Parallel Batch Self-Organizing Map on Graphics Processing Unit Using CUDA

Habib Daneshpajouh[1,2(✉)], Pierre Delisle[1], Jean-Charles Boisson[1], Michael Krajecki[1], and Nordin Zakaria[2]

[1] Centre de Recherche en STIC (CReSTIC), Université de Reims Champagne-Ardenne, Reims, France
daneshpajouh.habib@gmail.com, {pierre.delisle, jean-charles.boisson, michael.krajecki}@univ-reims.fr
[2] High Performance Cloud Computing Center (HPC3), Universiti Teknologi PETRONAS, Seri Iskandar, Malaysia
nordinzakaria@utp.edu.my

Abstract. Batch Self-Organizing Map (Batch-SOM) is being successfully used for clustering and visualization of high-dimensional datasets in a wide variety of domains. Although the structure of its training algorithm has a high potential for parallelization, focus of the previous efforts has been on the original Step-wise SOM. This gap is due to the facts that Batch-SOM requires some extra precautions (specially in its initialization phase), and it took quite a while since its introduction that researchers affirmed the desirability of using it in practice over the Step-wise SOM. Hence, the purpose of this paper is to propose a GPU parallelization model and implementation for the Batch-SOM using CUDA. The most computationally expensive parts of its training algorithm (such as steps to compute distance between each data vector and neuron, and determining the Best Matching Unit based on minimum distance) are identified and mapped on GPU to be processed in parallel. The proposed implementation shown significant speedups of 11× and 5× compared to the sequential and parallel CPU implementations respectively.

Keywords: Self-Organizing Map · CUDA · Clustering · Parallel SOM GPGPU

1 Introduction

Self-Organizing Map (SOM) proposed by Kohonen [1] is an unsupervised neural network that provides a low-dimensional (i.e. one or two dimensional) representation of multidimensional data vectors. SOM uses a data compression technique called Vector Quantization (VQ) to perform dimensionality reduction. VQ compression works by finding local averages of the dataset (represented by centroids in K-Means algorithm and neuron weights in SOM). In contrast to classical neural networks which require an input vector together with an associated target vector to be provided beforehand, the key feature of SOM is the ability to find internal structure of data without any supervision. In a nutshell, SOM works by associating each of the input vectors to one of its neurons in an iterative process in such a way that the overall

E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 87–100, 2018.
https://doi.org/10.1007/978-3-319-73353-1_6

distance between the neurons and their associated input vectors is minimized. The main goal of this process is to maintain the most important topological and/or metric relationships within the dataset in resulted low-dimensional network.

Since its introduction, SOM is being frequently used for clustering, visualization and data exploration problems in different domains such as industry, finance, natural sciences, biomedical analysis and linguistics [2]. SOM has a large application potential in engineering domain as well such as visualization of machine states, fault identification, process analysis and monitoring, and adaptive detection of quantized signals. By applying SOM to clustering problems, it will not only approximate the density function of the input samples (as most of classical clustering algorithms like K-Means do using VQ), but also provides a low-dimensional nonlinear projection of the high-dimensional datasets by topological arrangement of its neurons.

However, with the size of today's real-world datasets increasing sharply and complexity of data mining algorithms like SOM, quality of the result is not the only factor to measure the success of an algorithm, but its computational performance matters a lot too. Fortunately, just like other members of neural networks family, by having multiple computing nodes called neurons, SOM has a high potential of being parallelized. Moreover, a variation of SOM with modified training algorithm called Batch-SOM (in contrast to the original SOM with sequential step-wise training algorithm) makes it even more suitable for parallelization. According to Kohonen [3], by taking care of certain preliminaries, the result quality of the Batch-SOM is equal to (or even better than) the original SOM for majority of datatypes.

Several works have been done in the past for parallelization of SOM using different platforms on both Central Processing Unit (CPU) and Graphics Processing Unit (GPU). However, majority of the previous works emphasized on the original step-wise SOM and there is a lack of effort for parallelizing the Batch-SOM on GPU. Hence, the aim of this paper is to provide a parallelization model for the Batch-SOM using NVIDIA Compute Unified Device Architecture (CUDA) platform. The proposed GPU implementation shown significant speedups compared with the famous SOMToolbox [4] (a reference CPU implementation for the Batch-SOM provided by the Kohonen's team), and also the authors' own sequential and parallel CPU implementations.

The remaining of this paper is structured as follows. Section 2 presents the original SOM algorithm and Batch-SOM. Section 3 surveys the previous works on parallelization of SOM on GPU. The proposed GPU parallelization model of Batch-SOM is explained in Sect. 4. The performance and comparison results of the proposed model are presented in Sect. 5. Finally, Sect. 6 concludes the paper and proposes some future works.

