

PARALLELIZATION OF VECTORIZED SELF-ORGANIZING MAPS IN  
HARDWARE ACCELERATOR ARCHITECTURES

BY

OMAR X. RIVERA MORALES

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY  
IN  
COMPUTER SCIENCE AND STATISTICS

UNIVERSITY OF RHODE ISLAND

2022

DOCTOR OF PHILOSOPHY DISSERTATION  
OF  
OMAR X. RIVERA MORALES

APPROVED:

Dissertation Committee:

Major Professor Lutz Hamel

Noah M. Daniels

Resit Sendag

Brenton Deboef

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2022

## ABSTRACT

This dissertation presents the culmination of research performed over six years into developing a parallel and stochastic implementation to the University of Rhode Island's (URI) Computer Science Department Vectorized Self Organizing Maps (VSOM) algorithm. The Parallel VSOM (Par-VSOM) and the High-Level Synthesis VSOM (HLS-VSOM) algorithms are inspired by ideas from tensor algebra and are implemented using parallel kernels and vectorization in modern hardware accelerators.

The map quality generated by the algorithm is of significant importance since higher quality maps provide in-depth knowledge that allows the researcher to identify clusters of information within the datasets. Furthermore, of importance is developing a more efficient and scalable parallel solution, such that it can be executed in newer hardware accelerator architectures. The URI Computer Science Department addressed part of these challenges, leading to its Vectorized Self-Organizing Maps Central Processing Unit (CPU) solution. The VSOM CPU solution was the first vectorized SOM algorithm with sufficient processing throughput to execute the algorithm 60 times faster than Kohone's iterative algorithm. In addition, the VSOM produced quality maps that matched the Kohone's SOM and outperformed the quality of the maps produced by the BatchSOM.

Due to the significant results achieved with the VSOM and the algorithm's vectorization nature, we decided to use the VSOM as the starting point for our proposed algorithms. Furthermore, the state of the art hardware accelerators offer hardware vectorization capability and serve as the perfect environment to improve the previous speed-up gains obtained in CPUs.

The successor to the VSOM CPU-based algorithm has further pushed the limit of state of the art by providing a Graphical Processor Unit (GPU) parallel solution

that has undergone testing in the Amazon Web Service (AWS) cloud. The GPU solution has generated the same map quality as the VSOM CPU-based solution and provides scalable speedup enhancements over the original Kohone's SOM algorithm and the VSOM CPU implementations using large maps. The obtained scalable speedup made the GPU solution URI's fastest for the most optimal solution for larger maps. More importantly, the GPU algorithm provides a roadmap for a higher-performance algorithm hosted in Field Programmable Gate Array (FPGA).

URI's successor to the GPU Par-VSOM algorithm provides an embedded accelerator architecture solution in an FPGA environment. The FPGA experimental results demonstrate that we are not sacrificing map accuracy for performance. The FPGA solution provides a speedup enhancement over the VSOM CPU and the Kohone's SOM algorithm implementation with maps and datasets with the same dimensionality constraints. In addition, compared to GPU implementations, the HLS-VSOM outperforms the SOM GPU variants by two or more orders of magnitude.

Two schools of thought clearly stand out as part of our literary search of other groups performing state-of-the-art parallel SOM solutions: network partitioning and data partition methodology. The network partitioning strategy separated the maps in multiple section to obtain some level of parallelism. To accomplish this, this method employs separated threads to calculate the winning neurons updates of map partitions. This result in faster execution but the separation of the maps in subsections adds complexity to the analysis of the data and may result in a lower quality of the entire merge maps. The data partitioning methodology is a more common approach, where the data is distributed among the individual threads for faster parallel execution. A very popular variant of the methodology is the BatchSOM, which executes the best matching unit (BMU) part of the algorithm

in parallel is unable to preserve a consistent map quality. This approach provides a good option for parallelism but does not allow for a complete parallel solution as the PAR-VSOM and HLS-VSOM variants.

The URI's Par-VSOM and HLS-VSOM solutions have advantages over other state-of-the-art parallel SOM algorithms, the most notable advantage being their high performance. The higher performance can be attributed to the use of fully vectorized data structures, neighborhood caching, asynchronous memory speed gains, pipelining, loop unrolling, and array partitioning. As a result, URI's parallel algorithm solutions are leading the way toward highly optimized SOM algorithms, thereby providing a high-performance alternative to SOM algorithm.

## ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Lutz Hamel for his advise, cooperation, and most importantly, for sharing his wisdom and patience throughout the research milestones necessary to materialize this dissertation into reality. His astounding ability to project significant ideas into academic papers is something I admire and hope to achieve in the future. Furthermore, I will always appreciate his willingness to include me as part of his research team and guide me in generating outstanding Ph.D. research and writing quality academic papers.

In addition to Dr. Hamel, I would like to thank the rest of my dissertation committee: Dr. Noah Daniels, Dr. Resit Sendag and Dr. Gordon Dash for reviewing the content within this dissertation, for taking the time to serve and participate in the comprehensive examination and dissertation defense.

Also, I am very grateful with NUWC team for proofreading my papers, making recommendations, and providing a long-term grant to focus on this research efforts.

Finally yet importantly, I would like to thank my wife, Mariana, and my daughter, Elena. I want to apologize for some absence during the last six years; I thank you both for your understanding, patience, and support in helping me reach my educational goals.

## PREFACE

This dissertation is written in the manuscript format. It consists of two manuscripts organized as follows:

Manuscript 1:

Omar X. Rivera Morales, Lutz Hamel “Par-VSOM:Parallel and Stochastic Self-Organizing Map Training Algorithm”, will be submitted to 14th International Conference on Neural Computation Theory and Applications NCTA 2022 : 14th International Joint Conference on Computational Intelligence (IJCCI '22), Valletta, Malta, 2022.

Manuscript 2:

Omar X. Rivera Morales, Lutz Hamel, “High-Level Synthesis Parallelization and Optimization of Vectorized Self-Organizing Maps”, submitted to the 18th International Conference on Data Science (ICDATA'22), Nevada, USA, 2022

## TABLE OF CONTENTS

<b>ABSTRACT</b> . . . . .	ii
<b>ACKNOWLEDGMENTS</b> . . . . .	v
<b>PREFACE</b> . . . . .	vi
<b>TABLE OF CONTENTS</b> . . . . .	vii
<b>MANUSCRIPT</b>	
<b>1 Par-VSOM: Parallel and Stochastic Self-Organizing Map Training Algorithm</b> . . . . .	1
1.1 Abstract . . . . .	2
1.2 Introduction . . . . .	3
1.3 The SOM and VSOM Algorithms . . . . .	4
1.3.1 The SOM and VSOM Competitive Step . . . . .	5
1.3.2 The SOM and VSOM Update Step . . . . .	7
1.4 Related Work . . . . .	9
1.4.1 SOM Parallel Hybrid Methods . . . . .	9
1.4.2 SOM Vectorization . . . . .	10
1.4.3 SOM in Multiple Parallel Architectures . . . . .	10
1.5 Par-VSOM: Parallel Vectorized SOM . . . . .	11
1.5.1 Hardware For Parallel Vectorization . . . . .	11
1.5.2 Par-VSOM Algorithm . . . . .	11
1.5.3 Limitations . . . . .	15
1.6 Experiments . . . . .	16



