Proceedings of Machine Learning Research 95:264-279, 2018 ACML 2018

A Scalable Heterogeneous Parallel SOM Based on MPI/CUDA

Yao Liu LIUYAO@CC.ECNU.EDU.CN
Jun Sun LZZR@LIVE.COM
Qing Yao YAOQINGJS@FOXMAIL.COM
Su Wang SWANG@CC.ECNU.EDU.CN
Kai Zheng KZHENG@CS.ECNU.EDU.CN
Yan Liu YLIU@CC.ECNU.EDU.CN
School of Computer Science and Software Engineering, East China Normal University, 3663 N. Zhongshan Rd., Shanghai, China

Editors: Jun Zhu and Ichiro Takeuchi

Abstract

Self-Organizing Map (SOM) is a kind of artificial neural network used in unsupervised machine learning, which is widely applied to clustering, dimension reduction and visualization for high-dimensional data, etc. There are two major versions of the training algorithm: original algorithm and batch algorithm. Compared with the original, the batch algorithm has some advantages including faster convergence and less computation, and is suitable for parallelization. However, it is still confronted with the challenge of efficiency in the case of massive data, high-dimensional data or a large-scale map. In this paper, a scalable heterogeneous parallel SOM based on the batch algorithm is proposed which combines process-level and thread-level parallelism by MPI and CUDA. To boost the parallel efficiency on GPUs and make full use of the high floating-point computing capability, we design matrix operations for the the most time-consuming steps, the computation of best match units and weights update, making the steps available for the implementation by cuBLAS. In addition, the memory optimization methods are adopted. The experiments show that the proposed heterogeneous parallel SOM is effective, efficient and scalable.

Keywords: self-organizing map; GPU; MPI; CUDA; heterogeneous parallel

1. Introduction

Self-Organizing Map (SOM) proposed in 1982 by Kohonen (2013) is a kind of artificial neural network, which is widely applied to clustering, dimension reduction and visualization for high-dimensional data, etc. Human brain neuron units respond selectively to some specific stimuli, whose topological locations correspond to some specific stimulus orderly in brain map. By simulating this biological characteristic, an artificial neural network named SOM was developed with principles of self-organizing and competitive learning. It was successfully applied to facial emotion recognition by Majumder et al. (2014), image classification by Liu et al. (2015) and bioinformation analysis by Shah and Luo (2017).

SOM training algorithm is heuristic which is similar to gradient descent optimization in the procedure of tuning the model (weights of nodes in the map) by multiple iterations. During the training, the competitive learning strategy is adopted to make the nodes self-

© 2018 Y. Liu, J. Sun, Q. Yao, S. Wang, K. Zheng & Y. Liu.

organized. However, only in some simple low-dimensional cases, converging to the global optimum by SOM in the predictable iterations has been proven mathematically. In the high-dimensional cases, the large number of iterations has to be pre-set to ensure the training quality. Moreover, the training process of every iteration is time-consuming in the case of massive data, high-dimensional data or a large-scale map. Therefore, it's necessary to parallelize SOM to shorten the training time.

Due to the high floating-point computing capability, the Graphics Processing Unit (GPU) is adopted as the parallelization platform, which was applied to numerous machine learning models such as conditional random field (Ai et al., 2017), gradient boosting (Mitchell and Frank, 2017), sequential minimal optimization for support vector machine (Wen et al., 2018).

There are some researches focused on parallelizing SOM. Wang et al. (2014) proposed a SOM hardware system for special purpose based on Field Programmable Gate Array (FPGA), in which some operations unsuited for FPGA were simplified. Although FPGA based SOM implementation achieves high speed, it is weak in flexibility and difficult to adapt various sizes of the map and topological shapes. For example, the Gaussian function frequently used in SOM is difficult to implement efficiently on FPGA. On GPU platform, Xiao et al. (2015) designed a parallel SOM algorithm using OpenGL rather than more popular Compute Unified Device Architecture (CUDA) where distances calculation and weight updating were implemented through compute shader, vertex shader and fragment shader. The experiments showed it is faster than the CUDA counterpart. Works listed above can accelerate SOM training to a certain extent, but they are deficient in scalability and they are still confronted with the challenge of efficiency in the case of massive-scale computing. Moreover, the works are based on the original, incremental algorithm, but the batch SOM algorithm is recommended by the SOM inventor — Kohonen (2013) due to less computation and faster convergence speed. Furthermore, besides the Euclidean distance, other distance metrics are available in the batch SOM algorithm.

Parallelization of batch SOM algorithm has gradually become a research hotspot. Sarazin et al. (2014) designed two scalable SOM-MapReduce algorithms on Spark platform using MapReduce paradigm, which scale the batch SOM algorithm up to 24 hosts. This work demonstrated the high scalability. However, it was programmed in Scala language which is not as efficient as C/C++, and it used CPU only. Somoclu (Wittek et al., 2017) is an open-source parallel implementation of batch SOM algorithm. It is able to rely on MPI for distributing the workload in a cluster, and it could be accelerated by GPU. It achieves state-of-the-art performance in open-source SOM software. However, the high float-point computing performance of GPU does not be fully utilized.

In this paper, a scalable Heterogeneous Parallel SOM (HPSOM) is proposed based on Message Passing Interface (MPI) and CUDA. It could be scaled up to multiple GPUs on multiple hosts using the heterogeneous parallelism of process-level and thread-level. Moreover, the matrix operation methods are designed and the memory optimization methods are adopted. In the experiments, the comparison with Matlab and Somoclu proves the effectiveness. The experiments also show HPSOM on 8 GPUs provides an acceleration of 4861 times over the serial execution on a CPU at best, which outperforms Somoclu and also remains satisfied GPU parallel efficiency.

2. Self-Organizing Map

Self-Organizing Map produces a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples. The visible part of a self-organizing map is the map space, which consists of components called nodes or neurons, and each of them is associated with a weight vector. A typical SOM structure is illustrated in Fig. 1, where circles represent nodes organized in hexagonal grids. Let $\boldsymbol{x} = [x_1, x_2, \ldots, x_{dn}]^{\mathrm{T}} \in \mathbb{R}^{dn}$ be an input instance and the weight vector of node i is denoted by $\boldsymbol{m}_i = [m_{i,1}, m_{i,2}, \ldots, m_{i,dn}]^{\mathrm{T}} \in \mathbb{R}^{dn}$, henceforth the model $\boldsymbol{M} = [\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_{nn}]$ is called codebook customarily.