2 SOM Algorithm

We follow Kohonen [3] to explain the two variations of SOM algorithm, and will be using the following notations for this section and also the rest of this paper:

- $x(t)$: a real n-dimensional Euclidean input vector, where integer t signifies a step in the sequence.
- X: sequence of all input vectors $\{x(t)\}$.
- m_i: a model (neuron), where i is its spatial index in SOM lattice.

- M_i: a variable sequence of all models (neurons) $\{m_i(t)\}$.
- m_c: a model (neuron) with closest distance to the input data vector passed to SOM lattice, and is located in the center of its neighbourhood. It is also called Best-Matching Unit (BMU) in SOM terminology.
- D_i: a variable set of all distances between each data vector x(t) and model (neuron) m_i.

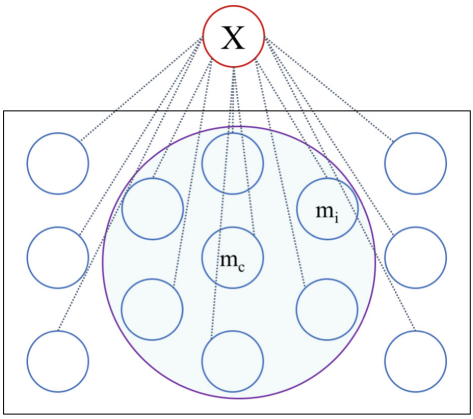

Fig. 1. Illustration of a Step-wise SOM.

Some parameters of SOM like number of neurons in each dimension of the lattice, maximum number of epochs (iterations) and learning rate (in case of Step-wise SOM) should be found using some statistical methods and heuristics that work on the basis of input data.

Learning algorithm of the original Step-wise SOM (shown in Fig. 1) known as "competitive learning" is described in the following:

1. Weights of the neurons are initialized. Initialization can be done either randomly or linearly using Principal Component Analysis (PCA) technique.
2. An input vector *x(t)* is chosen at random from *X* and broadcasted to lattice.
3. The *x(t)* is compared to each and every $m_i(t)$ to find the winner (BMU) in such a way that the neuron with index *c* is in the smallest distance from *x(t)* compared to all other neurons as in the following formula:

$$c = \text{argmin}_i\{\|x(t) - m_i(t)\|\} \tag{1}$$

4. The neighbourhood radius of the BMU $h_{ci}(t)$ is now calculated. The neighbourhood function has a vital role in SOM and its smooth convergence by producing large values at initial stages (typically covers the whole lattice) and decreasing overtime. This function can be defined as follows:

$$h_{ci}(t) = \alpha(t)\, e[-[x(t) - m_c]^2/2\alpha^2(t)] \tag{2}$$

where $\alpha(t) < 1$ (also called "learning rate") and $\sigma(t)$ are monotonically (e.g. exponentially) decreasing scalar functions of t, and $[x(t) - m_c]^2$ is the square distance between the neuron m_c and vector $x(t)$.

5. The weights of the BMU and each of its neighbouring neurons (found in step 4) are adjusted to be closer to the input vector. The closer a neuron is to BMU, the more its weights get altered. The formula in below calculates the new weights for each affected neuron:

$$m_i(t+1) = m_i(t) + h_{ci}(t)\,[x(t) - m_i(t)] \tag{3}$$

where $m_i(t)$ and $m_i(t+1)$ are the current and new weights of the neuron m_i respectively.

6. Steps 2 to 5 are repeated until the maximum number of epochs (i.e. iterations) is reached.

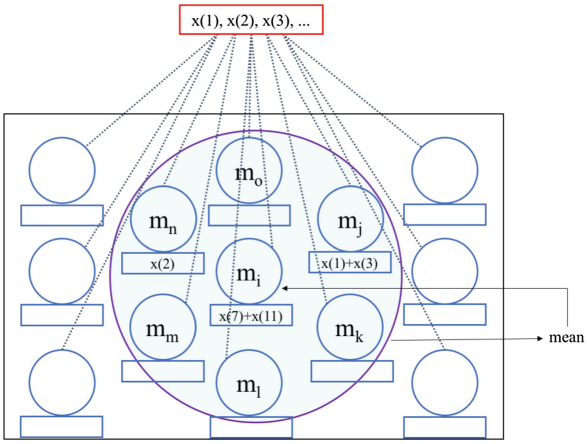

Fig. 2. Illustration of a Batch-SOM.

