


# Parallel Implementation of a Machine Learning Algorithm on GPU

Salvatore Cuomo<sup>1</sup> · Pasquale De Michele<sup>1</sup> ·  
Emanuel Di Nardo<sup>2</sup> · Livia Marcellino<sup>2</sup> 

Received: 11 July 2017 / Accepted: 27 December 2017 / Published online: 30 December 2017  
© Springer Science+Business Media, LLC, part of Springer Nature 2017

**Abstract** The capability for understanding data passes through the ability of producing an effective and fast classification of the information in a time frame that allows to keep and preserve the value of the information itself and its potential. Machine learning explores the study and construction of algorithms that can learn from and make predictions on data. A powerful tool is provided by self-organizing maps (SOM). The goal of learning in the self-organizing map is to cause different parts of the network to respond similarly to certain input patterns. Because of its time complexity, often using this method is a critical challenge. In this paper we propose a parallel implementation for the SOM algorithm, using parallel processor architecture, as modern graphics processing units by CUDA. Experimental results show improvements in terms of execution time, with a promising speed up, compared to the CPU version and the widely used package SOM\_PAK.

**Keywords** Machine learning · Fast data analysis and retrieval · Self-organization map · GP-GPU

---

✉ Livia Marcellino  
livia.marcellino@uniparthenope.it

Salvatore Cuomo  
salvatore.cuomo@unina.it

Pasquale De Michele  
pasquale.demichele@unina.it

Emanuel Di Nardo  
emanuel.dinardo@uniparthenope.it

<sup>1</sup> Department of Mathematics and Application, Complesso Universitario di Monte Sant' Angelo, University of Naples Federico II, Via Cintia, Naples, Italy

<sup>2</sup> Department of Science and Technology, Centro Direzionale Isola C4, University of Naples Parthenope, Naples, Italy

## 1 Introduction

Machine learning is closely related to computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. A self-organizing map (SOM) is a type of unsupervised neural networks that showed to be very effective to produce a visualization from a set of data with a high number of dimensions. In other words, SOM is a learning algorithm which supports data exploration in an eclectic number of different application fields, due to the substantial independence of their expressive possibilities from the actual origin of data to be examined. The value of SOM, that relies on the capability of creating an internal, coherent spatial representation of data from differently organized sources, is a precious resource in a phase of computing history in which there is abundance of multidimensional data from different, coordinated or uncoordinated sources, that led to the rise of Big Data and related applications. Anyway, two of the four V's of Big Data (velocity, variety, volume and veracity) produce a push for performances, due to the need to fast process large amounts of data; and, in general, even when they do not qualify to be classified in the Big Data realm, modern applications present a richness of available data that need to be harvested quickly in order to produce valuable results [5, 7, 15].

SOMs are a well established approach to data classification and representation, with many decades of study and application: here we decided to point out some of the main references, that happen to be spread over the first three decades of their long story, starting in 1981, relying on the most comprehensive surveys and some interesting additional references, but excluding the first works, that are covered in the surveys, and are anyway available as sources cited in the surveys to the readers that may prefer a direct experience of the papers of the first decade. For a comprehensive introduction and a historical perspective on SOM and related applications the reader can refer to [20] and [21], that widely present SOM and survey some selected application fields. In [23] the authors survey instead a number of effective engineering applications of SOM in the first decades of use, ranging from process and systems analysis (e.g. fault identification, visualization of machine states), statistical patterns recognition (e.g. speech recognition, texture identification, computer vision), robotics (e.g. arm control, navigation), telecommunications (e.g. signal detection, channel equalization, interference cancellation, codification of images) and other fields, including a very large number of useful references. Other examples of SOM based applications are anomaly detection (e.g. [13]), face recognition in different conditions ([35], in combination with other techniques). Nevertheless, SOM have been also used in Data Science applications due to their efficiency in clustering problems, in support to Data Mining applications: interested readers may find an example in [36], while [2] provides a comparison between SOM and other classical analysis techniques of Data Science interest, namely Cluster Analysis and Principal Component Analysis, for the exploration of large data sets. Their classification features have been also applied to document categorization and classification, both on collections (e.g. [11, 24]) and on the web (e.g. [4]), due to their capability of semantic classification, and, in general, of producing abstractions [20, 21, 36]. It is relevant to underline the fact that SOM may be used in multilevel approaches, to compose structures of SOMs with specialized features that collaborate,

with an appropriate composition in stages, to solve problems with a stepwise logic (e.g. [16,23,36]).

From the theoretical point of view, one of the most relevant features of SOM is their capability of building an inner spatial representation out of external data. The nature and the characteristics of this mapping ability has been extensively studied, due to its importance, and to the fact that, being it generated by a non supervised approach, the trustability of the emergent representation is a critical factor in applications. For a discussion on topology preservation in the internal representation with respect to the structure of external data, we suggest to read [18], while for a quantitative examination of neighborhood preservation between the two domains we suggest [3], that explores continuity, resolution and coherence with input probability distributions of the mapping. Finally, [22] deals with the problem of making large SOMs and [17] presents some variants to SOM, with [1] pointing out an extension that allows dynamicity and controlled growth.

Within this quest for performances, in this work we present an implementation of a SOM on CUDA-GPU architectures. GPUs are an important and cost effective resource that has shown its exceptional effectiveness in computing applications, at the point that they emerged as a fundamental stage of what are currently known as GPGPU (General Purpose computing on Graphics Processing Units). The benefits of GPU hardware derive from the efficiency of software libraries capable of offering the access to the computing power of graphics processors, do not have the architectural constraints of CPUs, because they are capable of massive parallelism essentially due to the high number of processors available for computation.<sup>1</sup> In the following, we assume that the reader is already familiar with the main aspects of GPGPU applications: for a survey, we suggest [27,38] and [28] as a starting point, while for a quick glance on performance modeling and related issues about GPGPU architectures we suggest [9,12].

The use of GPU to support the execution of SOM based application is well established in the literature. This is a natural evolution, as SOM greatly benefit of the massive parallelism offered by GPU. One of the first proposed solutions is reported in [43], and many interesting alternatives and developments toward distribution exist: for high dimensional SOM [26,40,41], for massively parallel SOM based on cellular approaches [37,42], or the recent [30]. The number of applications is significant, both exploiting SOM or batch SOM: in [39] authors deal with text mining in a map-reduce based application; in [31,34] authors present applications to computer vision, describing a parallel image processing pipeline; moreover, in [29] a large data real time classification application is implemented; finally, in [6,32] an optical flow estimator is provided.