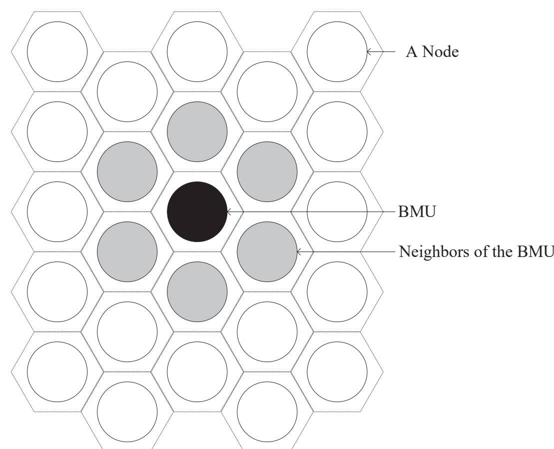


Figure 1: A SOM hexagonal grid with 25 (5 × 5) nodes

After training, input data are mapped onto the map space following the "matching by similarity" rule: more similar input data are associated with nodes that are closer in the grid, whereas less similar nodes are situated gradually farther away in the grid.

To train a map, the competitive learning strategy is adopted. There are two major versions of the training algorithm: original algorithm and batch algorithm. The original algorithm is an incremental procedure. In every step of an epoch, an input instance is selected randomly from dataset. According to the Euclidean distances in input space, the closest node to the input instance is found, which is known as the Best Match Unit (BMU). By the learning strategy, both the weight vector of BMU and weight vectors of all nodes in the BMU-centric neighborhood should be updated. In Fig. 1, the black node is the BMU, and the grey nodes are in its neighborhood where the radius of the neighborhood is 1. During an epoch, the above steps are repeated until every input instance has been selected. The training process of an epoch is repeated until the specified number of epochs reaches. There are two epoch-variable training parameters of: learning rate and radius of neighborhood, which descend every epoch. Multiple updates and large amount computation are needed in every epoch. It is not suitable for parallelization due to the incremental procedure.

Compared with the original SOM algorithm, the batch SOM algorithm has some advantages including faster convergence and less computation, and is suitable for parallelization because a large proportion of data dependency is eliminated and the update times of the

codebook is decreased. And the learning rate $\alpha(t)$ is no more needed. The procedure is as follows.

1. Initialize the codebook $\boldsymbol{M}$ randomly.
2. Find BMUs for every input instance by Eq. (1).

$$\|\boldsymbol{x} - \boldsymbol{m}_c\| = \min_i\{\|\boldsymbol{x} - \boldsymbol{m}_i\|\}, i < nn, \tag{1}$$

where c denote the BMU of instance $\boldsymbol{x}$.

3. Update the codebook using

$$\boldsymbol{m}_i = \frac{\sum_j n_j h_{j,i} \boldsymbol{x}_{m,j}}{\sum_j n_j h_{j,i}}, \tag{2}$$

where $\boldsymbol{x}_{m,j}$ is used to denote the inputs which share the same BMU j, and n_j is the number of these inputs. The notation $h_{c,i}$ denotes the Gaussian function:

$$h_{c,i} = h_0 \exp(-\|\boldsymbol{r}_i - \boldsymbol{r}_c\|^2/\sigma^2), \tag{3}$$

where $\boldsymbol{r}_i$ and $\boldsymbol{r}_c$ are the coordinates of nodes i and c respectively, h_0 is initial radius of the neighborhood.

The Gaussian function in Eq. (3) is the essential of self-organization. The input data affect not only BMU c, but also all nodes in N_c. The degree decreases as the distance between c and the neighbor node increases. A significant advantage of the batch SOM algorithm is that only once update of model every epoch is needed. There is less computation in every epoch in batch SOM algorithm than in the original SOM algorithm. The updating times is reduced to nn (the number of nodes) compared to $nn \times nx$ (the number of input instances) of the original. It has been proved that the batch has the same convergence (Cheng, 1997).

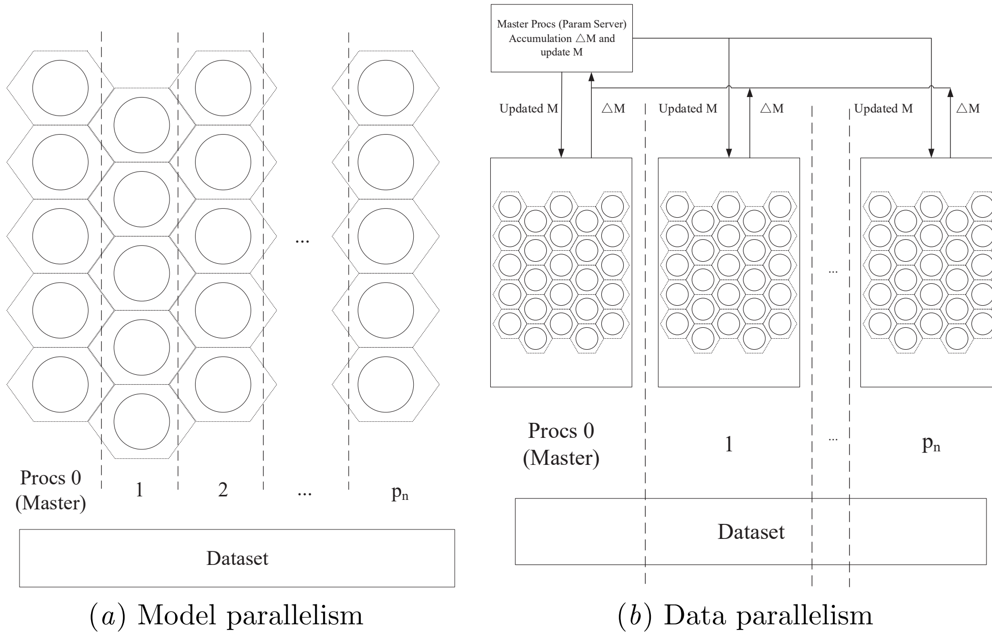
3. A Scalable Heterogeneous Parallel SOM

In this section, we present a scalable heterogeneous parallel SOM. Firstly two approaches of parallelism across hosts, model parallelism and data parallelism, are discussed. We analyze their advantages and disadvantages and choose the later as our technical route. And then we extend the data parallelism to the multi-GPU platform, design the framework of HPSOM and implement it. Meanwhile, we design matrix operations for the most time-consuming steps, the computation of best match units and weights update, making the steps available for the implementation by cuBLAS. In addition, we present several memory optimization methods.