On the other hand, as Fig. 2 shows, the Batch-SOM works as in the following steps:

1. Weights of the neurons are initialized. Initialization can be done either randomly or linearly using PCA technique.
2. All the data vectors are passed to the lattice at once, with each neuron having its own sub-list. Each data vector is compared with each and every neuron to find its BMU. The index of each data vector goes to the sub-list of its BMU.
3. Finally, the new weights of each neuron are calculated as the weighted mean (the term "generalized median" is used by Kohonen to cover any type of data including the non-numeric types) of all the sub-lists in its neighbourhood. The mean's weight is determined by the distance of each neuron to the neuron that is getting the update.
4. Steps 2 to 3 are repeated until the maximum number of epochs is reached.

3 Related Work

Although there is a wide range of parallelization efforts for SOM, the focus of this report is only on those implemented on GPUs. The works on GPU implementations of SOM can be divided into two groups: first group includes those works that used graphics programming techniques and (vertex and fragment) shaders, while the second group belongs to those works that used CUDA and Open Computing Language (OpenCL) platforms to perform general computing on GPU. These two platforms completely revolutionized the way in which the programs are implemented on GPUs.

One of the early works on GPU parallelization of SOM is done by Zhongwen et al. [5] in which a GPU implementation of SOM is provided by converting the computational data into texture format and using the OpenGL texture buffers to perform the computation and data storage directly on GPU. The test environment for the proposed implementation contained Intel Pentium-4 2.4 GHz for CPU computation, and ATI 9550 and NVIDIA 5700 for GPU computation. By using 80 data vectors for training, the authors claimed speedups of up to 2× and 4× on the NVIDIA and ATI cards respectively.

Another work proposed at the same year as the above work is by Campbell et al. [6] in which the authors proposed a variation of SOM called Parameter-less SOM (PLSOM) which does not require to manually choose the "learning rate" parameter. They used shader functions to port SOM training computations on GPU. By trying different number of nodes for SOM lattice and a dataset including 1000 randomly generated and uniformly distributed 2-dimensional vectors, the authors claimed a speedup of up to 9× using their GPU implementation run on a GeForce 6800 card compared to a sequential CPU implementation run on a machine with 6 Intel Itanium-2 processors.

Xiao et al. [7] proposed a parallel GPU implementation for Batch-SOM using shader functions with the lattice of neurons being organized in 2D texture format. A vertex shader is responsible for finding the nearest neuron as well as the corresponding texel position for each training vector. Then, two fragment shaders are used to update the training vector and its neighbours. The authors evaluated the performance of their algorithm on an Intel Core2 Q8200 CPU and GeForce GTX 280 GPU. They claimed to achieve 15× to 40× speedup compared to a sequential CPU implementation by using different sizes of input data.

Despite the speedups that the aforementioned works achieved compared to CPU implementations, the high level of required knowledge (of graphics programming) and extra works that they were facing for converting the data structure and processing style of SOM to somewhat compatible with graphics APIs (like OpenGL) makes their approaches inconvenient in practice.

On the other hand, the major works after the introduction of CUDA are reviewed in the following. A GPU implementation of SOM is presented by Hasan et al. [8] using an open-source GPU Machine Learning Library called GPUMLib which is based on CUDA. In this work, weights initialization of neurons and subsequent update of the weights are done on host (CPU) while middle steps for finding the distance between neurons and input vectors followed by finding the BMU are done on device (GPU).

They performed an experiment on biomedical gene expression data using a NVIDIA Tesla C2075 graphics hardware and an Intel Xeon high-performance computer. The authors claimed that their GPU implementation achieved more than 3× speed up compared to the CPU implementation.

Davidson [9] proposed a parallel implementation of SOM using OpenCL on NVIDIA GPUs. In this work, Euclidean distance as the most common distance measure for SOM is replaced by Manhattan distance. Two kernels are used to find the BMU for each input vector in parallel. The performance of the OpenCL implementation is compared with a well-known SOM (CPU implementation) library called SOM_PAK [10] and shown a speedup factor of 10×.

Wang et al. [11] proposed a GPU implementation of SOM applying to travelling salesman problem using a concept called "parallel cellular matrix" that partitions the Euclidean plane of input data into a suitable number of uniform cell units. They used Nvidia GeForce GTX 570 Fermi graphics card containing 480 CUDA cores for their experiment which resulted average acceleration factors of 5.49, 12.68 and 39.74 (with respect to small, medium and large size of input data) compared to the CPU implementation.