In this paper, based on the approach proposed in [25], we propose a novel GPU-implementation of SOM. Our algorithm and its implementation are characterized by the exploitation of the cuBLAS<sup>2</sup> library, with its optimized routines for basic linear algebra operations. Moreover, we propose an improvement to the balanced approach both in the input stage and in the computational step whit respect to the original paral-

---

<sup>1</sup> <http://www.nvidia.com>.

<sup>2</sup> <https://developer.nvidia.com/cublas>.

lel, suggested in [25]. This allows to obtain a fully-parallel algorithm in which threads simultaneously work on a single element each, significantly reducing synchronization and waiting time. Lastly, compared to the existing parallel methods proposed in literature: the main advantage of our solution, that will be described in detail in the rest of the paper, is the load balancing capability for the computing workload.

The paper is organized as follows. Section 2 presents the definition of the SOM algorithm. Section 3 describes the approach of this work. In Sect. 4 we report the experiments performed to show the effectiveness of the approach. Finally, Sect. 5 closes the paper.

## 2 The Self-Organizing Maps Algorithm

A Self-Organizing Map, also known as SOM, is a kind of unsupervised neural network that produces a representation of training samples in a low dimensional space preserving topology properties of samples. This property makes the SOM particularly useful for displaying high dimension data.

The first model for SOM has been described in [19] and is also known as *Kohonen Maps*. In this type of neural network the output neurons are organized in low-dimensional grids (2D or 3D). Each input is connected to all output neurons. In other words, the SOM model is a fully connected network where each neuron has a weight vector of the same size of input vector and the size of the input vector is generally much higher than the size of the output grid.

Aim of the network is to specialize different parts of the lattice to react to input patterns to reflect the behaviour of cerebral cortex in the human brain. It is used a kind of training called *competitive*. Each training step is organized in the following way:

- For each input vector the Euclidean distance from all neurons in the map is computed;
- The most representative neuron of the input vector is that which has minimum distance and it is called *Best Matching Unit* (BMU);
- Distance based on the BMU position in the map is computed for each neuron to find its neighbourhood;
- All neuron in the neighbourhood are updated during adaptation phase using a BMU influence function and learning rate.

The approach depends on the distance of the neurons.

More precisely, let  $I(t)$  be an input vector at time  $t$  sent to the network, suitable weights  $W_v$  are defined, for each neuron  $v$ , in order to determine the most representative neuron, as follows:

$$W_v(t + 1) = W_v(t) + \Theta(v, t)\alpha(t)[I(t) - W_v(t)] \quad (1)$$

where  $I(t) - W_v(t)$  is the *error regularization parameter*, i.e. the difference between the weight vector at time  $t$  and the input at time  $t$ ,

$$\sigma(t) = \sigma_0 \exp\left(-\frac{t}{\log \sigma_0}\right)$$

is the *neighbourhood size*,

$$\alpha(t) = \alpha_0 \exp\left(-\frac{t}{t_{end}}\right)$$

is the *learning rate* and  $\Theta(v, t)$  is the *BMU influence*, which depends of the distances in the network between the BMU and the neuron  $v$ . Formally, it is:

$$\Theta(v, t) = \exp\left(-\frac{dist(BMU, v)^2}{2\sigma(t)^2}\right) \quad (2)$$

This process results in a movement of neuron toward the input, so that, at the end of the training process, all neurons are respectively the best representative of an input vector. The SOM procedure is divided into two main phases: the *training phase* for the network learning and the *classification phase* to check if an input belongs to a certain class (here the only step to perform is the search for the BMU).

It is clear that the computational complexity of the first phase is a critical aspect, because there is a large amount of data to be processed, as shown in the Algorithm 1.

---

**Algorithm 1** The SOM algorithm: training phase.

---

**input:**  $I(t)$ ,  $W_v(t)$ ,  $\Theta(v, t)$ ,  $\sigma(t)$

for (each neuron  $v$ )

1. search the BMU: compute  $I(t) - W_v(t)$  by using the Euclidean distance formula;
2. search of BMU neighborhood: checking the distance between neurons and BMU;
3. weights update: by using the relationship (1)

endfor

---

### 3 Our GPU-Parallel SOM Implementation

Our proposal provides an alternative parallelization of the *training phase*, that is the bottleneck for standard SOM. The fact that we are dealing with an unsupervised learning case implies that each neuron depends on all others.

With these assumptions it is very difficult to execute parallel neuron training, so our approach tries to enhance the main points in which a speed up may allow to achieve better performances. We analysed the following three steps of the previous Algorithm 1: (1) search of BMU; (2) search of BMU neighbourhood; (3) adaptation (weights' update).

For each of these steps, we have implemented a CUDA parallel kernel. Therefore, the parallel version of the Algorithm 1 is listed in the Algorithm 2.

In the following, for each kernel, we will focus on two fundamental points: how block size and grid size are built and how the kernel algorithm works. However, we will first provide some information on the environment which we use and on the utility that improve the efficiency of our implementation.

---

**Algorithm 2** Our GPU-parallel SOM algorithm: training phase.
 

---

```

input:  $I(t)$ ,  $W_v(t)$ ,  $\Theta(v, t)$ ,  $\sigma(t)$ 
for (each neuron  $v$ )
    call the kernel findBMU
    call the kernel findBMUDistances
    call the kernel adaptation
endfor
  
```

---

### 3.1 Structure

The implementation provides a three dimensional map in which two dimensions are given by the map dimensions, while the third one is given by the weight vector for each point in the map; this map is implemented as a mono-dimensional map to take advantage of array indexing. All computations are done in the GPU internal memory, to avoid transfer delays between main memory and device memory. We remember that in the GPU device, we have two kind of memories, namely the *global memory* (shared by all elements in execution on GPU: thread) and a *shared memory* (shared by all threads in a block) that allow threads to works in a smaller but faster memory. In order to take benefit from this kinds of memory, all computations are designed to reduce the number of operations in global memory and to increase those that may be executed in the shared memory. Finally, neurons weight vectors are randomly uniform initialized.

A key feature of our algorithm is the use of the cuBLAS library. The cuBLAS library is an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA-CUDA runtime. It allows the user to access the computational resources of NVIDIA Graphics Processing Unit (GPU). Starting with CUDA 6.0, the cuBLAS Library now exposes two sets of API, the regular cuBLAS API which is simply called cuBLAS API and the CUBLASXT API. To use the cuBLAS API, the application must allocate the required matrices and vectors in the GPU memory space, fill them with data, call the sequence of desired cuBLAS functions, and then upload the results from the GPU memory space back to the host. The cuBLAS API also provides helper functions for writing and retrieving data from the GPU. To use the CUBLASXT API, the application must keep the data on the Host and the Library will take care of dispatching the operation to one or multiple GPUs present in the system, depending on the user request.