3.1. Parallelization across Hosts

There are two main approaches to parallelize a neural network among processes: model parallelism and data parallelism. They are illustrated in Fig. 2.

In the model parallelism approach, the model $\boldsymbol{M}$ is split into several parts. Every process trains a different part of the model $\boldsymbol{M}$. In this way, a part of $\boldsymbol{M}$ is stored in a process. This approach has been used in deep neural networks such as AlexNet (Krizhevsky



(a) Model parallelism (b) Data parallelism

Figure 2: Two parallelism approaches

et al., 2012), a kind of convolutional neural network. For the batch SOM algorithm, node $\boldsymbol{m}$ is necessary to find the input instances which take $\boldsymbol{m}$ as BMU, so that every process needs the whole dataset $\boldsymbol{X}$ to calculate the distances using Eq. (1). And the slave processes reduce the distances to the master process. The master process finds the BMU of every input instance according to the received distances. After that, the master process broadcast the BMUs to all processes to update the assigned part of $\boldsymbol{M}$ using Eq. (2).

The model parallelism approach is suitable for very large model which is too large to be stored in the memory of single host. However, in SOM, the nodes (neurons) in the same neighborhood affect each other due to the competitive learning strategy, which may be assigned to different processes. That will result in extra communication. Furthermore, the scalability of model parallelism is limited, because the scale of model is limited usually.

An alternative approach is data parallelism, dividing the dataset $\boldsymbol{X}$ into several parts $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{pn}$ (pn denotes the number of processes), and the whole codebook is stored in every process. Every process trains the whole model with different part of the dataset. The slave processes send the training results to the master process (parameter sever) every epoch. For the batch SOM algorithm, every process calculates the partial numerators and denominators of $\boldsymbol{m}_i$ in Eq. (2), and then reduce them to the parameter sever. The parameter server updates the codebook. Then the updated model $\boldsymbol{M}$ is broadcasted to the slave processes. At the end of every epoch, all the processes are synchronized.

The data parallelism approach needs less communication and synchronization than the previous. And the scalability of data parallelism is higher than that of model parallelism, because it's determined by the size of dataset which are probably very large. In this paper, data parallelism is adopted and extended to multi-GPU platform.

3.2. Parallelization on GPUs across Hosts

Heterogeneous parallelization is a popular method to accelerate neural network training. Among the heterogeneous hardware devices commonly used, GPU designed to execute thousands of threads concurrently has high parallelism and throughput. To adopt data parallelism on GPUs, it is necessary to divide the dataset $\boldsymbol{X}_i$ into several parts $\boldsymbol{X}_{i,1}, \boldsymbol{X}_{i,2}, \ldots, \boldsymbol{X}_{i,gn}$ (gn donotes the number of GPUs). The framework of data parallelism of HPSOM is proposed and illustrated in Fig. 3.

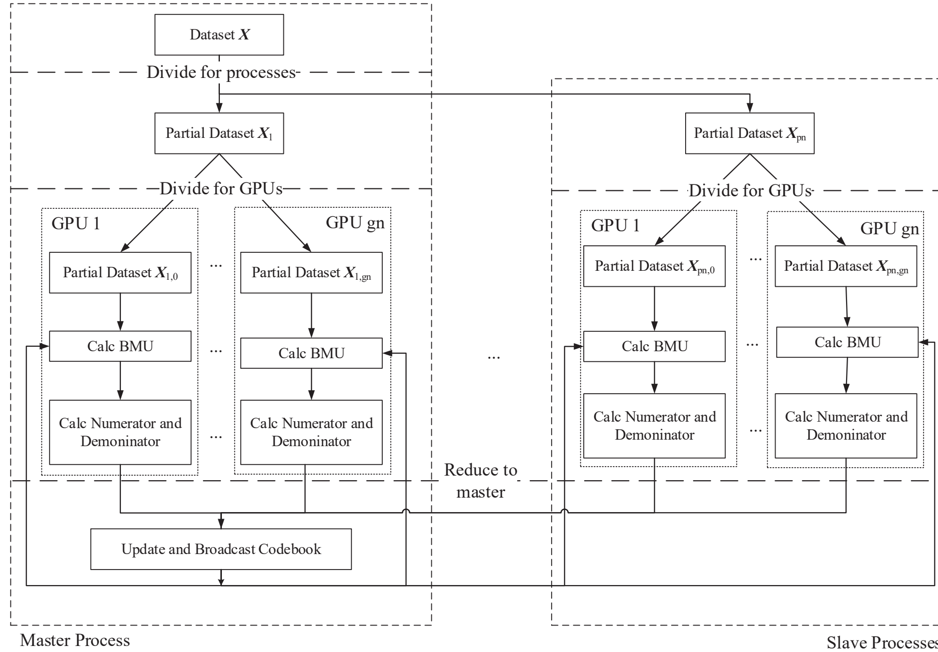


Figure 3: Framework of data parallelism of HPSOM

Usually, a hosts has more than one GPU. The GPUs could access the main memory of the host, and own independent video memories. Within MPI/CUDA heterogeneous programming model, multiple GPUs in a host could be manipulated by either one process (one-to-many mode) or multiple processes (one-to-one mode). The number of processes of one-to-one mode is more than that of one-to-many mode. The whole codebook $\boldsymbol{M}$ is needed by every process, so the one-to-one mode needs more main memory. Thus we choose the the one-to-many mode.

To implement this parallelization, inter-process communication and synchronizing across hosts are needed, which are provided by MPI. MPI is a message passing standard that facilitates the development of parallel applications. It is the most popular for distributed memory parallelism, and is suitable for coarse-grained (process-level) parallelism.

Based on the discussion, the flowchart of HPSOM is designed and shown in Fig. 4. The main steps are as follows.