Although the works reviewed above resulted significant speedups by their GPU implementations, one must note that the focus of these works has been mostly on the original SOM algorithm and there is a lack of effort on GPU parallelization of the Batch-SOM. While the original SOM algorithm was proposed in 1980s and the batch training algorithm a couple of years later, it took quite a while (by using it for different application areas in several research works) to confidently use the Batch-SOM as an alternative of the original SOM that might justify the lack of effort for GPU parallelization of this algorithm. As mentioned by Kohonen in [3], despite the need for taking care of some extra preliminaries (specially in the initialization phase) when working with the Batch-SOM, it is usually preferred over the original SOM in practice for several reasons:

- It does not have the time-variable learning rate parameter.
- It converges faster than the step-wise method.
- It can be generalized for the non-vectorial data too.

In addition, the Batch-SOM is more scalable to data size because of its high potential for parallelization. Therefore, the parallel-GPU model proposed in this paper focuses on the Batch-SOM by first, identifying the most computationally expensive stages of its learning algorithm, and then, using high-parallelizm potential of GPUs and also efficient processing techniques provided by CUDA platform to perform the computation of these stages.

4 Parallel Batch-SOM on CUDA

As explained by Kohonen [1], the most computationally expensive part of the Batch-SOM is the step to find the BMU for each data vector which consists of many distance comparisons. Our profiling analysis affirmed this too by showing that finding

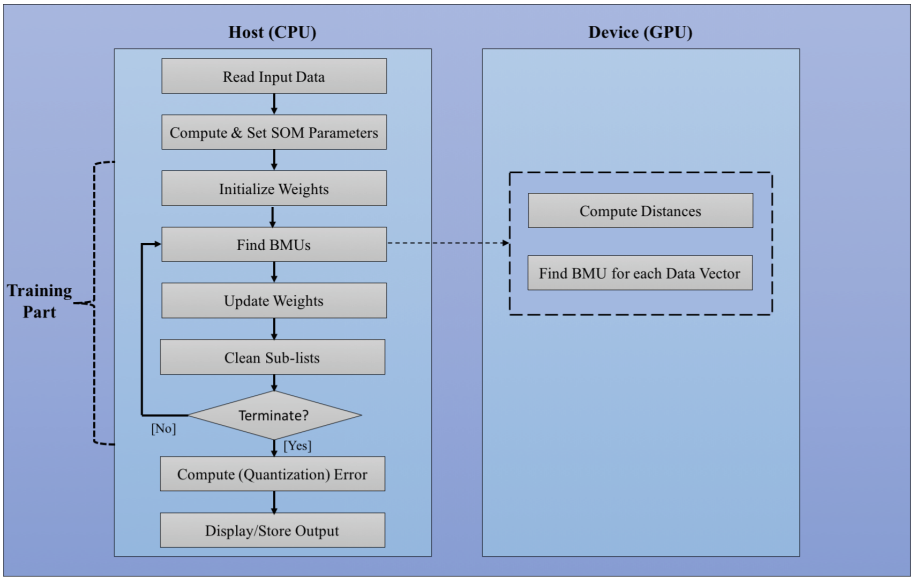

Fig. 3. Parallel Batch-SOM flowchart.

BMUs takes about 81 to 95.6% of the training's total processing time when it is run sequentially on CPU for 1000 and 7000 sizes of input data respectively. One may note that the step to update the weights takes a much smaller portion of the total processing time due to the fact that the radius of neighbourhood reduces overtime, requiring less and less computation to update the neurons. Hence, in our proposed parallelization model, computation of the step to find BMUs is ported to GPU.

Figure 3 depicts flowchart of our proposed parallel Batch-SOM. It starts with reading input data on the host side. Then, some SOM parameters such as data size, a single vector size, lattice radius etc. are computed and set. The next step (which plays a vital role in the result's quality of the Batch-SOM) is to initialize the lattice by setting the weights of its neurons to some initial values. The choice of initialization method (either randomly or linearly) depends on the data and problem types. Describing the subtleties of this choice is beyond the scope of this paper, but some useful suggestions are provided by Principe and Miikkulainen [12].

Following the initialization phase, the training loop begins by going through the procedure to find BMUs for all the input data vectors. This procedure consists of two major parts, both ported to GPU using two kernels to be processed. In its first part, distance between each data vector and neuron is computed. Then, in its second part, the BMU of each data vector is determined by finding the neuron in closest distance. The next step of the training algorithm is to update the weights of neurons based on the calculations of the previous step. The update step aims to transform the SOM lattice in such a way that it would be closer to distribution of the input dataset. At the end of the

training loop, all the sub-lists associated with neurons are cleaned and termination condition (which is usually a fixed value for maximum number of iterations) is checked. If the condition is met, Quantization Error of the resulted clustering is computed and the output is displayed/stored; otherwise, the loop continues to iterate by going back to the step of finding BMUs.

