. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016.

[8] M. F. Mustapha, N. E. Abd Khalid, A. Ismail, and M. Manaf. Enhancing Parallel Self-organizing Map on Heterogeneous System Architecture. *Soft Computing in Data Science*, pp. 162-174, 2017.

[9] N. E. B. A. Khalid, M. F. B. Mustapha, A. B. Ismail, and M. B. Manaf. Parallel Self-organizing Map Using Shared Virtual Memory Buffers. *Studies in Computational Intelligence*, pp. 49-58, 2017.

[10] P. Gajdoš and J. Platoš. GPU Based Parallelism for Self-Organizing Map. *Proceedings of the Third International Conference on Intelligent Human Computer Interaction (IHCI 2011)*, Prague, Czech Republic, August, 2011, pp. 231-242, Jul. 2012.

[11] T. Richardson and E. Winer. Extending parallelization of the self-organizing map by combining data and network partitioned methods. *Advances in Engineering Software*, vol. 88, pp. 1–7, Oct. 2015.

[12] H. Daneshpajouh, P. Delisle, J.-C. Boisson, M. Krajecki, and N. Zakaria. Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA. *High Performance Computing*, pp. 87-100, Dec. 2017.

[13] M. Harris. Optimizing parallel reduction in CUDA. *Proceedings of ACM SIGMOD*, 21, 104-110, 2007.

[14] T. Carneiro Pessoa, J. Gmys, F. H. de Carvalho Júnior, N. Melab, and D. Tuyttens. GPU-accelerated backtracking using CUDA Dynamic Parallelism. *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4374, Nov. 2017.

[15] NVIDIA. Nvidia CUDA C Programming Guide. *(Online) http://docs.nvidia.com/cuda/-cuda-c-programming-guide/index.html* (accessed in June 2018)

[16] I. Faraji and A. Afsahi. Hyper-Q aware intranode MPI collectives on the GPU. *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware - ESPM '15*, 2015.

[17] Wu-chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010.