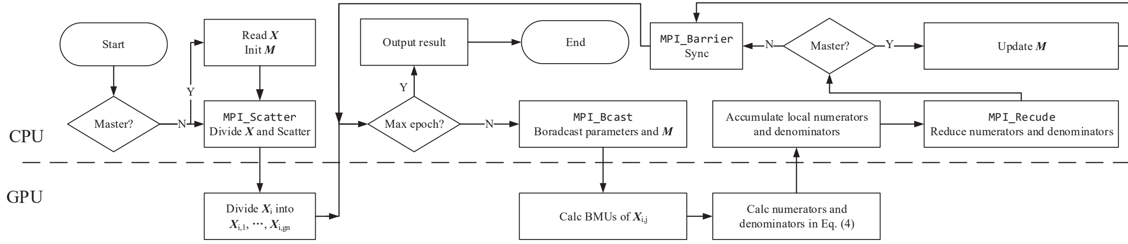


Figure 4: Flow of HPSOM

1. All processes get the total number of processes and the current process ID by `MPI_Comm_size`, `MPI_Comm_rank` respectively. Then the master process reads training parameters and input dataset $\boldsymbol{X}$, and initials the codebook $\boldsymbol{M}$.
2. Master process divides $\boldsymbol{X}$ equally into $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{pn}$, and they are scattered to every process by `MPI_Scatter`. After getting $\boldsymbol{X}_i$, the process i divides it equally to gn pieces, where gn denotes the number of GPUs in a host.
3. The parameters and codebook $\boldsymbol{M}$ are broadcasted to every process by `MPI_Bcast`.
4. BMUs are calculated on GPUs in parallel.
5. The numerators and denominators are calculated on GPUs in parallel.
6. They are copied to the main memory by `cudaMemcpy` and are accumulated respectively in corresponding processes.
7. The locally accumulated numerators and denominators in every process are reduced to the master process by `MPI_Reduce`.
8. The codebook $\boldsymbol{M}$ is calculated and updated in the master process. Then `MPI_Barrier` is called for synchronization.
9. Repeat step 3 to 8 until the number of epochs is reached.

3.3. Optimization Methods

Due to the hardware characteristics, a rough implementation on multiple GPUs will not necessarily improves the performance, or even gets a worse performance on certain workload. Although the GPU cores is much more than that of CPU, they are only efficient at the simultaneous processing of large blocks of data, such as matrix and vector operations. Moreover, the cuBLAS, a Basic Linear Algebra Subprograms (BLAS) library, has been included in the NVIDIA CUDA SDK, which is very high efficient, supporting vector addition, scalar multiplication, dot products, and matrix multiplication. Re-structuring some modules of SOM into the matrix operations can significantly improve the efficiency.

3.3.1. Matrix Operations for Calculating BMUs

Calculating distances is the most time-consuming during calculating BMUs. To optimize the calculation of BMUs, the calculation of distances is re-structured into matrix operations. Suppose there are xdn instances in $\boldsymbol{X}_{i,j}$, therefore $xdn \times nn$ distances are to be calculated by the corresponding GPU. Instead of the Euclidean distance, the squared Euclidean distance is adopted in this paper to simplify the calculation of distances. A matrix of distances denoted

by $\boldsymbol{D}$ whose elements represent the squared Euclidean distance could be decomposed into three matrices.

$$\begin{aligned}
\boldsymbol{D} &= \begin{bmatrix} \|\boldsymbol{x}_1-\boldsymbol{m}_1\|^2 & \|\boldsymbol{x}_1-\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{x}_1-\boldsymbol{m}_{nn}\|^2 \\ \|\boldsymbol{x}_2-\boldsymbol{m}_1\|^2 & \|\boldsymbol{x}_2-\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{x}_2-\boldsymbol{m}_{nn}\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|\boldsymbol{x}_{xdn}-\boldsymbol{m}_1\|^2 & \|\boldsymbol{x}_{xdn}-\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{x}_{xdn}-\boldsymbol{m}_{nn}\|^2 \end{bmatrix} \\
&= \begin{bmatrix} \|\boldsymbol{x}_1\|^2 & \|\boldsymbol{x}_1\|^2 & \dots & \|\boldsymbol{x}_1\|^2 \\ \|\boldsymbol{x}_2\|^2 & \|\boldsymbol{x}_2\|^2 & \dots & \|\boldsymbol{x}_2\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|\boldsymbol{x}_{xdn}\|^2 & \|\boldsymbol{x}_{xdn}\|^2 & \dots & \|\boldsymbol{x}_{xdn}\|^2 \end{bmatrix} - 2 \begin{bmatrix} \boldsymbol{x}_1^{\mathrm{T}}\boldsymbol{m}_1 & \boldsymbol{x}_1^{\mathrm{T}}\boldsymbol{m}_2 & \dots & \boldsymbol{x}_1^{\mathrm{T}}\boldsymbol{m}_{nn} \\ \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{m}_1 & \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{m}_2 & \dots & \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{m}_{nn} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{x}_{xdn}^{\mathrm{T}}\boldsymbol{m}_1 & \boldsymbol{x}_{xdn}^{\mathrm{T}}\boldsymbol{m}_2 & \dots & \boldsymbol{x}_{xdn}^{\mathrm{T}}\boldsymbol{m}_{nn} \end{bmatrix} \\
&+ \begin{bmatrix} \|\boldsymbol{m}_1\|^2 & \|\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{m}_{nn}\|^2 \\ \|\boldsymbol{m}_1\|^2 & \|\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{m}_{nn}\|^2 \\ \vdots & \vdots & \ddots & \vdots \\ \|\boldsymbol{m}_1\|^2 & \|\boldsymbol{m}_2\|^2 & \dots & \|\boldsymbol{m}_{nn}\|^2 \end{bmatrix} \\
&= \boldsymbol{X}_{sq} - 2\boldsymbol{X}^{\mathrm{T}}\boldsymbol{M} + \boldsymbol{M}_{sq}
\end{aligned} \tag{4}$$

The fine-grained data parallelism (thread-level) for addition or subtraction of matrices could be implemented by cuBlas. It is obvious that every element in $\boldsymbol{X}_{sq}$ is the inner product of $\boldsymbol{x}_i$ and itself. Thus, $\boldsymbol{X}_{sq}$ can be calculated as the follows. 1. For every instance $\boldsymbol{x}_i$ in $\boldsymbol{X}$, $\boldsymbol{x}_i$ is element-wise multiplied by itself. 2. The previous result is left-multiplied by an all-ones matrix ($nn \times dn$, the number of nodes $\times$ the number of dimension) by `cublasSgemm`. 3. The product is transposed by `cublasSgeam`. In the same way, $\boldsymbol{M}_{sq}$ could be calculated. Finally, the matrix $\boldsymbol{D}$ could be calculated by addition and subtraction of matrices $\boldsymbol{X}_{sq}$, $\boldsymbol{M}_{sq}$ and $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{M}$.