### 3.2 Search of BMU

The first step of training consists in searching the BMU corresponding to the step 1 of the Algorithm 1. This is accomplished by computing the Euclidean distance among input vector and all neurons weight vectors. Its complexity depends on the map size and weight vector size. The parallelization of this step ensures the achievement of best performance all over the network. In the following we describe the kernel `findBMU` and its configuration for this first step.

### 3.2.1 Block and Grid Size

Our implementation uses a three dimensional array, so an accurate thread organization analysis is needed. The kind of network that we are studying easily exceeds the third dimension in block-size, so in this case, it is not possible to take advantages of GPU hardware design. To overcome this problem, a bi-dimensional block structure is used where the number of columns (second dimension) is equal to the weight size. Instead the number of neurons processed at the same time became the first dimension. In this way it is possible to build a block-size that is totally dependent on the weight size, that is the element on which the most computations are executed.

Other relevant efficiency factors of our design is that computations always happen on a number of blocks that is a power of 2, and that an entire warp (corresponding to 32 simultaneously threads) is always used. In order to achieve this, a virtual padding is applied when needed to the weight size, to align it to the next power of 2. If needed, the same adjustment is done on the first dimension. This settings can be obtained by following the steps:

- (1) Retrieve the maximum number of simultaneous threads
- (2) Compute the next power of 2 on the weight size;
- (3) Divide the number of threads by the weight size, so to obtain the maximum number of rows;
- (4) Compute the minimum between the maximum number of rows and the maximum number of rows in the matrix;

At this point it is also possible to compute the grid-size, number of total blocks. It is easily equal to the map columns size on the columns, first dimension is the next power of 2 of the matrix rows divided for the block row size.

### 3.2.2 Kernel

The kernel `findBMU` is the implementation of a task executed on the GPU. The data set needed in device memory, to enable the kernel to run, includes: (1) the input vector; (2) the neurons' weight; (3) the input vector; the map size; (4) the output array for distances. As already mentioned, the aim is to compute Euclidean distance in a parallel fashion. Therefore, we start by computing the square of the difference of the elements. All differences between the elements of the input vector and the weight vector are put in the shared memory. To avoid global memory access the value in shared memory is multiplied with itself, all threads need to wait the end of this first task. Considering the code fragment reported in Algorithm 3, it is possible to observe that it requires one access to global memory for `mtx` and one for `vec`, instead of two accesses for both.

Next, all elements on each row are summed. For this sum, as shown in Algorithm 4, a parallel reduction is applied in order to maximize the number of parallel working threads, reducing memory accesses and sum operations (as suggested in [14]).

As performed this sum, the first position of each row has been replaced by the specified sum, computed on that row. Subsequently, the square root is computed and assigned to the output array. Finally, a search of the minimum of these distances, to find the position of BMU in the map, is carried. This will be done using the

---

**Algorithm 3** Kernel `findBMU`: efficient access to global memory.

---

```

...
idxShared=rowIdx*blockDim.y + dimIdx;
dShared[idxShared] = mtx[mtxRowIdx*n*z + colIdx] - vec[dimIdx];
dShared[idxShared] *= dShared[idxShared];
...
__syncthreads();
...

```

---

**Algorithm 4** Kernel `findBMU`: parallel reduction.

---

```

...
FOR (int i = blockDim.y/2; i > 0; i-)
  IF (dimIdx < i)
    index=rowIdx*blockDim.y + dimIdx
    dShared[index] += dShared[i+(index)];
  ENDIF
ENDFOR
__syncthreads();
...

```

---

`cublasIdamin` function of the cuBLAS library. By this function, which finds the smallest index of the minimum magnitude element of a vector, we achieve a significant increasing in terms of performance.

### 3.3 Search of BMU Neighborhood

The second step consists in the search of the distance between neurons and BMU by using matrix coordinates instead of weight vectors, corresponding to the step 2 of the Algorithm 1. This is fundamental, because BMU needs to influence other neurons based on their position to permit their weight vector gets close to input vector. Although this operation is sufficiently simple also on big maps, as it only deals with two dimensions, it is possible to have a further performance gain because all data are already on device memory. In the following we describe the kernel `findBMUDistances` and its configuration for this second step.

#### 3.3.1 Block and Grid Size

The main problem of this phase is that a remapping is needed between three dimensional indexing and two dimensional indexing. In fact, computing happens on a two dimensions structure and weight size is not needed. The general algorithm is the same of Sect. 3.2.1, but a fake weight size of an arbitrary power of 2 is used, instead of number of columns in the grid is equal to the number of columns in the matrix plus one divided by the false weight size.

#### 3.3.2 Kernel

In the kernel `findBMUDistances` the computation is based on a squared Euclidean distance on two dimensions. So the input data are given by the position of the BMU



on the map, map size and output vector. Computations are performed easily only on index as shown in Algorithm 5.

---

**Algorithm 5** Kernel `findBMUDistances`: squared Euclidean distance.

---

```

...
d2=pow((double) (colIdx - bmuColIdx), 2);
dist[neighborPos] = pow((double) (rowIdx - bmuRowIdx), 2) + d2;
...

```

---

### 3.4 Adaptation (Weights Update)

In the adaptation step, corresponding to the step 3 of the Algorithm 1, there is a weight updating of neighbor neurons, by means of a movement towards the input vector, with an important influence degree by BMU, using a gaussian function and a learning rate that decay at each iteration. We observe that this step can not be considered as a RBF interpolation because it is not used a layer output which derives from an function RBF, but the Gaussian functions are used only to calculate the influence of the neighbors. One idea would be to combine the two strategies to have better results, see [8, 10, 33]. As previous, in the following we describe the kernel adaptation and its configuration for this last step.

#### 3.4.1 Kernel, Block and Grid Size

The configuration of block and grid are similar to previous sections, while the kernel adaptation requires the following parameters:

- the BMU position;
- the map size;
- the neurons map;
- the input vector;
- distances from BMU;
- the current epoch;
- the total epoch;
- the learning rate;
- the current neighborhood size;

All neurons are updated according to their distance from BMU. Each thread checks if the distance of the current neuron is less than the double of the current neighbourhood size, then the influence is computed and each element of the weight vector is updated by threads after Eq. 1, as shown in Algorithm 6.