	<b>Page</b>
1.6.1 Hardware setup . . . . .	16
1.6.2 Par-VSOM setup and Hyper-Parameters . . . . .	16
1.6.3 Results . . . . .	21
1.7 Conclusions . . . . .	25
List of References . . . . .	26
<b>2 High-Level Synthesis Parallelization and Optimization of Vectorized Self-Organizing Map . . . . .</b>	<b>29</b>
2.1 Abstract . . . . .	30
2.2 Introduction . . . . .	31
2.3 High Level Synthesis . . . . .	32
2.4 Vectorization of Self-Organizing Maps . . . . .	34
2.4.1 The SOM and VSOM Competitive Step . . . . .	35
2.4.2 The SOM and VSOM Update Step . . . . .	37
2.5 High-Level Synthesis VSOM . . . . .	38
2.5.1 HLS VSOM Algorithm . . . . .	38
2.5.2 Pipelining and Dataflow . . . . .	39
2.5.3 HLS VSOM Horizontal Unrolling (Vectorization) . . . . .	41
2.5.4 HLS Par-VOM Memory Transformations . . . . .	43
2.5.5 HLS Matrix Reduction with Systolic Arrays . . . . .	44
2.6 Related Work . . . . .	45
2.6.1 Stochastic SOM with FPGA SoC . . . . .	46
2.6.2 A Scalable SOM based on a Sequential Systolic NoC . . . . .	46
2.6.3 High Level Synthesis (HLS) for K-means algorithm . . . . .	47

	Page
2.6.4 High-Performance Computing Applications via High-Level Synthesis . . . . .	47
2.6.5 SOMs in GPUs . . . . .	48
2.7 Experiments . . . . .	48
2.7.1 Hardware setup . . . . .	48
2.7.2 HLS-VSOM setup and Hyper-Parameters . . . . .	49
2.7.3 Results . . . . .	54
2.8 Conclusions . . . . .	56
List of References . . . . .	59

## APPENDIX

<b>A Introduction and review of the problem . . . . .</b>	<b>62</b>
A.1 Introduction . . . . .	62
A.2 Review of the Problem . . . . .	63
A.2.1 The SOM and VSOM Algorithm . . . . .	64
A.2.2 Parallel SOM . . . . .	70
A.2.3 Hardware Architectures for Parallel SOMs . . . . .	73
A.2.4 Parallel Vectorized SOM . . . . .	76
A.2.5 High Level Synthesis for Parallel SOMs . . . . .	77
A.2.6 Systolic Array with HLS . . . . .	78
List of References . . . . .	80
<b>B Methodology and Source code . . . . .</b>	<b>82</b>
B.1 Methodology . . . . .	82
B.1.1 Research Design . . . . .	83

	<b>Page</b>
B.1.2 Data Sets . . . . .	84
B.2 Readme File . . . . .	85
B.3 Source Code . . . . .	91
B.3.1 Par-VSOM Cuda Kernel . . . . .	91
B.3.2 HLS-VSOM . . . . .	103
List of References . . . . .	120
<b>C Conclusion . . . . .</b>	<b>122</b>

## MANUSCRIPT 1

**Par-VSOM: Parallel and Stochastic Self-Organizing Map Training  
Algorithm**

by

<sup>1</sup>Omar X. Rivera Morales, Lutz Hamel will be submitted to 14th International Conference on Neural Computation Theory and Applications NCTA 2022 : 14th International Joint Conference on Computational Intelligence (IJCCI '22), Valletta, Malta, 2022.

---

<sup>1</sup>Omar X. Rivera Morales and Lutz Hamel are with Department of Computer Science and Statistics, University of Rhode Island, Kingston, RI, 02881, Email {oxriveramorales, lutzhamel}@uri.edu.

## 1.1 Abstract

This work proposes Par-VSOM, a novel parallel version of VSOM, a very efficient implementation of stochastic training for self-organizing maps inspired by ideas from tensor algebra. The new algorithm is implemented using parallel kernels on GPU accelerators. It provides performance increases over the original VSOM algorithm, PyTorch Quicksom parallel version, Tensorflow Xpysom parallel variant, as well as Kohonen's classic iterative implementation. Here we develop the algorithm in some detail and then demonstrate its performance on several real-world datasets. We also demonstrate that our new algorithm does not sacrifice map quality for speed using the convergence index quality assessment.

## 1.2 Introduction

The self-organizing map (SOM) is a neural network designed for unsupervised machine learning [1]. The generated maps are powerful data analysis tools applied to diverse areas such as atmospheric science, nuclear physics, pattern recognition, medical diagnosis, computer vision and other data domains [2, 3, 4]. See reference [1] for a more comprehensive literature survey. Here we introduce the Parallel VSOM (Par-VSOM), a parallel implementation of the efficient VSOM algorithm [5]. The novel approach presented here, replaces all iterative constructs of the SOM algorithm with kernels running in a hardware accelerator to perform vector and matrix operations in parallel. The algorithm kernels provide substantial performance increases over Kohonen’s SOM iterative algorithm, the *XpySom*[6], and *Quicksom* [7, 8] parallel BatchSOM implementations.

The training of the SOM is computationally demanding, but a great advantage of SOMs is that the computations can be parallelize with algorithm modifications like in the BatchSOM or using hardware vectorization. Currently, various types of hardware accelerators are easily available, allowing us to process Big-Data [9] datasets using high-performance computers (HPC), Graphical Processing Units (GPU), and Field Programmable Gate Arrays (FPGA)[10, 11]. This research provides an alternative efficient SOM algorithm to accelerate the training of highly complex rectangular maps.

Our experiments demonstrate that our parallel algorithm is better suited for highly computational demanding maps, such as the maps generated with large SOMs. Using a large number of neurons provides a higher resolution clustering of the data and facilitates the pattern recognition during the analysis, as shown in Figure 1. Furthermore, the maps produced by the Par-VSOM are equivalent in quality to the maps produced by the original SOM iterative algorithm. The

current Par-VSOM model is parallel and multi-threaded, and therefore well suited as a replacement for other parallel algorithms to train the self-organizing maps.