3.3.2. Matrix Operations for Batch Update

The Eq. (2) is the essential of the batch SOM algorithm, which could be divided into calculations of numerator and denominator. They can be re-structured into matrix operations. After the following optimization, weight vectors of all nodes can be updated simultaneously on a GPU. The Gaussian function in the numerator and denominator of Eq. (2) is frequently used, and the values form a matrix denoted by $\boldsymbol{H}$ here. The element $h_{j,i}$ in matrix $\boldsymbol{H}$ is the Gaussian value of instance j and node i, which is irrelative with other elements. So, the matrix $\boldsymbol{H}$ can be computed in parallel.

$$\boldsymbol{H} = \begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,nn} \\ h_{2,1} & h_{2,2} & \dots & h_{2,nn} \\ \vdots & \vdots & \ddots & \vdots \\ h_{xdn,1} & h_{xdn,2} & \dots & h_{xdn,nn} \end{bmatrix}. \tag{5}$$

For the denominators, the matrix operation is that $\boldsymbol{H}$ is left-multiplied by a xdn-dimensional vector of ones and then multiplied by n_j. And the calculation of the numerators can be optimized as the matrix multiplication of $\boldsymbol{X}_{i,j}$, $\boldsymbol{H}$ and n_j. Both of these calculations can be implemented by `cublasSgemm`. The numerators and denominators calculated here

indicates the "impact" of the partial dataset ($\boldsymbol{X}_{i,j}$) to the codebook $\boldsymbol{M}$, and they should be reduced to the master process to update the whole model.

3.3.3. Optimizing Memory

In order to make full use of the memory bandwidth, memory optimization methods should be adopted. In CUDA, the basic unit of parallelism is thread named "kernel". A wrap is made up of 32 threads usually, considered as a basic schedule and execution unit on GPU. In addition, the wrap is also relevant to the global video memory access. The more memory access transactions are coalesced, the higher global video memory bandwidth can be utilized. Coalesced memory access should be aligned, and contiguous by the threads in a wrap to enhance the cache hit ratio. The codebook $\boldsymbol{M}$ is a two-dimensional array stored in the global video memory, which is accessed by the kernels that calculate the quotient in Eq. (2). In the kernels, the address of element (i, j) of $\boldsymbol{M}$ is calculated in Eq. (6), where *width* is the width of the array.

$$\text{add}(\boldsymbol{M}_{i,j}) = \text{add}(\boldsymbol{M}_{0,0}) + width \times j + i \tag{6}$$

To make fully coalesced, the *width* in Eq. (6) has to be set to multiples of wrap size. In the implementation, the codebook $\boldsymbol{M}$ is allocated with the width rounded up to the closest multiple of wrap size. In other kernels, only build-in types of variables and one-dimensional array are used, and the memory accesses are coalesced automatically by the compiler.

Additionally, the other optimization method is the zero-copy memory technique. In the above design, data exchange via PCI Express bus between video memory and main memory is strived to be minimized. A common procedure is that data firstly are computed by GPU, and then they are transmitted to main memory for the sequential communication among hosts. However, time cost of the exchange is still non-negligible. To maximally overlap the cost of computation and transmission via PCI-E, zero-copy memory technique should be used. The zero-copy memory is pinned memory allocated in main memory which is accessible for kernels. The use of zero-copy memory here allows the computation and transmission to be carried out simultaneously, avoiding explicit copies. As shown in Fig. 5, some of time cost is hidden.

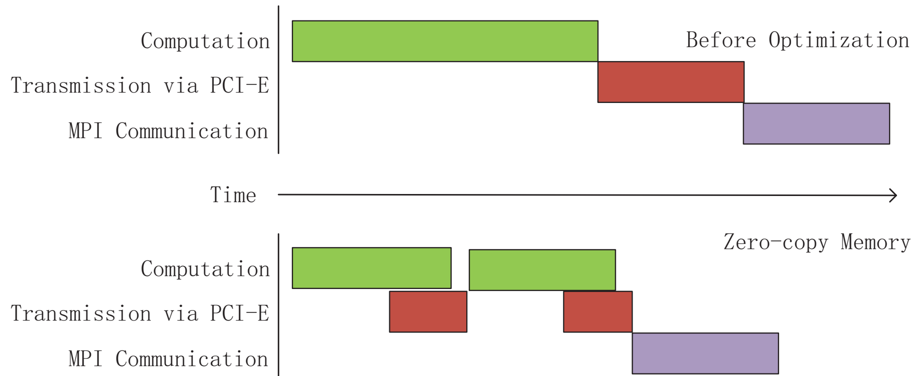


Figure 5: Optimization of zero-copy memory

4. Performance Evaluation

The main purpose of HPSOM is to shorten training time, remaining the training quality. The proposed HPSOM is evaluated by training quality (quantization error and topographic error), speedup and GPU parallel efficiency in this paper. Matlab Neural Network Toolbox and Somoclu are chosen as the baselines to evaluate the performance. The toolbox which are able to create, train and visualize SOMs is a built-in toolbox of Matlab, the famous scientific computing software. However, either parallel computation or GPU acceleration for SOM is still unavailable in the toolbox of Matlab 2018a (MathWorks, 2018). The other baseline is Somoclu 1.7.3 which is an open-source parallel SOM software. HPSOM is programmed in CUDA/C++ language with the libraries of Intel MPI 2017 and CUDA 8.

The experiments are performed in the cluster made up of four hosts whose hardware and software are listed in table 1.