In this case our solution does not use shared memory, because the overall time needed to copy the value in the shared memory and the waiting of the neurons results in a longer time than that required to execute the operation directly in global memory.

**Algorithm 6** Kernel adaptation: parallel neurons check and update.

---

```

...
IF (distance[distIdx] < radiusSquare)
    influence = neighborhoodFunction(distance[distIdx], radiusSquare,
    learningRate);
    map[position] += influence * decay * (input[ty] - map[position]);
ENDIF
...

```

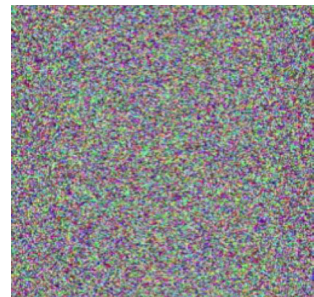
---

## 4 Experiments

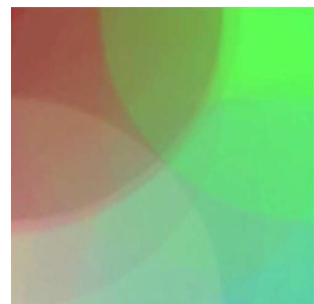
In this section a comparison between CPU and GPU implementations is proposed, to actually estimate the performance improvement. GPU hardware is a nVidia Quadro K4200 with compute capability 3. Load factor varies with map size and weight vector size, with a fixed input size of 100 samples and a total number of epochs equal to 1000 (Figs. 1, 2).

The first part of the evaluation was to determine whether the maps produced by our GPU-parallel SOM are reliable and correct. To test how the algorithm carries out the training phase and how changes at every new step you chose to test a classic example, which shows both as learning proceeds, both the topological properties of the neural network in question. This test is a learning colors test: we start from a 3D set of random input, where each weight represents the 32-bit value of RGB channels in an image. Each pixel in the image represents a neuron, thus the size of SOM reflect the image size. Initially, the values are chosen randomly, as shown in Fig. 3.

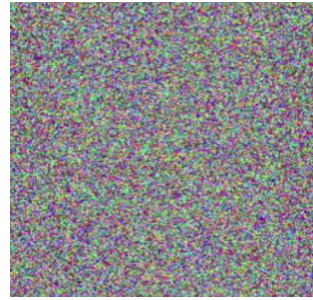
**Fig. 1** Random initialization of the neuron weights



**Fig. 2** Ordered distribution of colors (Color figure online)



**Fig. 3** Random weights' initialization



**Fig. 4** Ordered distribution of colors (Color figure online)



As learning goes the color begin to be distributed in accordance with natural color system (see Fig. 4), and at each step we verify that the range of the neurons decreases more and more covering a smaller area, until act locally.

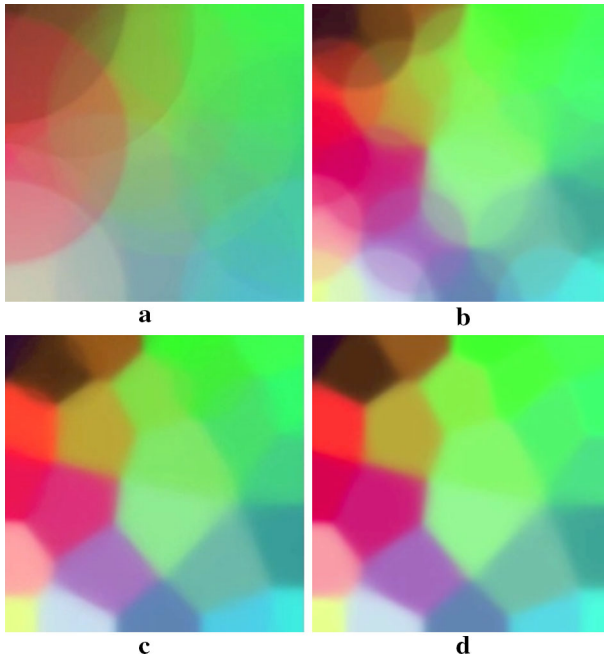
In this case, the learning continues until the end of the iterations, but an alternative criterion may be to verify the quantization error between neurons and input and stop the process when this is small enough. In Fig. 5 we show some learning step.

The first comparison is based on the total time needed to train the networks. In Figs. 6 and 7 weight vector size is respectively set to 16 and 64, and it is possible to observe a significant speed-up of GPU over CPU. The same results are reported in Tables 1 and 2 in order to show effective times obtained from the tests.

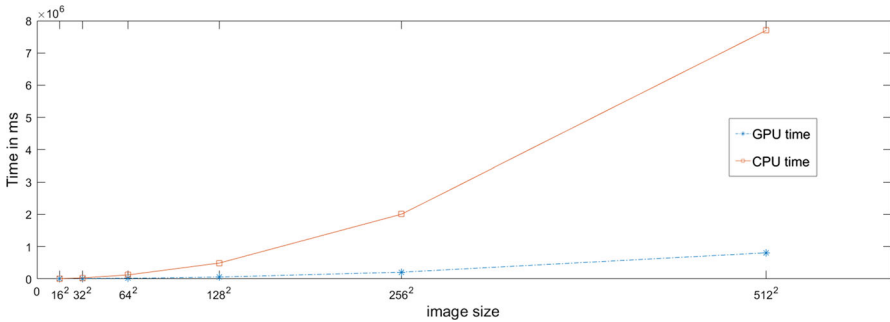
Moreover, results from sequential SOM (on CPU) and our parallel SOM algorithm (on GPU) were compared to results from SOMPAK<sup>3</sup> (on a single processor) in Table 3. The SOM\_PAK program package contains all programs necessary for the correct application of the Self Organizing Map algorithm in the visualization of complex experimental data. The first version 1.0 of this program package was published in 1992 and since then the package has been updated regularly to include latest improvements in the SOM implementations.

For the second test, from Figs. 8, 9, 10, 11, 12 and 13, average times are shown for each operation described in Sect. 3. It is clear, and in line with our expectations, that GPU outperforms CPU.

<sup>3</sup> <http://www.cis.hut.fi/research/som-research/nncr-programs.shtml>.