The paper is organized as follows: In Section 1.3, we start our discussion with an overview of the SOM and a brief introduction to the VSOM [5] vectorized rules, which can be viewed as an implementation of a competitive learning scheme comprised of a competitive step and an update step with vector and matrix training. The relevant details about related research work are included in Section 2.6. As part of Section 1.5, we develop the Par-VSOM vector-based parallel training and examine the data level parallelisms achievable using vectorized single instruction with multiple data (SIMD) registers and discuss the limitations. Under Section 2.7, we included the study of the performance of our parallel vectorized training implementation by comparing it to various CPU and GPU SOMs variants. Finally, in Section 2.8, we conclude our discussion with a summary of the observations and some future research ideas under consideration.

### 1.3 The SOM and VSOM Algorithms

The origins of the self-organizing maps model can be traced back to the Vector Quantization (VQ) method [1]. The VQ is a signal-approximation algorithm that approximates a finite “codebook” of vectors  $m_i \in R^n, i = 1, 2, \dots, k$  to the distribution of the input data vector  $x \in R^n$ . In the SOM context, the approximated codebook allows us to categorize the nodes and form an “elastic network,” which becomes a meaningful, coordinated map or grid system.

From a computational perspective, the SOM can be described as a mapping of high dimensional input data onto a low dimensional neural network projected as a 2D or three-dimensional (3D) constrained topological map [12]. The mapping is accomplished by assuming that the input data set is a real vector such as  $x_k = [\xi_1, \xi_2, \dots, \xi_n]^T \in R^n$ . The SOM neuronal map can be defined as a model containing

the parametric real vector  $m_i = [u_{i1}, u_{i2}, \dots, u_{in}]^T \in R^n$  associated with the neurons' weights. If we consider the distance between the input vector  $x_k$  and the neuron vector  $m_i$  then we can establish an initial minimum distance relation between the input and the neurons by calculating the Euclidean distances. Then, these distances are used to identify the best matching unit (BMU) index with equation (14).

$$c =_i (||\mathbf{m}_i - \mathbf{x}_k||^2) \quad (1)$$

To define the SOM in terms of matrix and vector operations it is assumed that the map's neurons are stored in a  $n \times d$  matrix  $\mathbf{M}$  where each row  $i$  represents the neuron  $\mathbf{m}_i$  with  $d$  components,

$$\mathbf{M}[i, ] = \mathbf{m}_i = (m_1, \dots, m_d)_i, \quad (2)$$

with  $i = 1, \dots, n$ . The training data  $x$  consists of a set  $D = \{\mathbf{x}_1, \dots, \mathbf{x}_l\}$ . The set can be defined as a  $l \times d$  matrix where each row  $k$  represents the training vector  $\mathbf{x}_k$  with  $d$  components,

$$\mathbf{D}[k, ] = \mathbf{x}_k = (x_1, \dots, x_d)_k, \quad (3)$$

with  $k = 1, \dots, l$ .

Essential details to consider include (1) the dimensionality  $d$  for the input, and (2) the neuron vectors are required to be the same size for well-defined matrix operations.

### 1.3.1 The SOM and VSOM Competitive Step

In the competitive step, we find the BMU for a particular training instance  $\mathbf{x}_k$ . In the classic SOM we use an iterative process to find the BMU using 1. Here the  $i = 1, 2, \dots, n$  represents the index of the neurons in the map and  $\mathbf{m}_i$  represents



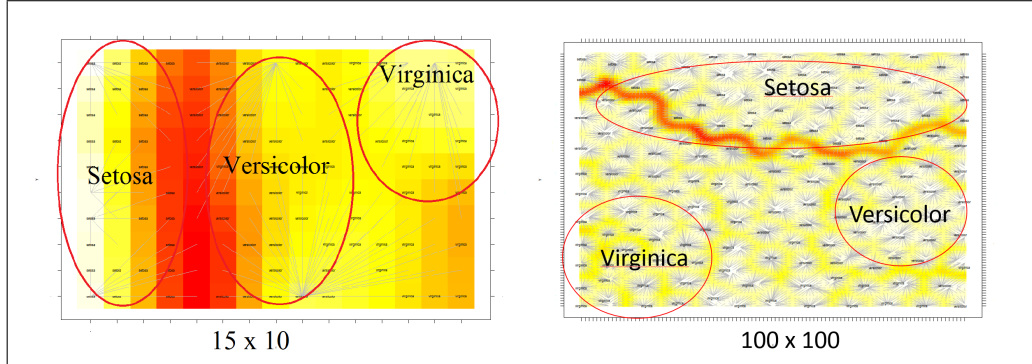


Figure 1: IRIS 15x10 small SOM and IRIS 100x100 large SOM, neuronal heatmaps patterns with different resolutions.

the neuron in index  $i$ . The *argmin* is a function that returns the minimum value and  $c$  contains the index of the BMU.

In the VSOM context this step requires us to calculate the Euclidean distance as a set of vector and matrix operations. These operations find the  $c$  index associated with the neuron with the minimum distance to the training instance. The BMU  $c$  index corresponds to the neuron in the map with the highest resemblance to the particular  $\mathbf{x}_k$  selected for training during the epoch.

The first step to calculate the BMU requires us to compute a matrix  $\mathbf{X}$  to hold a randomly selected training vector. The matrix  $\mathbf{X}$  in equation (4) is defined with a component sizes of  $n \times d$ , where each row is holding the current epoch *training vector*  $\mathbf{x}_k = (x_1, x_2, \dots, x_d)_k$ , which is randomly selected from matrix  $\mathbf{D}$ ,

$$\mathbf{X} = \mathbf{1}^n \otimes \mathbf{x}_k. \quad (4)$$

Here, the symbol  $\otimes$  represents the outer product and  $\mathbf{1}^n$  is a column vector defined as,

$$\mathbf{1}^n = \underbrace{(1, 1, \dots, 1)}_n^T. \quad (5)$$

Since  $\mathbf{1}^n$  is a column vector and  $\mathbf{x}_k$  is a row vector the operation in (4) is well defined. After populating the instance matrix  $\mathbf{X}$  with the duplicated  $\mathbf{x}_k$  values,

equations (6), (7) and (8) are used to compute the square of the Euclidean distances between all the map neurons and the selected input vector,

$$\mathbf{\Delta} = \mathbf{M} - \mathbf{X} \quad (6)$$

$$\mathbf{\Pi} = \mathbf{\Delta} \circ \mathbf{\Delta} \quad (7)$$

$$\mathbf{s} = \mathbf{\Pi} \times \mathbf{1}^d \quad (8)$$

In equation (6) we calculate the difference between the matrices with an element-by-element matrix subtraction. In equation (7) we use the Hadamard product to allow us to calculate the  $\mathbf{\Pi}$  matrix, in this context  $\circ$  represents the element-by-element matrix product and  $\mathbf{X}$ ,  $\mathbf{M}$ ,  $\mathbf{\Delta}$  and  $\mathbf{\Pi}$  are all  $n \times d$  matrices.