As mentioned above, computational burden of the procedure to find BMUs for each data vector is carried out by GPU. Algorithm 1 provides the pseudocode of this procedure. It begins by defining an array of CUDA streams [13] in order to be able to perform the data transfer from host to device and kernel execution in an asynchronous manner. The idea of CUDA streams comes from the fact that kernel execution in CUDA is an asynchronous action and once the host launches the kernel, it can proceed with next instruction in the program. We follow a 2-Way concurrency model (Fig. 4) for using streams in which the weights data is split into chunks to be transferred to GPU. Following the transfer of each chunk of data, our first kernel *computeDistances* (Algorithm 2) is called to operate on the respective chunk. This kernel is responsible for taking a single *n*-dimensional data vector and neuron, computing the Euclidean distance between them and storing the result in a global distance array. The global distance array is one-dimensional and stores distance values in a linear hierarchical structure consisting of streams, blocks and threads. After processing all the streams, the second kernel *minReduction* is launched to find the minimum distance for each data vector using the famous "parallel minimum reduction" algorithm. The *minReduction* kernel is lunched with the number of blocks equal to number of input vectors (so each block is responsible for finding the BMU for one input vector), and number of threads per block equal to half of total number of neurons. This kernel works on the global distance array that was initially stored in the device memory. Following the execution of the second kernel, the distance data is transferred from device to host. Eventually, the *findBMUs* procedure ends with calling the function *fillInSubLists* which puts the index of each data vector in its BMU's sub-list.

```
1:  procedure findBMUs()
2:      Define an array of streams
3:      for i ← 1 to NUM_STREAMS do
4:          Create CUDA streams
5:      end for
6:      for i ← 1 to NUM_STREAMS do
7:          Calculate and set memory offset
8:          Asynchronously copy a chunk of weights from host to device memory
9:          Lunch the kernel computeDistances <<< numBlocks, numThreadsPerBlock, 0, stream[i] >>>(…)
10:     end for
11:     Lunch the kernel minReduction <<< numBlocksRed, numThreadsPerBlockRed >>>(…)
12:     Copy the distance data from device to host
13:     Call the function fillInSubLists(distances[]) to put the index of each data vector into its BMU's sub-list
14: end procedure
```

Algorithm 1. Pseudocode of the *findBMUs* procedure

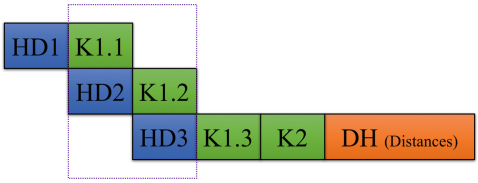

Fig. 4. 2-Way concurrency pipeline of CUDA streams used in our parallel implementation

```
1:  kernel computeDistances(...)
2:      Calculate and set vectorStartIndex
3:      Calculate and set vectorEndIndex
4:      Calculate and set weightStartIndex
5:      Calculate and set weightEndIndex
6:      Load the weights data into shared memory
7:      Synchronize the threads
8:      dist ← 0
9:      for vIdx ← vectorStartIndex to vectorEndIndex , wIdx ← weightStartIndex to weightEndIndex do
10:         dist ← dist + euclidean distance between the vector's and weight's element
11:     end for
12:     Calculate and set distanceIndex in the global distance array
13:     distArray[distanceIndex] ← dist
14: end kernel
```

Algorithm 2. Pseudocode of the *computeDistances* kernel

To further describe the details of computations on GPU, Fig. 5 illustrates data transfers and execution of the proposed parallel implementation. There are three major sets of data in this program, each stored in a one-dimensional array. First (before the start of training loop), the set of all input data vectors X is transferred to device memory. Then, the set of weights of all neurons M_i is divided into chunks (three chunks in the case of Fig. 5) and transferred to the device memory using streams. Determining the number of streams depends on the factors such as the GPU architecture (number of connections between host and device, available registers etc.) and also data size. Although the recent GPU architectures provide the ability to have a high number of streams, the balance between data size and overhead of creating too many streams must be taken into account in order to achieve an efficient concurrency.