**Fig. 5** Training in progress

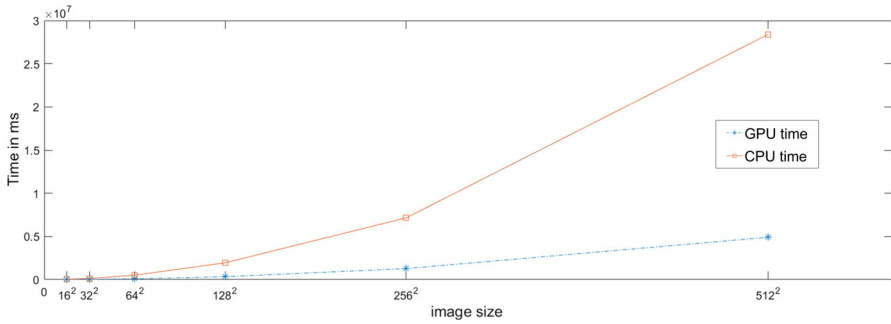


**Fig. 6** Comparison between GPU and CPU execution times in ms (weight = 16)

Further experiments are carried out by means of the cuda VISUAL profiler tool.<sup>4</sup> The VISUALprof allows to collect and view profiling data when the software runs. By using it, we observed the performance of our three kernels. In the following we report the results obtained.

More precisely, the first step in analysing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of

<sup>4</sup> <https://developer.nvidia.com/nvidia-visual-profiler>.



**Fig. 7** Comparison between GPU and CPU execution times in ms (weight = 64)

**Table 1** Training time (in ms) and when the weight is equal to 16

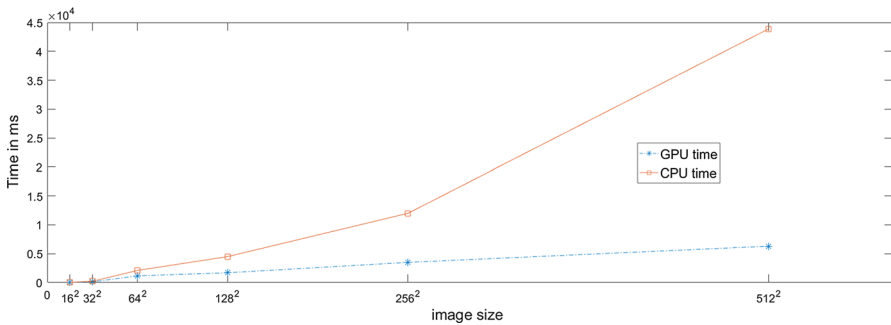
Arch.	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512
GPU	$9.86 \times 10^3$	$1.41 \times 10^4$	$2.16 \times 10^4$	$5.96 \times 10^4$	$2.11 \times 10^5$	$8.12 \times 10^5$
CPU	$9.72 \times 10^3$	$3.42 \times 10^4$	$1.26 \times 10^5$	$4.94 \times 10^5$	$2.01 \times 10^6$	$7.70 \times 10^6$

**Table 2** Training time (in ms) and when the weight is equal to 64

Arch.	16 × 16	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512
GPU	$1.55 \times 10^4$	$3.12 \times 10^4$	$9.23 \times 10^4$	$3.31 \times 10^5$	$1.27 \times 10^6$	$4.90 \times 10^6$
CPU	$3.37 \times 10^4$	$1.28 \times 10^5$	$5.04 \times 10^5$	$1.94 \times 10^6$	$7.15 \times 10^6$	$2.84 \times 10^7$

**Table 3** Training time (in ms) for SOM’s algorithms on CPU, on CPU with SOM\_PAK and on GPU, and weight is equal to 64

	16 × 16	64 × 64	128 × 128
GPU	$1.55 \times 10^4$	$9.23 \times 10^4$	$3.31 \times 10^5$
CPU	$3.37 \times 10^4$	$5.04 \times 10^5$	$1.94 \times 10^6$
SOM_PAK	$4.80 \times 10^3$	$1.57 \times 10^5$	$1.27 \times 10^6$



**Fig. 8** findBMU kernel: average times GPU and CPU, per single operation, weight = 16

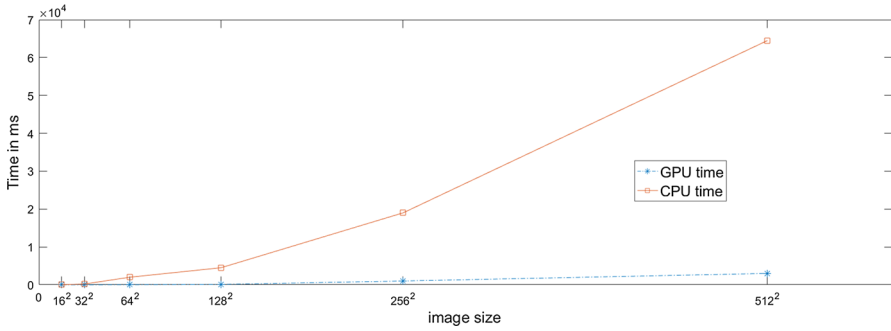


Fig. 9 findBMUDistances kernel: average times GPU and CPU, per single operation, weight = 16

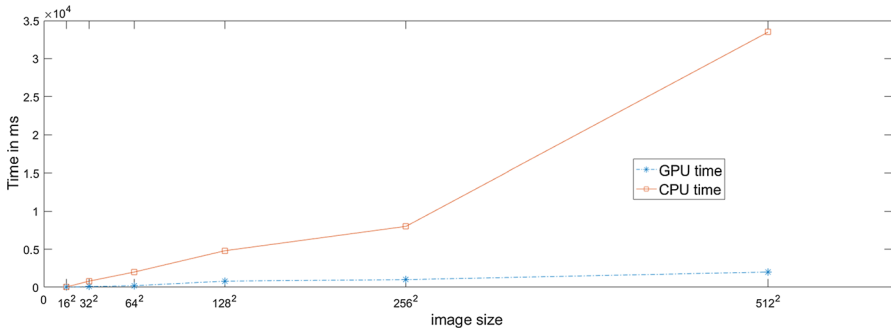


Fig. 10 adaptation kernel: average times GPU and CPU, per single operation, weight = 16