Lastly, in equation (8) we use a ‘row sum’ matrix reduction to compute the vector  $\mathbf{s}$  of size  $n$ . Here,  $\mathbf{1}^d$  is a column vector similar to (5) with the dimensionality defined by the value of  $d$ .

### 1.3.2 The SOM and VSOM Update Step

In the classic stochastic SOM, after completing the BMU calculations, the updates to the neuronal weights are accomplished using the training instance  $x_k$  to influence the best matching neuron and its surrounding neighborhood.

$$\mathbf{m}_i \leftarrow \mathbf{m}_i - \eta(\mathbf{m}_i - \mathbf{x}_k)h(c, i) \quad (9)$$

The weights update step in equation (9), affects every neuron inside the neighborhood radius of influence. Here, the learning rate  $\eta$  serves as a scaling factor between 0 and 1. The  $h(c, i)$  acts as the loss function, where  $i = 0, 1, \dots, n$  and it can be defined as,

$$h(c, i) = \begin{cases} 1 & \text{if } i \in \Gamma(c), \\ 0 & \text{otherwise,} \end{cases} \quad (10)$$

where  $\Gamma(c)$  is the neighborhood of the best matching neuron  $\mathbf{m}_c$  with  $c \in \Gamma(c)$ . In the classic SOM, the learning factor and the loss function both decreased monotonically over time [1].

In the VSOM, the update step for all the neurons in the map is accomplished with matrix operations and is defined as,

$$\mathbf{M} \leftarrow \mathbf{M} - \eta \mathbf{\Delta} \circ \mathbf{\Gamma}_c. \quad (11)$$

Here,  $\eta$  is the learning rate,  $\mathbf{\Delta}$  contains the calculations of the difference between the neurons and the selected training instance as computed in (6), and the symbol  $\circ$  represents the Hadamard product. Similarly to the SOM, in the VSOM, the learning rate  $\eta$  is linearly reduced as epochs increase.

However, our experimental results demonstrate that a constant learning rate  $\eta$  generates higher quality convergence indexes in large map instances. Initially, the update rule for each best matching neuron has a very large radius of influence and includes all the neurons on the map. After multiple training epochs, the neighborhood radius around the BMU gradually shrinks to the point that the field of influence only includes the best matching neuron  $\mathbf{m}_c$  as shown in (12).

$$\Gamma(c)|_{t \gg 0} = \{c\}. \quad (12)$$

The competitive and the update steps are computed during each epoch using the randomly selected training instances until some convergence criterion is fulfilled. After reaching a maximum convergence, every neuron will be assigned to an specific data point forming clusters in the grid and preserving the neighborhood topology as shown in Figure 2.

Algorithm 1 and 2 summarizes the matrix and vector operations required for the parallel Par-VSOM training. For a more detailed explanation of the SOM and VSOM algorithms, see reference [5].

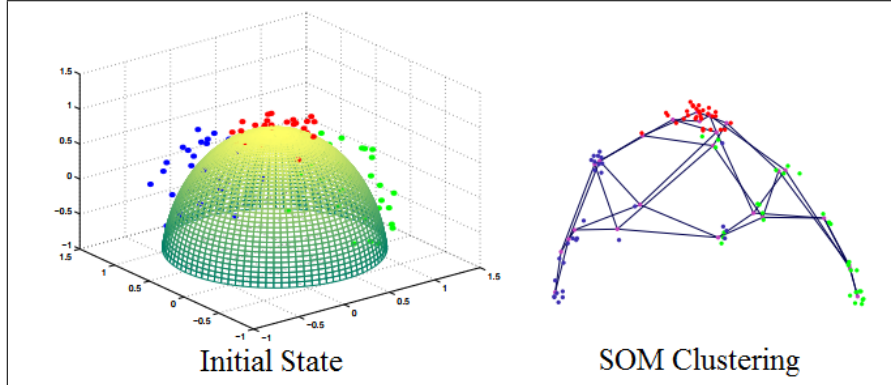


Figure 2: SOM preserving the neighborhood topology in 3D space [12].

## 1.4 Related Work

In this section, we look at prior work related to parallel SOM algorithms and applications to pattern recognition. Recent parallel self-organizing maps research has demonstrated promising improvements using various parallel methods. Some of the methodology mentioned in current scientific publications on this topic include: combining data and network partitioning techniques [10, 13], exploiting cache effects [14], using map-reduce programming paradigm [?, 15, 16], replacing the SOM iterative construct with vector and matrix operations [5], and using various types of accelerated architectures for parallelism [17, 18, 11, 19, 20]. In addition, recent publications demonstrate how to utilize SOM as a pattern recognition tool [21, 22, 23]. In general, recent research publications share similar goals such as: finding new applications, improving optimal performance and increasing speed-up using different SOM approaches.

### 1.4.1 SOM Parallel Hybrid Methods

The combination of data and network partitioned parallel methods develop by Richardson et al. [10] splits up the map to compute the best matching calculation and nodes update on separate threads. This hybrid methodology also divides the data amongst individual threads for data partition parallelism. As part of their

research findings, they concluded that parallelizing the classic SOM algorithm using such techniques in a GPU can save computation time and increase the speed-up by nearly 15X in maps with 10,000 points and 5 dimensions. A similar method was proposed by Silva et al. [13], achieving a performance increases of 1.27X training large maps on a small HPC cluster.

#### 1.4.2 SOM Vectorization

The VSOM [5] by Hamel replaced all the iterative constructs of the standard stochastic SOM algorithm with vector and matrices operations. The VSOM implementation resulted in a performance increase of up to 60X faster after running 10000 iterations in a 25 X 20 map. Since the VSOM seems to be offering the highest speed-up increase of all the current SOM research publications, our research is focus on the parallelization of the VSOM algorithm and its implementation in hardware accelerators.

#### 1.4.3 SOM in Multiple Parallel Architectures

Among the SOM parallel approaches previously discussed, no too many offer an available open source repository to validate the research findings or continue with further investigations. In this paper, we decided to compare our proposed parallel implementation with some of the widely available parallel SOM projects packages. As part of the GPU comparisons we utilize, *Quicksom* [8] which offers a parallel GPU Batch-SOM algorithm implemented using the Python PyTorch framework and speed-ups results showed at least a 20 speed-up over the CPU version using bioinformatics datasets [7]. In addition, we also included a comparison with *XpySom* [6] a parallel Batch-SOM variant implemented using the Google Tensorflow 2.0 framework and Python Numpy library. The *XpySom* package is based on the Minisom[24], a non-parallel, minimalistic and Numpy based widely

known implementation of the SOM. The *XpySom* research paper [19] indicates their parallel variants outperform the popular SOM GPU package *Somoclu* by two and three orders of magnitude.