After transferring each chunk of the weights data, the host (CPU) launches an instance of the *computeDistances* kernel on multiple blocks of threads. Since in our implementation each thread is responsible for computing the distance between a single data vector and neuron, the total number of threads required is the total number of data vectors times the total number of neurons in the lattice. It is noteworthy that all the streams launch an almost equal number of threads and blocks. The maximum number of threads allowed in each block and the amount of available per-block resources such as shared memory determine the number of blocks and number of threads per block in each stream. The part of the weights data assigned to each block is loaded into shared memory. This is because the number of neurons is usually much smaller than the number of data vectors. So, each neuron weights are accessed many more times than

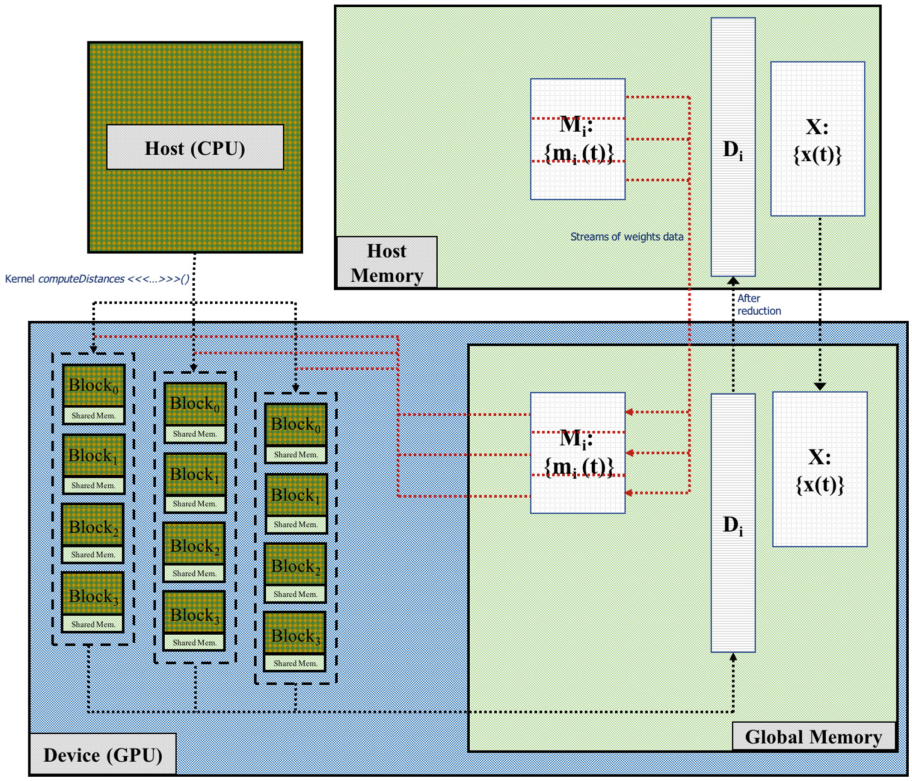

Fig. 5. Data transfers and execution of the proposed parallel implementation on GPU

the data vectors. Finally, each thread writes the result of its distance computation into the distance array that will be transferred to host memory after the reduction kernel finishes its computation.

5 Comparison and Results

In order to test the performance of the proposed parallelization model, we compared our implementation with three different implementations:

- **SOMToolbox:** A CPU-based single-thread reference implementation of the Batch-SOM, developed in MATLAB by the Kohonen's team. This package provides the ability to train a SOM network with different parameters and compute various errors, quality and measures for the SOM. It also provides visualization of the result, and correlation and cluster analysis.
- Our own sequential **CPU** implementation: This is developed in *C* by following the SOMToolbox and acts as the groundwork of our parallel implementations.

- Our own **OpenMP** implementation: This is developed in *C* in order to have an alternative parallel implementation to be compared with our CUDA implementation. Exactly like the proposed Batch-SOM on CUDA, in the OpenMP implementation, computation of the procedure *findBMUs* is done in parallel using different number of processing cores ranging from 2 to 16.

Table 1. The specifications of the machine used in our experiments

CPU (Bi-processors)	Memory	GPU	OS
Intel Xeon E5-2650V2 16 Cores (in total) 2.60 GHz	64 GB	NVIDIA Tesla K20x	Bullx Linux Server Release 6.1

All the implementations work on double precision and the necessary optimization flags are used in compilation of CPU implementations. The specifications of the machine used in our experiments are presented in Table 1. We used the dataset which was the result of Non-dominated Sorting Genetic Algorithm II (NSGA-II) runs applied to an instance of Vehicle Routing Problem with Time Windows (VRPTW) having 100 customer points with 200-dimensional integer data vectors. Each data vector occupies 800 bytes of memory. These data vectors are essentially GA chromosomes, each of which is a solution to this instance of VRPTW.