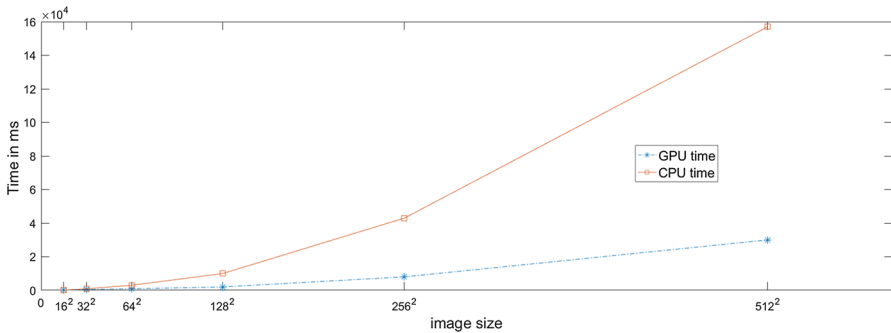
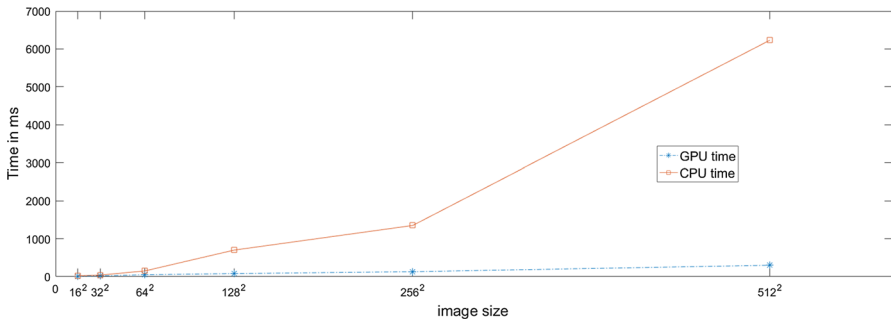


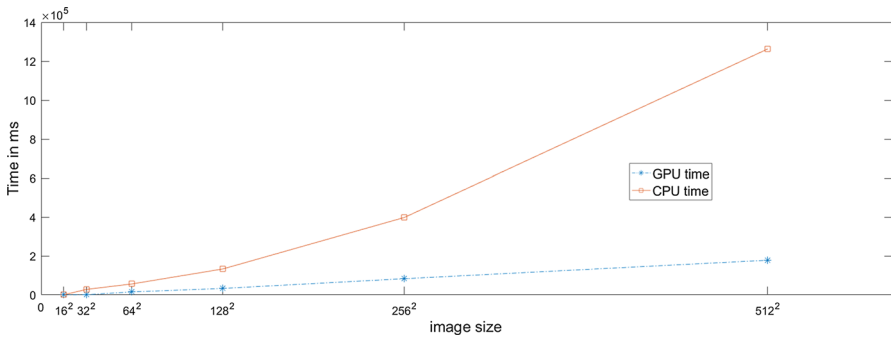
Fig. 11 findBMU kernel: average times GPU and CPU, per single operation, weight = 64

latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel’s actual occupancy.

For the findBMU kernel results are shown in Table 4. The results indicate that the performance of kernel findBMU is most likely limited by instruction and memory latency. You should first examine the information in the “Instruction And Memory



**Fig. 12** findBMUDistances kernel: average times GPU and CPU, per single operation, weight = 64



**Fig. 13** adaptation kernel: average times GPU and CPU, per single operation, weight = 64

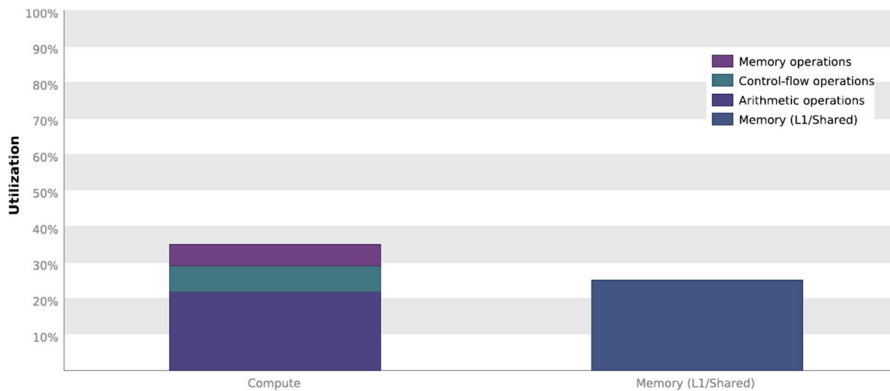
**Table 4** Analysis report for the findBMU kernel

Duration	11.97 ms
Grid size	[32,256,1]
Block size	[8,128,1]
Registers/thread	20
Shared memory/block	8 KiB
Shared memory requested	48 KiB
Shared memory executed	48 KiB
Shared memory bank size	4 KB

Latency” section to determine how it is limiting performance. The findBMU kernel report analysis exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of “Quadro K4200”. These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues (see Fig. 14).

Similar results are obtained for kernels findBMUDistances and adaptation. The analysis report for these kernels is shown in Tables 5 and 6 and Figs. 15 and 16.

The last test shows the effects of using the GPU in the case in which the network size is not a power of 2. The aim of this test is a simple consideration: the hardware



**Fig. 14** findBMU kernel report analysis

**Table 5** Analysis report for the findBMUDistances kernel

Duration	47.39 ms
Grid size	[2,128,1]
Block size	[128,2,1]
Registers/thread	23
Shared memory/block	0 KiB
Shared memory requested	48 KiB
Shared memory executed	48 KiB
Shared memory bank size	4 KB

**Table 6** Analysis report for the adaptation kernel

Duration	4.413 ms
Grid size	[32,256,1]
Block size	[8,128,1]
Registers/thread	36
Shared memory/block	0 KiB
Shared memory requested	48 KiB
Shared memory executed	48 KiB
Shared memory bank size	4 KB

architecture performs at best with problems that have a size that is a power of 2, but this condition rarely happens in real life applications, as it is very unlikely that the input dataset may produce a problem with an optimal size. The experiment is conducted on a map size of 72 rows, 82 columns and a weight vector size of 58 elements, using a random number generator (in order to test the software on a sequence as generic as possible). Figure 17 shows that performance enhancement is almost unchanged. Results demonstrate that using GPU to solve some problems, in this case a training of a not-fully parallelizable neural network, can produce large improvements.