## 1.5 Par-VSOM: Parallel Vectorized SOM

### 1.5.1 Hardware For Parallel Vectorization

Our novel parallel implementation is based on the VSOM algorithm proposed by Hamel [5]. On the VSOM, the stochastic SOM training is redefined to execute as a set of vector and matrix operations. Since all the matrix data elements are independent of each other, they can be executed as coarse-grained “embarrassingly parallel” tasks to exploit multiple hardware threads (or cores) available in the devices [25]. In the VSOM context, the vectorization of the calculations can be implemented as vector instructions, which are also known as SIMD instructions and are a form of Data-Level Parallelism. These vector instructions apply the same operation over multiple data elements (like integers and floating-point values) concurrently, given that these items are stored contiguously in vector/SIMD registers [26]. In modern Intel and AMD CPU architectures, these vector instructions are known as Advance Vector Extensions (AVX), AVX2 and AVX-512 instruction sets.

In contrast, the GPUs with their substantial amount of nodes allows for the creation of thousands of threads to perform vector calculations simultaneously. Furthermore, the current NVIDIA GPUs can access their memory much faster when accessing adjacent data concurrently. This is optimized when groups of 32 GPU threads or warps do the request simultaneously, causing “memory coalescing” [27].

### 1.5.2 Par-VSOM Algorithm

In the classic SOM with iterative operations, the operations per column are solved sequentially. This serial dependency results in high overhead and additional

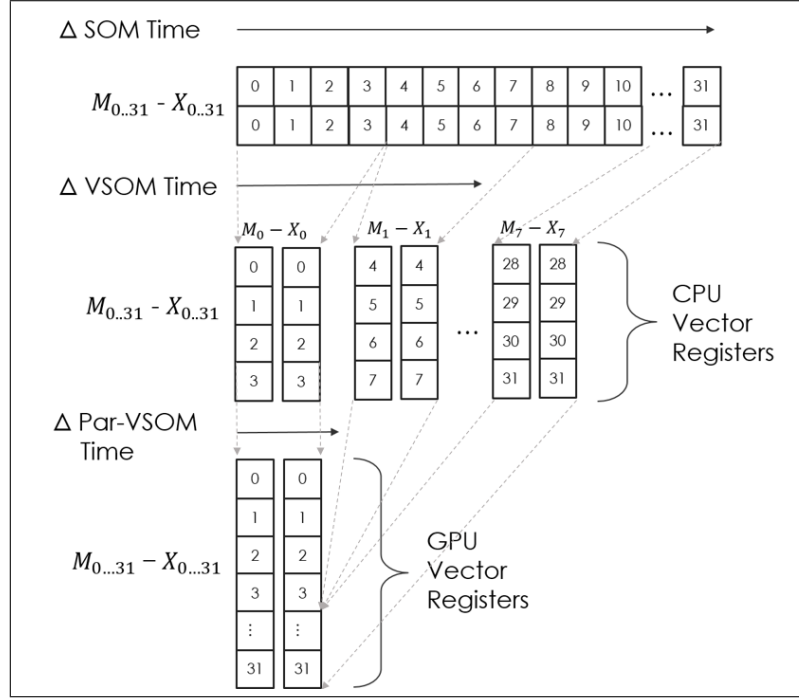


Figure 3: The time comparison of  $\Delta$  calculation during the competitive step for SOM, VSOM, and PAR-VSOM demonstrate modern architecture advances in vectorization capability increases the primitive operations’ overall speed-up performance.

latency during every training epoch. Conversely, the VSOM vector and matrix operations are vectorized by the compiler, and they are executed in the CPU as vector operations. To illustrate, in a data set with 32 instances, the VSOM using vectorized operations will need to execute a total of four “*minus*” operations to compute a  $\Delta$  matrix entirely. Using the VSOM vectorization, the  $\Delta$  matrix “*minus*” operation can be completed with a speed-up increase of 4X compared to the SOM, as illustrated in Figure 3.

In the Par-VSOM, the vector and matrix operations of the original VSOM are replaced with parallel computational kernels executing in hardware accelerators architecture. The parallel kernels manipulate the matrices columns in a unified vector  $V_u$  as shown in equation (13). In the kernel, the matrices are expressed as tuples of column vectors and encapsulated into one unifying vector. Based on

the data of our example in Figure 3, the unifying vector technique will result in executing the 32 elements operation in one single vectorized operation, providing a performance increase of 128x.

$$\mathbf{V}_{\mathbf{u}}[i * n] = (t_1, \dots, t_n)_1 \cup (t_1, \dots, t_n)_2 \dots \cup (t_1, \dots, t_n)_i \quad (13)$$

In the unifying vector equation (13), we have shown how the matrices can be express in terms of tuples. In the Par-SOM algorithm (1) and (2), we are assuming all the matrices of the VSOM are implemented as a data structure consisting of multiple tuples  $(t_1, t_2, \dots, t_n)$  where each column is represented by the tuple  $(t_n)_i$  with  $i$  representing the dimensionality of the matrix and  $n$  the number of instances. This technique allows the data-level parallelism to occur by executing all the matrix operations as optimized vector operations inside the  $\Phi$  kernels as presented in algorithm 1 and 2.

In our GPU implementation, we decided to use CUDA Thrust. Considering that the Par-VSOM is a parallel and vectorized implementation of the VSOM algorithm, the Thrust template is an ideal candidate due to the vast number of vector functions available. In addition, Thrust manages all the CUDA kernel initialization, memory transfers and allocation in the background, and provides highly optimized libraries for vector operations [28].

Since most of the VSOM algorithm consists of matrix operations, we utilized Thrust specialized transformation and reduction functions to process the matrices as vectors. In the case of a matrix with three columns, storing 3d points as an array of float3 in CUDA is generally a bad idea, since array accesses are not properly coalesced [28]. To address this memory access issue, the number of rows  $n$  was used as a delimiter to identify the beginning and the end of each column in the unifying vector  $Vu$ . The column-wise encapsulation of the matrix transforms the



---

**Algorithm 1** The Par-VSOM training algorithm.

---

```

1: Given:
2:    $D \leftarrow \{\text{training instances, a } l \times d \text{ matrix}\}$ 
3:    $M \leftarrow \{\text{neurons, a } n \times d \text{ vector of tuples}\}$ 
4:    $\eta \leftarrow \{\text{learning rate } 0 < \eta < 1\}$ 
5:    $\Gamma(c) \leftarrow \{\text{neighborhood function for some neuron } c\}$ 
6:    $\text{minIndex}(s) \leftarrow \{\text{func, returns location of min. val in } s\}$ 
7:    $\Phi \leftarrow \{\text{Vectorized kernel operation, with all matrices}$ 
8:      $\text{columns unified as tuples in a single column vector.}\}$ 

9: Repeat:
10: /**Select a matrix training instance as vector
11: for some } k = 1, \dots, l \text{ and } f = 1, \dots, d : ***/
12:
13:    $x_k \leftarrow D[k][1] \cup D[k][2] \dots \cup D[k][f]$ 
14:
15: /**Find the winning neuron using vectorized kernels ***/
16:    $X \leftarrow \Phi_x(1^n \otimes x_k)$ 
17:    $\Delta \leftarrow \Phi_\Delta(M - X)$ 
18:    $\Pi \leftarrow \Phi_\Pi(\Delta \circ \Delta)$ 
19: /**Sum of vector subsections (rowsum) ***/
20:    $s \leftarrow \Phi_s(\Pi_{1\dots(n*1)} + \Pi_{(n*1)\dots(n*2)} +$ 
21:      $\dots \Pi_{(n*(d-1)\dots(n*d)})$ 
22:    $c = \text{minIndex}(s)$ 
23:
24: /**Update neighborhood with vector operations ***/
25:    $\Gamma_c \leftarrow \Phi_\Gamma(\Gamma(c))$ 
26:    $M_{new} \leftarrow \Phi_{M_{new}}(M_{current} - \eta \Delta \circ \Gamma_c)$ 
27: done
28: return  $M_{new}$ 