Table 1: The hardware and software of a host

Type	Detail
CPU	Intel E5-2650 v4 2.20GHz
Memory	128GB DDR4
GPU	2× NVIDIA Tesla P100
Network	56Gbit/s InfiniBand
OS	Red Hat Enterprise Linux Server 7.2
MPI	Intel MPI 2017
CUDA	CUDA 8.0

4.1. Training Quality

Several quality metrics of SOM are reviewed by Pölzlbauer (2004), and two of these are most commonly used: quantization error (QE) and topographic error (TE). QE calculated by Eq. (7) is a metric of the average distance between the input data and its BMU, with a smaller value indicating a better quality. In Eq. (7), nx denotes the number of input instances.

$$QE = \frac{1}{nx}\sum_{i=1}^{nx} \|\boldsymbol{x}_i - \boldsymbol{m}_c\| \tag{7}$$

The other quality metric of SOM is TE, the measurement of topology preservation, denoting the proportion of input instances whose first and second BMUs are not adjacent. It is formulized as Eq. (8), where $c(\boldsymbol{x})$ and $c'(\boldsymbol{x})$ are the BMU and second BMU of $\boldsymbol{x}$. And a smaller value indicates a better quality.

$$TE = \frac{1}{nx}\sum_{i=1}^{nx} t(\boldsymbol{x}_i), \quad t(\boldsymbol{x}) = \begin{cases} 0 & \text{if } c(\boldsymbol{x}) \text{ and } c'(\boldsymbol{x}) \text{ are neighbors,} \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

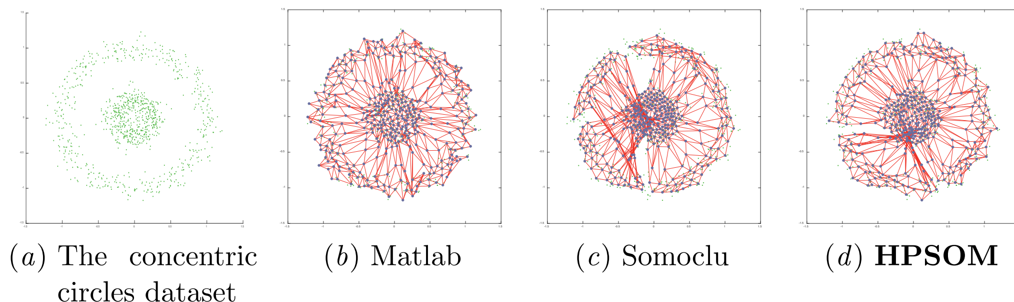
The experiment is performed with several public datasets. They are the pen-based recognition of handwritten digits dataset (Digits), the repeat consumption matrices dataset

Table 2: Training Qualities

Dataset	Circles		Digits		Tweets		Iris		Breast Cancer	
Num of Inst	150		1797		30320		150		569	
Num of Dim	2		64		11260		4		30	
Metric	TE	QE	TE	QE	TE	QE	TE	QE	TE	QE
Somoclu	0.043	0.02106	0.3033	2.2670	0.0852	**9.1954**	0.1800	**0.0722**	0.0914	1.0144
Matlab	0.084	0.02642	0.3122	2.8576	**0.0420**	10.5549	**0.1681**	0.2267	**0.0279**	**0.7282**
HPSOM	**0.026**	**0.01811**	**0.2916**	**2.2543**	0.0555	10.7157	0.1867	0.0758	0.0615	0.7768

(tweets from New York area, Tweets), the iris dataset (Iris), the breast cancer Wisconsin dataset (Breast Cancer). The datasets are from UCI machine learning repository by Dheeru and Karra Taniskidou (2017). They are used to train a 10×10 SOM with the same parameters, including 10000 epochs, hexagonal grid, and linear neighborhood radius reduction (from 3 to 1). In additionally, we generate a two-dimensional dataset (Circles) with 150 instances for visualization, whose shape looks like two concentric circles. It is illustrated in Fig. 6(a). The circles dataset is trained with a 20×20 SOM for the higher resolution, the other training parameters are the same.

As shown in Table 2, the proposed HPSOM gets the same level of training quality compared with the baselines. Especially for the circles dataset, it could be perceived visually from Fig. 6 that HPSOM achieves better training quality. The dark points represent the weight positions of the nodes in input space. The nodes in Fig. 6(d) fit the data better since less nodes are located in the gap between two circles.



(a) The concentric circles dataset (b) Matlab (c) Somoclu (d) **HPSOM**

Figure 6: The circles dataset and weight positions of nodes in input space

4.2. Speedup

In this experiment, time costs used in training the map in the same scale (10×10) by Matlab, Somoclu and HPSOM are measured. And the training parameters are same, including 5000 epochs.

This experiment is performed with the repeat consumption matrices dataset (tweets from New York area) which consists of 30320 instances with 11260 dimensions. We choose

the first 400, 800, 1600, 3200, 6400, 12800, 25600 instances from the original to evaluate the performance with various numbers of instances. In addition, the speedup (Cuomo et al., 2017) is defined as Eq. (9), where T_1 denotes the time cost of serial execution on a CPU; T_p denotes the time cost of parallel execution on p GPUs. The results are shown in Table 3 and Fig. 7.

$$S_p = \frac{T_1}{T_p} \tag{9}$$

Table 3: Time costs with various numbers of instances

Dataset	Time in seconds								
Num. of	Matlab	Somoclu			HPSOM				
Inst.	CPU	CPU	1GPU	8GPUs	CPU(Serial)	1GPU	2GPUs	4GPUs	8GPUs
400	25389.9	4210.5	329.6	85.3	4145.8	6.1	12.8	36.2	35.0
800	62852.0	8545.7	643.5	133.1	8328.8	9.4	14.0	37.2	34.8
1600	90374.3	17474.9	1222.4	215.1	17495.7	15.81	12.4	40.2	34.4
3200	146695.2	34413.9	2417.1	366.9	34446.7	30.3	24.2	43.2	38.8
6400	232230.7	68071.7	4718.0	662.4	67780.1	61.9	38.8	49.0	40.7
12800	423461.5	136594.6	9328.7	1239.0	137441.0	121.1	71.9	64.6	46.4
25600	812235.9	291177.0	18399.7	2383.7	292931.4	248.2	130.6	96.8	61.9
30320	939879.3	334786.2	21921.2	2803.9	333162.5	253.2	158.6	121.4	68.5