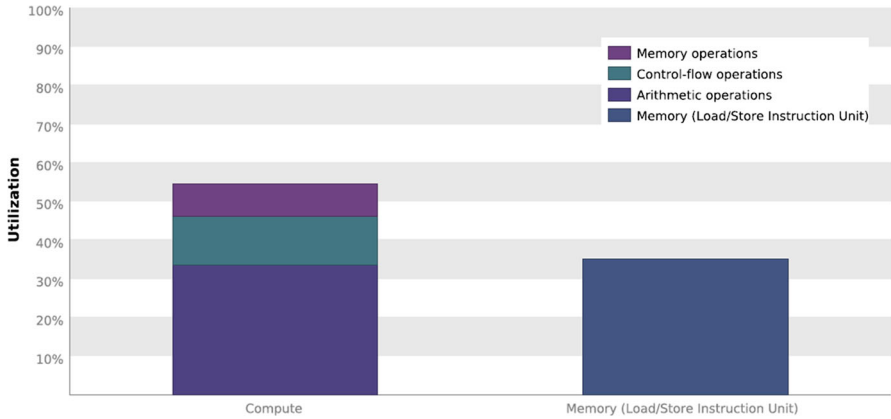


Fig. 15 `findBMUDistances` kernel report analysis

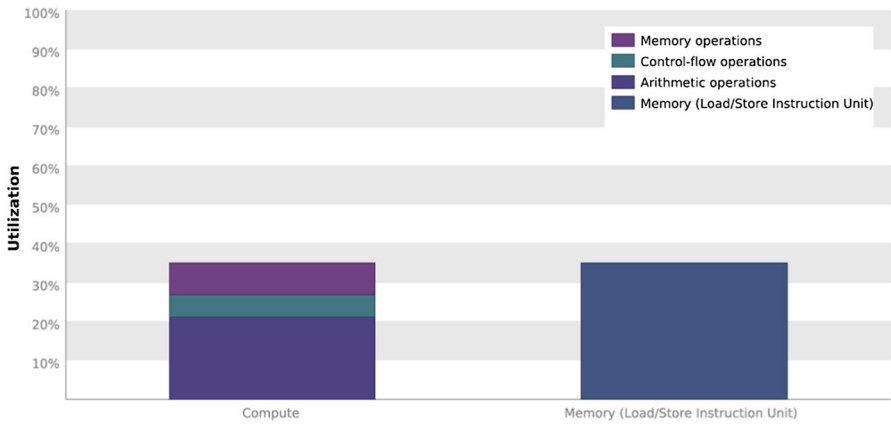
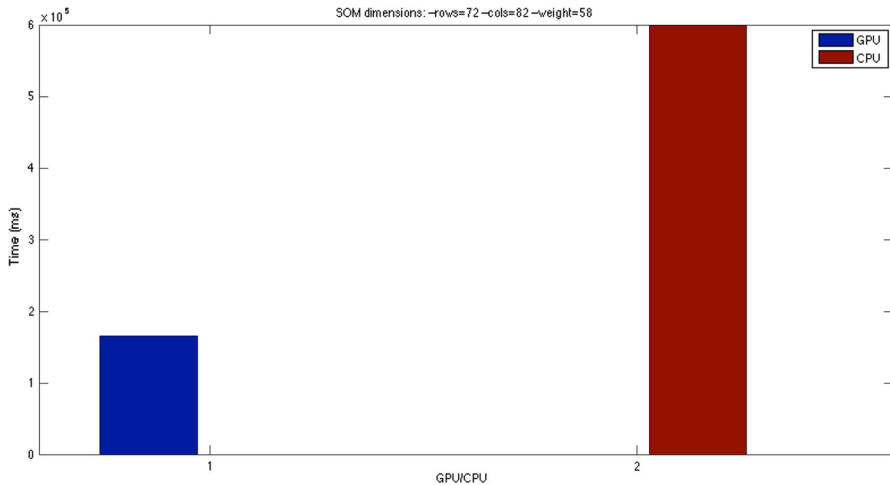


Fig. 16 `adaptation` kernel report analysis

## 5 Conclusions

In this work we proposed a parallel implementation for a machine learning algorithm based on SOM. Our software exploit the computational power of GPU-CUDA and uses the optimized library cuBLAS, provided by nVIDIA for linear algebra operations. The parallel strategy implemented provides an alternative parallelization of the training phase, that is the bottleneck for standard SOM, based on a simultaneous work of threads, significantly reducing synchronization and waiting time. The results demonstrate very interesting improvements and a significant speed-up of GPU over CPU versions.



**Fig. 17** Performance in case the map size is not a power of 2, for GPU (in blue) and CPU (in red) (Color figure online)

## References

1. Alahakoon, D., Halgamuge, S.K., Srinivasan, B.: Dynamic self-organizing maps with controlled growth for knowledge discovery. *Trans. Neural Netw.* **11**(3), 601–614 (2000)
2. Astel, A., Tsakovski, S., Barbieri, P., Simeonov, V.: Comparison of self-organizing maps classification approach with cluster and principal components analysis for large environmental data sets. *Water Res.* **41**(19), 4566–4578 (2007)
3. Bauer, H.U., Pawelzik, K.: Quantifying the neighborhood preservation of self-organizing feature maps. *IEEE Trans. Neural Netw.* **3**(4), 570–579 (1992)
4. Chen, H., Schuffels, C., Orwig, R.: Internet categorization and search: a self-organizing approach. *J. Vis. Commun. Image Represent.* **7**(1), 88–102 (1996)
5. Chianese, A., Marulli, F., Moscato, V., Piccialli, F.: A “smart” multimedia guide for indoor contextual navigation in Cultural Heritage applications. In: 2013 International Conference on Indoor Positioning and Indoor Navigation, IPIN 2013 (2013)
6. Chianese, A., Piccialli, F., Riccio, G.: Designing a smart multisensor framework based on beaglebone black board. *Lecture Notes in Electrical Engineering*, vol. 330, pp. 391–397 (2015)
7. Chianese, A., Piccialli, F.: SmaCH: a framework for smart cultural heritage spaces. In: Proceedings—10th International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2014, pp. 477–2015 (2015). <https://doi.org/10.1109/SITIS.2014.16>
8. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: Reconstruction of implicit curves and surfaces via RBF interpolation *Appl. Numer. Math.* (2016, in press). <https://doi.org/10.1016/j.apnum.2016.10.016>
9. Cuomo, S., Galletti, A., Marcellino, L.: A GPU algorithm in a distributed computing system for 3D MRI denoising. In: Xhafa, F., Barolli, L., Messina, F., Ogilla, M.R. (eds.) 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Krakow, Poland, November 4–6, 2015, pp. 557–562 (2015)
10. Cuomo, S., Galletti, A., Giunta, G., Marcellino, L.: A novel triangle-based method for scattered data interpolation. *Appl. Math. Sci.* **8**(134), 67176724 (2014)
11. Cuomo, S., Michele, P.D., Piccialli, F., Galletti, A., Jung, J.E.: IoT-based collaborative reputation system for associating visitors and artworks in a cultural scenario. *Expert Syst. Appl.* **79**, 101–111 (2017)
12. D’Amore, L., Marcellino, L., Mele, V., Romano, D.: Deconvolution of 3D fluorescence microscopy images using graphics processing units. In: Wyrzykowski, R., et al. (eds.) Proceedings of Conference PPAM2011—International Conference on Parallel Processing and Applied Mathematics, PPAM 2011, Part I, LNCS 7203, pp. 690–699. Springer, Berlin (2012)