```

---

three-dimensional columns in to one  $Vu$  vector. This allows coalesced memory access and faster operation execution.

One of the important differences between the original VSOM algorithm and the Par-VSOM algorithm, is the data structure manipulation during the selection of the  $D$  matrix random training instance algorithm 1 (line 13). Here, the training computation transforms the selection into an  $X_k$  vector that includes all the matrix  $D$  columns and allows us to find the BMU using vector operations. To be able to use the optimized “minIndex( $s$ )” function in line 22, we reduced the  $\Pi$  vector with

---

**Algorithm 2** The Par-VSOM Neighborhood Function  $\Gamma$ .

---

```

1: given:
2:    $c \leftarrow \{\text{index of winning neuron}\}$ 
3:    $n \leftarrow \{\text{the number of neurons on the map}\}$ 
4:    $nsize \leftarrow \{\text{neighborhood radius}\}$ 
5:    $\mathbf{P} \leftarrow \{\text{an } n \times 2 \text{ vector with } \mathbf{p}_i = \mathbf{P}[i,] = (x_i, y_i)\}$ 
6:    $\mathbf{1}^n \leftarrow \{\text{constant column vector with value 1}\}$ 
7:    $\mathbf{0}^n \leftarrow \{\text{constant column vector with value 0}\}$ 
8:    $\Phi \leftarrow \text{Vectorized kernel operation, with all matrices}$ 
9:   columns unified as tuples in a single column vector.
10:   $\mathbf{x} \leftarrow \{\text{x values in vector first section: } 1, \dots, (\frac{n}{2} - 1)\}$ 
11:   $\mathbf{y} \leftarrow \{\text{y values in vector second section: } \frac{n}{2}, \dots, (n \times 2)\}$ 
12:
13:   $P_c \leftarrow \Phi_{pc}(P[c,])$ 
14:   $\mathbf{C} \leftarrow \Phi_C(\mathbf{1}^n \otimes \mathbf{p}_c)$ 
15:   $\mathbf{\Delta} \leftarrow \Phi_{\Delta}(\mathbf{P} - \mathbf{C})$ 
16:   $\mathbf{\Pi} \leftarrow \Phi_{\Pi}(\mathbf{\Delta} \circ \mathbf{\Delta})$ 
17:  /**Perform rowsum with vector subsections
18:   $\mathbf{d} \leftarrow \Phi_d(\mathbf{\Pi}_x + \mathbf{\Pi}_y)$ 
19:   $\mathbf{hood} \leftarrow \Phi_{hood}(\text{ifelse}(\mathbf{d} < (nsize \times 1.5)^2, \mathbf{1}^n, \mathbf{0}^n))$ 
20:  return hood

```

---

length  $n * d$  into a vector of length  $n$ , using operations equivalent to a rowsum across  $d$  dimensions in line 20.

Similarly, the Par-VSOM neighborhood Function  $\Gamma$  in algorithm 2, emulates the rowsum operations of algorithm 1 in line 18 by utilizing the vector elements representing the x and y columns accordingly and returns one vector that includes the distances of the neurons in the grid. In lines 19 to 20 using the computed distances, the vector neighborhood determination is performed and return a **hood** vector that activates the neurons considered to be part of the neighborhood by flipping to “1” their corresponding neurons index.

### 1.5.3 Limitations

#### Large Computational Workloads

The Par-VSOM is recommended for clustering problems requiring high computational workloads. To obtain our experimental results, we tested with multiple

datasets and various map sizes. The results demonstrated the Par-VSOM is not suitable for small maps, low-dimensional datasets, or minimal computational workloads. Here, we assume the users will have a GPU hardware accelerator available as part of their setup.

In general, Big data and other extensive datasets analysis requires generating large neuronal maps as part of the pattern analysis and clusters visualizations. The GPUs have become one of the default tools to process high complexity problems and are easily accessible in cloud environments, but we are aware that not everyone may have access to one.

## 1.6 Experiments

### 1.6.1 Hardware setup

All the Par-VSOM, *Xpysom* and *Quicksom* parallel experiments were performed using the Amazon AWS cloud service instances with Linux and Deep Learning Amazon Machine Images (AMI). The sequential CPU experimental setting included an Intel I7-7700K running at 4.20 GHz/ 4.50GHz turbo with four cores and capable of executing eight threads. The GPU tests were performed in an AWS P3.2xlarge with 18 virtual Intel Xeon E5 2686 CPU operating at 2.7 GHz/ 3.0 GHz turbo and an NVIDIA Tesla V100. The Tesla V100 contains 5120 NVIDIA Cuda cores with 16 Gb of HBM2 memory. The Tesla V100 memory clock setting was 877 Mhz with memory graphics clocked at 1530 Mhz.

### 1.6.2 Par-VSOM setup and Hyper-Parameters

The experimental setup utilized the default values of the SOM and VSOM *Popsom* [29]. For the *Quicksom*[7] and *Xpysom*[19] BatchSOM packages, we maintained the learning rate constant to obtain higher convergence indexes and tune the hyper-parameters as defined in Table 4.

Table 1: Par-VSOM Hyper-Parameters.