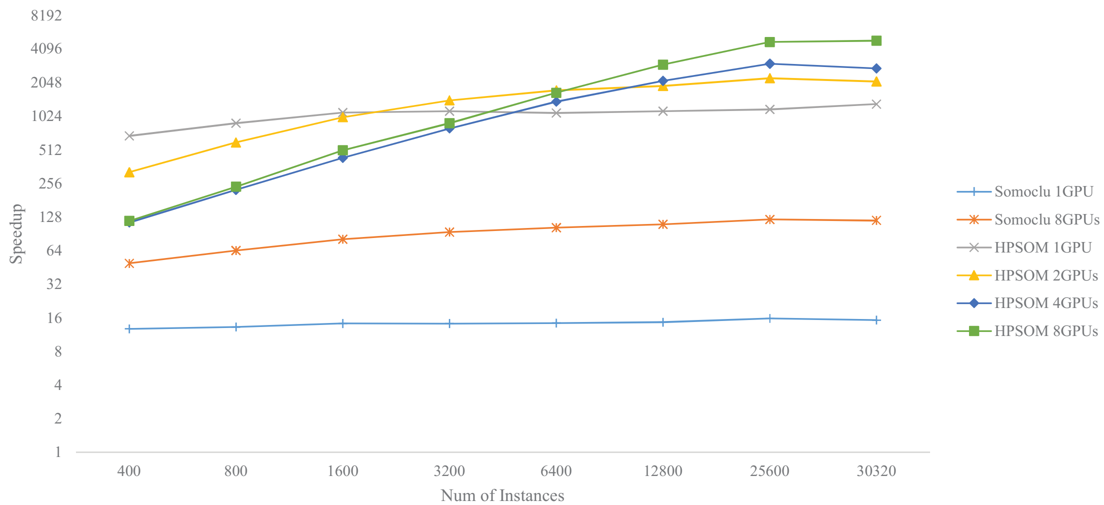


Figure 7: Speedups of the experiment

As shown in Table 3, the time costs of HPSOM are far less than the costs of Matlab and Somoclu under the same conditions. For the full dataset, the speedup of HPSOM on 8 GPUs reaches up to 4861, and it only takes 0.0073% and 2.44% of the time that used by Matlab and Somoclu respectively. As shown in Fig. 8, the speedups outperform the GPU baseline, i.e. Somoclu, reaching up to 1316, 2244, 3027 and 4861 on 1, 2, 4, 8 GPU(s) respectively at best. In the case of insufficient amount of instances, the training time of more GPUs is probably more than that of less GPU(s). The cost of inter-host communication and synchronization is obviously exposed because the scale of divided data is too small to satisfy the computing capability of GPU. In fact, the cost is mild in absolute amount. This demonstrates that the large scale parallel of HPSOM are more suitable for the large datasets in general.

4.3. GPU Parallel Efficiency

The GPU parallel efficiency is defined in Eq. (10) to evaluate the scalability, with a greater value indicating a greater scalability. In Eq. (10), gn denotes the number of GPUs, T_{g1} denotes the time cost of a GPU, and the time cost of gn GPUs is denoted by T_{gn}. The concept of parallel efficiency originates from Amdahl's law. Based on Amdahl's law, the GPU parallel efficiency defined here is only focusing on GPU because HPSOM and the baseline run on a CPU-GPU heterogeneous system and the major proportion of acceleration is offered by GPU.

$$PE = \frac{T_{g1}}{gn \times T_{gn}} \tag{10}$$

The repeat consumption matrices dataset (tweets from New York area) is used in the experiment. To evaluate the GPU parallel efficiency with the various dimension, we choose the first 40, 80, 160, 320, 640, 1280, 24560, 5120, 10240, and 11260 (all) dimensions from the dataset. The time costs and GPU parallel efficiencies are shown in Table 4 and Fig. 8.

As shown in Fig 8, the efficiencies of 2 GPUs are at the level of 0.8, and they are greater than those of 4 GPUs and 8 GPUs. The more GPUs there are, the less data each GPU needs to process. Therefore, the efficiencies get descend under the same dimension when more GPUs are used. The trend is reasonable, and the efficiencies of 8 GPUs are still at the level of 0.5. This demonstrates that HPSOM is scalable.

5. Conclusion

As a classical artificial neural network, self-organizing map is widely applied and confronted with the challenge of efficiency in the case of massive data, high dimensional data or a large-scale map. A scalable heterogeneous parallel SOM is proposed in this paper. While ensuring training quality, HPSOM could significantly accelerate the training of SOM, especially in the case of massive high-dimensional data. It is also able to be scaled up to multi-GPU efficiently. Furthermore, it is valuable for the parallelization and optimization on other machine learning models on multi-GPU. The major contributions include:

1. A two-level data parallelism framework of SOM is designed, which is suitable for multi-GPU platform.

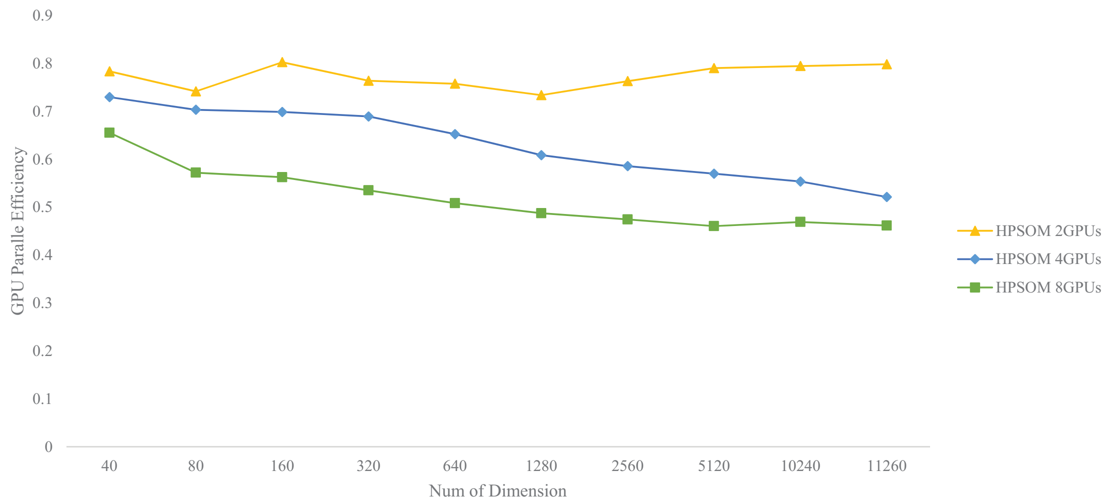


Figure 8: GPU Parallel efficiencies of the experiment

Table 4: Time costs with various numbers of input dimension

Dataset	Time in seconds								
Num. of	Matlab	Somoclu			HPSOM				
Dim.	CPU	CPU	1GPU	8GPUs	CPU(Serial)	1GPU	2GPUs	4GPUs	8GPUs
40	13306.4	3234.9	358.6	60.1	2204.1	90.4	57.7	31.0	17.2
80	25874.2	3516.8	504.3	68.9	3323.3	70.3	47.4	24.9	15.4
160	28628.1	5995.2	662.9	87.6	5564.3	75.6	47.1	27.0	16.8
320	49113.6	10526.9	959.6	133.6	10116.5	74.8	48.9	27.1	17.5
640	88588.4	19627.4	1713.7	214.9	19296.5	78.0	51.5	29.9	19.2
1280	141953.1	38317.8	3068.4	379.1	37577.2	87.3	59.4	35.8	22.4
2560	254466.9	74297.4	5521.8	699.8	74692.4	111.7	73.2	47.7	29.4
5120	444207.0	147791.0	10176.5	1314.7	149181.1	154.5	97.8	67.8	41.9
10240	86298.9	294240.3	19991.8	2555.3	302948.7	241.5	152.0	109.1	64.3
11260	939879.3	334786.2	21921.2	2803.86	333162.5	253.2	158.6	121.4	68.5

2. It is implemented based on MPI and CUDA with considerable speedup and satisfied parallel efficiency.

3. Several most time-consuming modules in the batch SOM algorithm are re-structured into matrix operations, which are implemented by cuBlas to greatly utilize the float computing capability of GPU.

4. Optimizations of memory access are presented, utilizing the memory more efficiently.

The experiments show that HPSOM is effective, efficient and scalable. Compared with Matlab and Somoclu, HPSOM provides the training quality at the same level. And the speedups of HPSOM are higher than those of Somoclu by an order of magnitude. In the case of the repeat consumption matrices dataset, HPSOM on 8 GPUs takes only 0.0073% and 2.44% of the time that used in training by Matlab and Somoclu respectively. In the best case, it offers an acceleration of 4861 times over the serial execution of CPU, while the GPU parallel efficiency still remains about 0.5.

Acknowledgments

This work is partially supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (2016A05) and the Development Program of Experimental Teaching Equipment of East China Normal University (2017006).

References

Golnoush Abaei, Ali Selamat, and Hamido Fujita. An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Knowledge-Based Systems*, 74:28–39, 2015.

Wei Ai, Kenli Li, Cen Chen, Jiwu Peng, and Keqin Li. DHCRF: A distributed conditional random field algorithm on a heterogeneous CPU-GPU cluster for big data. In *2017 IEEE 37th International Conference on Distributed Computing Systems*, pages 2372–2379. IEEE, 2017.

Amin Allahyar, Hadi Sadoghi Yazdi, and Ahad Harati. Constrained semi-supervised growing self-organizing map. *Neurocomputing*, 147:456–471, 2015.

Yizong Cheng. Convergence and ordering of kohonen's batch map. *Neural Computation*, 9 (8):1667–1676, 1997.

Kang-Wook Chon, Sang-Hyun Hwang, and Min-Soo Kim. GMiner: A fast GPU-based frequent itemset mining method for large-scale data. *Information Sciences*, 439:19–38, 2018.

Salvatore Cuomo, Pasquale De Michele, Emanuel Di Nardo, and Livia Marcellino. Parallel implementation of a machine learning algorithm on GPU. *International Journal of Parallel Programming*, 3:1–20, 2017.

Dua Dheeru and Efi Karra Taniskidou. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Teuvo Kohonen. Essentials of the self-organizing map. *Neural Networks*, 37:52–65, 2013.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

Nan Liu, Jinjun Wang, and Yihong Gong. Deep self-organizing map for visual classification. In *2015 International Joint Conference on Neural Networks*, pages 1–6. IEEE, 2015.

Anima Majumder, Laxmidhar Behera, and Venkatesh K. Subramanian. Emotion recognition from geometric facial features using self-organizing map. *Pattern Recognition*, 47(3): 1282–1293, 2014.

MathWorks. Train neural network - MATLAB train, May 2018. URL https://www.mathworks.com/help/nnet/ref/train.html.

Rory Mitchell and Eibe Frank. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science*, 3:e127, 2017.

Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4):69:1–69:35, 2015.

Ehsan Mohebi and Adil Bagirov. A convolutional recursive modified self organizing map for handwritten digits recognition. *Neural Networks*, 60:104–118, 2014.

NVIDIA. CUDA C programming guide, May 2018. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

Georg Pölzlbauer. Survey and comparison of quality measures for self-organizing maps. In *Proceedings of the Fifth Workshop on Data Analysis*, pages 67–82. Elfa Academic Press, 2004.

Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. SOM clustering using Spark-MapReduce. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1727–1734. IEEE, 2014.

Setu Shah and Xiao Luo. Exploring diseases based biomedical document clustering and visualization using self-organizing maps. In *19th IEEE International Conference on e-Health Networking, Applications and Services*, pages 1–6. IEEE, 2017.

Mahmoud Soua, Rostom Kachouri, and Mohamed Akil. GPU parallel implementation of the new hybrid binarization based on kmeans method (HBK). *Journal of Real-Time Image Processing*, 14(2):363–377, 2018.

Yimu Wang, Yun Pan, Yanchen Long, Xiaolang Yan, and Ruohong Huan. Parallel implementation of handwritten digit recognition system using self-organizing map. *Journal of Zhejiang University (Enginnering Science)*, (4):742–747, 2014.

Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. ThunderSVM: A Fast SVM Library on GPUs and CPUs. *Journal of Machine Learning Research*, 19(21):1–5, 2018.

Peter Wittek, Shi Gao, Ik Lim, and Li Zhao. somoclu: An efficient parallel library for self-organizing maps. *Journal of Statistical Software*, 78(9):1–21, 2017.

Yi Xiao, Ruibin Feng, Zi-Fa Han, and Chi-Sing Leung. GPU accelerated self-organizing map for high dimensional data. *Neural Processing Letters*, 41(3):341–355, 2015.