With respect to fine tuning of GA parameters for this instance of VRPTW, the population size was set to 100 with 70 as number of generations. Hence, the GA produced 7000 solutions (vectors) which we used as input data for SOM clustering. However, to see the behaviour of our parallel SOM against different sizes of input data, we extracted random subsets of 1000, 2000 and 4000 vectors from this data. The dataset being used can be retrieved from [14]. By using a heuristic provided in SOMToolbox, the maximum number of SOM iterations was set to 12 with a 15×7 lattice of neurons. Each implementation was run 50 times on different sizes of datasets.

Table 2 presents runtime comparison of the training part (as indicated in Fig. 3) of the CUDA, the OpenMP (on 2, 4, 8 and 16 cores), the sequential CPU and the SOMToolbox implementations on different sizes of input data which clearly shows that the proposed CUDA implementation is by far performing better compared to the other implementations. The speedup is more evident when the data size is increased. At its best performance when the number of input vectors is 7000, the proposed CUDA implementation provides a speedup of $5\times$, $11\times$ and $11\times$ compared to the OpenMP, the CPU and the SOMToolbox implementations respectively. Profiling information for the case of 7000 input vectors is presented by Table 3.

It is noteworthy that the slight advantage of our CPU implementation over the SOM Toolbox is because of two reasons. First, SOM Toolbox is implemented in Matlab which is typically slower than C programs. Second, unlike our implementation, SOM Toolbox includes some extra error checking and exception handling commands that is usual in all the public packages. Otherwise, our CPU implementation is the same as the one in SOM Toolbox.

Table 2. Runtime (of the training part) comparison between CUDA, OpenMP, sequential CPU and SOMToolbox implementations (time in milliseconds)

No. of Input vectors	Runtime	CUDA	OpenMP				Seq. CPU	SOM Toolbox
			2Cores	4Cores	8Cores	16Cores		
1000	Best	136	538	487	428	322	569	597
	Worst	151	549	504	433	365	584	607
	Std. dev.	142	541	495	430	341	577	601
2000	Best	215	1043	940	802	632	1137	1163
	Worst	232	1055	947	809	669	1152	1192
	Std. dev.	226	1049	942	806	651	1142	1181
4000	Best	287	1813	1571	1283	934	2012	2025
	Worst	297	1825	1593	1291	973	2029	2051
	Std. dev.	292	1817	1579	1287	959	2022	2036
7000	Best	371	3796	3288	2664	1952	4211	4262
	Worst	388	3803	3308	2685	1991	4256	4296
	Std. dev.	382	3797	3295	2672	1975	4230	4271

Table 3. Profiling information of the training part (as indicated in Fig. 3) for the case of 7000 input vectors

Section	Initialize weights ()	memcpy HtoD	Compute distances <<< >>> ()	Min reduction <<< >>> ()	memcpy DtoH	Update weights ()	Others
Time (ms)	37	9	132	24	21	141	7
Time (%)	9.97	2.43	35.58	6.47	5.66	38.01	1.88

Moreover, from the perspective of clustering quality, Table 4 provides the results of comparing our CUDA implementation with the SOMToolbox in terms of two well-known measures. The first measure is Average Quantization Error (AQE) which indicates the average of total distances of all data vectors to their respective cluster's centroids (i.e. their BMUs weights in case of SOM algorithm). The second measure is Average Silhouette Coefficient (ASC) which is one of the intrinsic methods for evaluating clustering quality by reflecting the average ratio of intra-cluster closeness to inter-cluster compactness. ASC has a range of $[-1, 1]$. The goal of SOM clustering is to minimize the AQE and maximize the ASC. In majority of cases, our CUDA implementation provided an equal quality to SOMToolbox while in some cases there is a slight difference (both higher and lower) between the two, which is normal because of random nature of the SOM algorithm.

Table 4. Clustering quality comparison between CUDA and SOMToolbox implementations (Average Quantization Error: lower is better – Average Silhouette Coefficient: higher is better)

No. of Input vectors	Quality	CUDA		SOMToolbox	
		Avg. quantization error	Avg. silhouette coefficient	Avg. quantization error	Avg. silhouette coefficient
1000	Best	60.8	0.9896	60.8	0.9896
	Worst	69.4	0.9854	68.6	0.9851
	Std. dev.	61.9	0.9875	62.1	0.9877
2000	Best	110.2	0.9892	110.2	0.9892
	Worst	113.7	0.9838	114.1	0.9838
	Std. dev.	110.3	0.9866	111.1	0.9869
4000	Best	195.1	0.9873	194.6	0.9873
	Worst	192.3	0.9783	191.4	0.9781
	Std. dev.	193.8	0.9823	193.5	0.9818
7000	Best	209.2	0.9625	209.2	0.9625
	Worst	211.6	0.9523	212.3	0.9566
	Std. dev.	210.7	0.9581	209.4	0.9598