<b>**Hyper-Parameters**</b>	<b>**Values**</b>
Training Iterations	$1 \times 10^0 \dots 1 \times 10^5$
Learning Rate $\eta$	0.7
Neighborhood Radius	Bubble, Gaussian(for <i>Quicksom</i> )
Map sizes	15x10, 150x100, 200x150
Datasets	Iris, Epil, WDBC

Table 2: Times and Speed-up gains of the Par-VSOM for different training algorithms using a  $200 \times 150$  map.

iter	Time SOM(s)	Time VSOM(s)	Time CPU	Time GPU	Time P-VSOM(s)	Time GPU	Time Xpysom(s)	Time CPU-GPU	Time QuickSom(s)	Time CPU-GPU	Time PyTorch	Speed-up SOM	Speed-up Par-VSOM/ VSOM	Speed-up Par-VSOM/ Xpysom	Speed-up Par-VSOM/ QuickSom
1	1.148	0.035	0.027	0.301	0.257	*** Iris D=4***									
10	1.350	0.046	0.029	0.319	0.257	42.5	1.3	11.1	9.5						
100	2.362	0.067	0.049	0.414	0.434	46.6	1.6	11.0	8.9						
1000	13.447	0.324	0.235	1.408	2.32	48.2	1.4	8.4	8.9						
10000	124.011	2.756	1.925	10.742	21.456	57.2	1.4	6.0	9.9						
100000	1228.811	26.210	18.275	110.900	212.791	64.4	1.4	5.6	11.1						
						67.2	1.4	6.1	11.6	*** Epil D=8***					
1	1.831	0.053	0.046	0.300	0.262	39.8	1.2	6.5	5.7						
10	1.949	0.058	0.049	0.313	0.259	39.8	1.2	6.9	5.3						
100	3.125	0.108	0.072	0.412	0.643	43.4	1.5	5.7	8.9						
1000	14.854	0.554	0.294	1.411	4.667	50.5	1.9	4.8	15.8						
10000	132.193	4.928	2.577	10.660	46.755	51.3	1.9	4.1	18.1						
100000	1306.793	47.560	22.535	115.372	462.908	58.0	2.1	5.1	20.5	*** WDBC D=30***					
1	0.966	0.152	0.125	0.303	0.262	7.7	1.2	2.4	2.0						
10	1.167	0.165	0.130	0.319	0.256	9.0	1.3	2.5	2.0						
100	3.161	0.342	0.174	0.416	0.762	18.2	2.0	2.5	4.4						
1000	23.236	2.076	0.601	1.387	6.386	38.7	3.5	2.3	10.6						
10000	222.034	19.105	4.712	11.389	63.871	47.1	4.1	2.4	13.6						
100000	2224.134	188.080	46.114	111.223	634.687	48.2	4.1	2.4	13.8						

Table 3: Quality of maps produced by the different training algorithms (SOM=Classic SOM, VSM=VSOM, P-V=Par-VSOM, X-P=Xpysom, Q-S=Quicksom and D=Dimensions).

iter	15x10					150x100					200x150					
	SOM	VSM	P-V	X-P	Q-S	SOM	VSM	P-V	X-P	Q-S	SOM	VSM	P-V	X-P	Q-S	
1	0.50	0.15	0.09	0.50	0.45	0.41	0.00	0.00	0.50	0.12	0.40	0.00	0.00	0.50	0.08	
2	0.43	0.53	0.48	0.37	0.49	0.02	0.45	0.49	0.50	0.50	0.34	0.45	0.49	0.47	0.50	
3	0.92	0.95	0.93	0.88	0.48	0.42	0.79	0.49	0.40	0.50	0.12	0.85	0.77	0.32	0.50	
4	0.93	0.91	0.91	0.92	0.37	0.92	0.91	0.96	0.28	0.48	0.92	0.93	0.91	0.29	0.48	
5	0.95	0.94	0.94	0.87	0.27	0.96	0.99	0.95	0.26	0.41	0.90	0.99	0.97	0.32	0.37	
						*** Iris, D=4***										
						*** Epil, D=8***										
1	0.03	0.14	0.15	0.72	0.40	0.12	0.00	0.00	0.46	0.06	0.12	0.00	0.0	0.46	0.13	
2	0.70	0.56	0.40	0.60	0.48	0.03	0.45	0.45	0.49	0.50	0.07	0.38	0.40	0.50	0.50	
3	0.92	0.92	0.94	0.81	0.80	0.31	0.68	0.53	0.36	0.50	0.27	0.40	0.64	0.41	0.50	
4	0.94	0.92	0.93	0.65	0.79	0.45	0.48	0.68	0.29	0.86	0.85	0.60	0.56	0.40	0.56	
5	0.96	0.91	0.93	0.95	0.78	0.85	0.97	0.96	0.40	0.84	0.91	0.98	0.93	0.38	0.54	
						*** WDBC, D=30***										
1	0.31	0.14	0.11	0.68	0.37	0.00	0.00	0.00	0.62	0.13	0.07	0.00	0.00	0.50	0.00	
2	0.50	0.53	0.50	0.67	0.66	0.08	0.51	0.45	0.53	0.55	0.27	0.55	0.44	0.50	0.50	
3	0.90	0.92	0.88	0.50	0.80	0.30	0.48	0.64	0.40	0.66	0.40	0.60	0.63	0.40	0.50	
4	0.92	0.90	0.90	0.69	0.67	0.47	0.81	0.80	0.43	0.89	0.52	0.85	0.85	0.44	0.50	
5	0.93	0.92	0.93	0.68	0.68	0.88	0.90	0.91	0.37	0.76	0.81	0.97	0.98	0.37	0.50	

As part of our tests, we compared the performance and the quality of the maps generated by our parallel Par-VSOM with two CPU SOM and two GPU SOM variants. The quality of the maps is based on the convergence index as define in [30]. The CPU single-node tests used the SOM and the VSOM algorithms included as part of the R language *Popsom* package with C bindings applications. In contrast, the parallel comparisons were done using the two GPU-based SOM packages; *Quicksom* with Python 3, Pytorch 1.4 and *Xpysom* using Tensorflow 2.0 in their implementation.

For our experiments we used three real-world datasets to train our algorithms:

1. Iris [31] - a dataset with 150 instances and 4 attributes that describes three different species of Iris.
2. Epil [32] - a dataset on two-week seizure counts for 59 epileptics. The data consists of 236 observations with 8 attributes. The dataset has two classes - placebo and progabide, a drug for epilepsy treatment.
3. Wisconsin Breast Cancer Dataset (wdbc) [33] - a dataset with 30 features and 569 instances related to breast cancer in Wisconsin, for our experiment we generated a random normalized sample of 100 instances. The dataset has two classes: malignant and benign.