13. González, F.A., Dasgupta, D.: Anomaly detection using real-valued negative selection. *Genet. Program Evol. Mach.* **4**(4), 383–403 (2003)
14. Harris, M.: *Optimizing Parallel Reduction in CUDA*, presentation packaged with CUDA Toolkit, NVIDIA Corporation (2007)
15. Hong, M., Jung, J.J., Piccialli, F., Chianese, A.: Social recommendation service for cultural heritage. *Pers. Ubiquit. Comput.* **21**(2), 191–201 (2017)
16. Kalteh, A., Hjorth, P., Berndtsson, R.: Review of the self-organizing map (SOM) approach in water resources: analysis, modelling and application. *Environ. Model. Softw.* **23**(7), 835–845 (2008)
17. Kangas, J.A., Kohonen, T.K., Laaksonen, J.T.: Variants of self-organizing maps. *Trans. Neural Netw.* **1**(1), 93–99 (1990)
18. Kiviluoto, K.: Topology preservation in self-organizing maps. In: *IEEE International Conference on Neural Networks*, vol. 1, pp. 294–299 (1996)
19. Kohonen, T.: Self-organized formation of topologically correct feature maps. *Biol. Cybern.* **43**(1), 5969 (1982)
20. Kohonen, T.: The self-organizing map. *Proc. IEEE* **78**, 1464–1480 (1990)
21. Kohonen, T.: The self-organizing map. *Neurocomputing* **21**(13), 1–6 (1998)
22. Kohonen, T., Somervuo, P.: How to make large self-organizing maps for nonvectorial data. *Neural Netw.* **15**(8–9), 945–952 (2002)
23. Kohonen, T., Oja, E., Simula, O., Visa, A., Kangas, J.: Engineering applications of the self-organizing map. *Proc. IEEE* **84**(10), 1358–1384 (1996)
24. Kohonen, T., Kaski, S., Lagus, K., Salojärvi, J., Honkela, J., Paatero, V., Saarela, A.: Self organization of a massive document collection. *Trans. Neural Netw.* **11**(3), 574–585 (2000)
25. Moraes, F.C., Botelho, S.C., Filho, N.D., Gaya, J.F.O.: Parallel high dimensional self organizing maps using CUDA. In: *Robotics Symposium and Latin American Robotics Symposium (SBR-LARS)*, 2012 Brazilian, Fortaleza, 2012, pp. 302–306. <https://doi.org/10.1109/SBR-LARS.2012.56>
26. Moraes, F., Botelho, S., Filho, N., Gaya, J.: Parallel high dimensional self organizing maps using CUDA, pp 302–306 (2012)
27. Neelima, B., Raghavendra, P.S.: Recent trends in software and hardware for GPGPU computing: a comprehensive survey. In: *2010 5th International Conference on Industrial and Information Systems*, pp. 319–324 (2010)
28. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. *Comput. Graph. Forum* **26**(1), 80–113 (2007)
29. Plato, J., Gajdo, P.: Large data real-time classification with non-negative matrix factorization and self-organizing maps on GPU, pp. 176–181 (2010)
30. Richardson, T., Winer, E.: Extending parallelization of the self-organizing map by combining data and network partitioned methods. *Adv. Eng. Softw.* **88**, 1–7 (2015)
31. Sharma, K.: High performance GPU based optimized feature matching for computer vision applications. *Optik Int. J. Light Electron Opt.* **127**(3), 1153–1159 (2016)
32. Shiralkar, M., Schalkoff, R.: A self-organization based optical flow estimator with GPU implementation. *Mach. Vis. Appl.* **23**(6), 1229–1242 (2012)
33. Song, T., Kerong, B., Liye, T., Zhang, L.: Combination of SOM and RBF based on incremental learning for acoustic fault identification of underwater vehicles. In: *Image and Signal Processing, 2008. CISP '08: 27–30 May 2008. IEEE, Washington (2008)*. <https://doi.org/10.1109/CISP.2008.418>
34. Strong, G., Gong, M.: Similarity-based image organization and browsing using multi-resolution self-organizing map. *Image Vis. Comput.* **29**(11), 774–786 (2011)
35. Tan, X., Chen, S., Zhou, Z., Zhang, F.: Recognizing partially occluded, expression variant faces from single training image per person with SOM and soft k-nn ensemble. *IEEE Trans. Neural Netw.* **16**(4), 875–886 (2005)
36. Vesanto, J., Alhoniemi, E.: Clustering of the self-organizing map. *Trans. Neural Netw.* **11**(3), 586–600 (2000)
37. Wang, H., Mansouri, A., Crput, J.C., Ruichek, Y.: Massively parallel cellular matrix model for superpixel adaptive segmentation map. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9414, pp. 325–336 (2015)
38. Wang, H.F., Chen, Q.K.: General purpose computing of graphics processing unit: a survey. *Chin. J. Comput.* **36**(4), 757–772 (2013)
39. Wittek, P., Darnyi, S.: Accelerating text mining workloads in a mapreduce-based distributed GPU environment. *J. Parallel Distrib. Comput.* **73**(2), 198–206 (2013)

40. Xiao, Y., Leung, C., Ho, T.Y., Lam, P.M.: A GPU implementation for LBG and SOM training. *Neural Comput. Appl.* **20**(7), 1035–1042 (2011)
41. Xiao, Y., Feng, R.B., Han, Z.F., Leung, C.S.: GPU accelerated self-organizing map for high dimensional data. *Neural Process. Lett.* **41**(3), 341–355 (2015)
42. Zhang, N., Wang, H., Creput, J.C., Moreau, J., Ruichek, Y.: Cellular GPU model for structured mesh generation and its application to the stereo-matching disparity map, pp. 53–60 (2013)
43. Zhongwen, L., Hongzhi, L., Zhengping, Y., Xincui, W.: Self-organizing maps computing on graphic process unit, pp 557–562 (2007)