6 Conclusion and Future Work

The Self-Organizing Map is a data mining algorithm being extensively used nowadays for clustering and visualization problems in different domains. SOM has a unique feature of not only providing an approximation of density function of the dataset, but also a nonlinear projection of the high-dimensional data vectors to a low-dimensional space. However, the complexity of its training algorithm and the size of today's real-world datasets makes it necessary to use some kinds of parallelization for its computation. GPUs proved to be one of the most powerful computing hardware nowadays. Hence, this paper proposed a parallelization model for the Batch-SOM, as it proved to have equal quality of the result compared to the original SOM, and it is more suitable for parallel computing. We compared our GPU implementation with other sequential and parallel CPU implementations and got significant speedups.

However, some developments can be done in future to enhance the model and implementation. From an algorithmic perspective, some useful suggestions are provided by Kohonen [3] such as multiplying the number of neurons to save computing time in constructing large SOMs, estimating the BMU location based on the previous searches, tabulating the indices of non-zero elements of sparse vectors to speed up the BMU search process and using a coarse numerical accuracy of the vectors to reduce the memory requirements of high-dimensional input data. On the other hand, the GPU architecture can benefit from future developments as well. Although the experimental case used in Sect. 5 conveniently fits a single GPU, this is not always the case. Considering the fact that the size of today's datasets (e.g. those from operations research, biology, medical image analysis etc.) might go beyond the capabilities of a

single GPU, extending the model and implementation to use multi-GPUs and also a super-computer-based implementation with multiple computing nodes, each with multiple GPUs might be useful.

Acknowledgments. This work is partially supported by Malaysia Fundamental Research Grant Scheme (FRGS) 1/2017/ICT01/UTP/02/2. The experiments reported in this work were performed on the ROMEO computational centre of Champagne-Ardenne, France (http://romeo.univreims.fr). The authors would like to thank J. Loiseau for his useful advices on the GPU implementation.

References

1. Kohonen, T.: Self-Organizing Maps. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-642-56927-2
2. Kohonen, T.: Essentials of the self-organizing map. Neural Netw. **37**, 52–65 (2013)
3. Kohonen, T.: MATLAB Implementations and Applications of the Self-Organizing Map, 201 p. Unigrafia Oy, Helsinki (2014)
4. Alhoniemi, E., Himberg, J., Parhankangas, J., Vesanto, J.: SOM Toolbox, http://www.cis.hut.fi/projects/somtoolbox. Accessed 15 Jan 2017
5. Zhongwen, L., Zhengping, Y., Xincai, W.: Self-organizing maps computing on graphic process unit. In: ESANN' 2005 - European Symposium on Artifitial Neural Networks, Bruges, Belgium, pp. 27–29 (2005)
6. Campbell, A., Berglund, E., Streit, A.: Graphics hardware implementation of the parameter-less self-organising map. In: Gallagher, M., Hogan, J.P., Maire, F. (eds.) IDEAL 2005. LNCS, vol. 3578, pp. 343–350. Springer, Heidelberg (2005). https://doi.org/10.1007/11508069_45
7. Xiao, Y., Leung, C.S., Ho, T.-Y., Lam, P.-M.: A GPU implementation for LBG and SOM training. Neural Comput. Appl. **20**, 1035–1042 (2011)
8. Hasan, S., Shamsuddin, S.M., Lopes, N.: Soft computing methods for big data problems. In: Cai, Y., See, S. (eds.) GPU Computing and Applications, pp. 235–247. Springer, Singapore (2015). https://doi.org/10.1007/978-981-287-134-3_15
9. Davidson, G.: A parallel implementation of the self organising map using OpenCL, (2015)
10. Kohonen, T.K., Hynninen, J., Kangas, J., Laaksonen, J.: SOM_PAK: The Self-Organizing Map Program Package, 27 p. (1996)
11. Wang, H., Mansouri, A., Creput, J.-C.: Massively parallel cellular matrix model for self-organizing map applications. In: 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 584–587. IEEE (2015)
12. Príncipe, J.C., José, C., Miikkulainen, R.: WSOM 2009. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-35230-0
13. NVIDIA: CUDA C Programming Guide (2017)
14. Daneshpajouh, H.: NSGA_II-VRP_C101 Dataset, https://drive.google.com/open?id=0ByIxLNivsQhyMVgtN0VFdTRsRms. Accessed 23 Apr 2017