These datasets are purposely selected to test the algorithm performance by increasing the dimensionality complexity of the input data. To measure the Par-VSOM performance, we ran each timing test three times and took the average time over these runs. The times reported are the time required for the CPU to perform the calculations and it is given in CPU seconds. Similarly, the quality tests were done by averaging three quality measurements using the convergence index (CI) explain in detail in [30] and included as part of the R *Popsom* Package

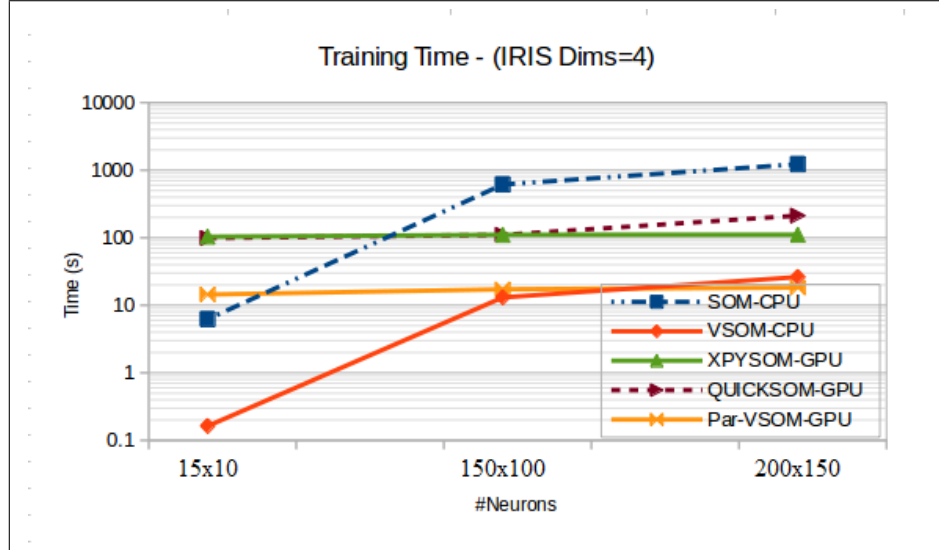


Figure 4: Iris Training Time

[29]. The CI provides a 0 to 1 numbering scale to measure the maps' quality, with 0 represents the lowest quality and 1 the highest quality. Furthermore, three map sizes were considered for these experiments,  $15 \times 10$  (small),  $150 \times 100$  (medium),  $200 \times 150$  (large), to see how the different implementations perform on different map sizes. In addition, we trained with various number of training iterations (in powers of 10) to discover what type of effect a change of training duration had on the implementations.

### 1.6.3 Results

In the large map environment results included in Table 2, we see the recurrent speed-up gains of the algorithm with larger maps. The large size of data buffers require for the calculations, the CPU cache memory size limitations and DDR4 lower clock rate does present an performance impact for the SOM and VSOM CPU variants. The large workload and substantial computational resources available in the GPU, allows the Par-VSOM performance scale further. Here, the Par-VSOM achieves a speed-up of 67 in comparison to the SOM. The table results



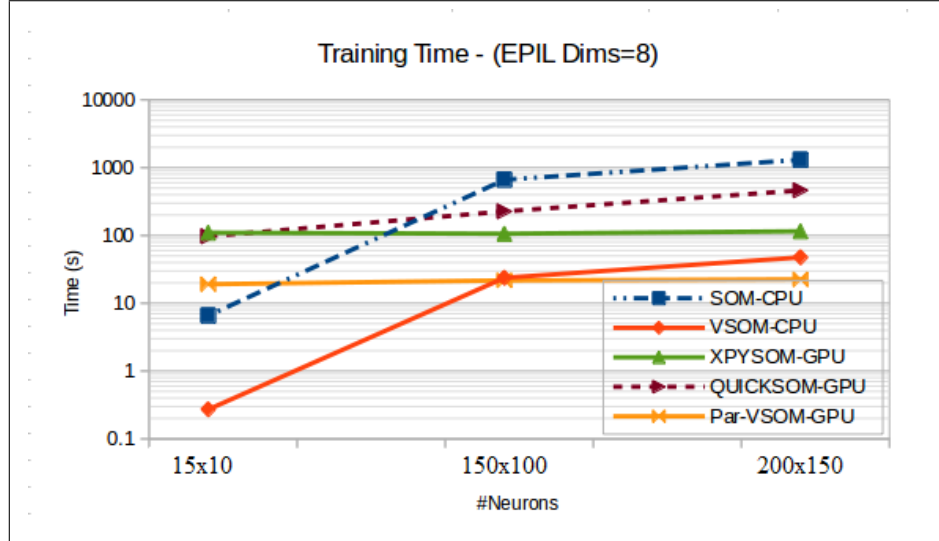


Figure 5: Epil Training Time

demonstrates, the Par-VSOM achieves superior speed-up in all the three datasets comparisons, surpassing the speed rates of all the other algorithm implementations. Due to the parallel BatchSOM algorithm used by the Xpysom and Quicksom, the speed-up patterns are influenced by the dimensionality and dataset instance size and do not follow a uniform pattern like the SOM speed-ups. In this large map environment, the Par-VSOM surpassed the SOM with a 67, the VSOM with a 4.1, *Xpysom* with 6.1 and the *Quicksom* by 20 speed-up increase.

The training time charts included in Figure 1.6.3, capture a generalize representation of the overall results. The Par-VSOM offers speedup performance increases for the three datasets in medium and larger size maps instances. The obtained results allows us to establish a direct relation between large neuronal maps and better achievable times using the Par-VSOM. That is, with a higher number of neurons an scalable speed up can be achieved.

The Table 7 illustrates the baseline quality of original algorithms using our three datasets. The results present us with a recurring behaviour in most of the maps, their is a pattern to decrease the convergence quality when the datasets

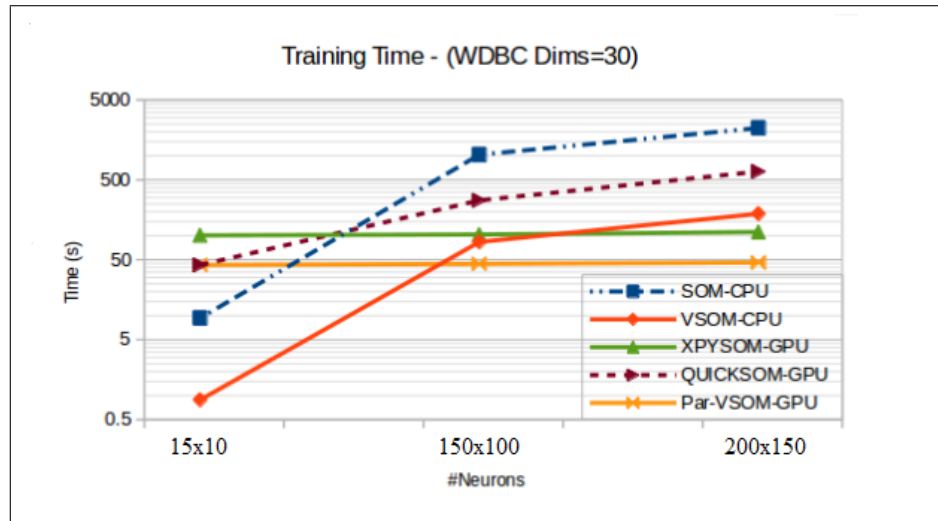


Figure 6: WDBC Training Time

dimensionality increases. However, we also identified as the size of the maps increases, there is tendency for the vectorized variants (VSOM and Par-VSOM) to generate higher quality maps. Furthermore, our testing demonstrates *Xpysom* and *Quicksom* SOM parallel versions can not reach a high convergence index when larger map sizes are used.

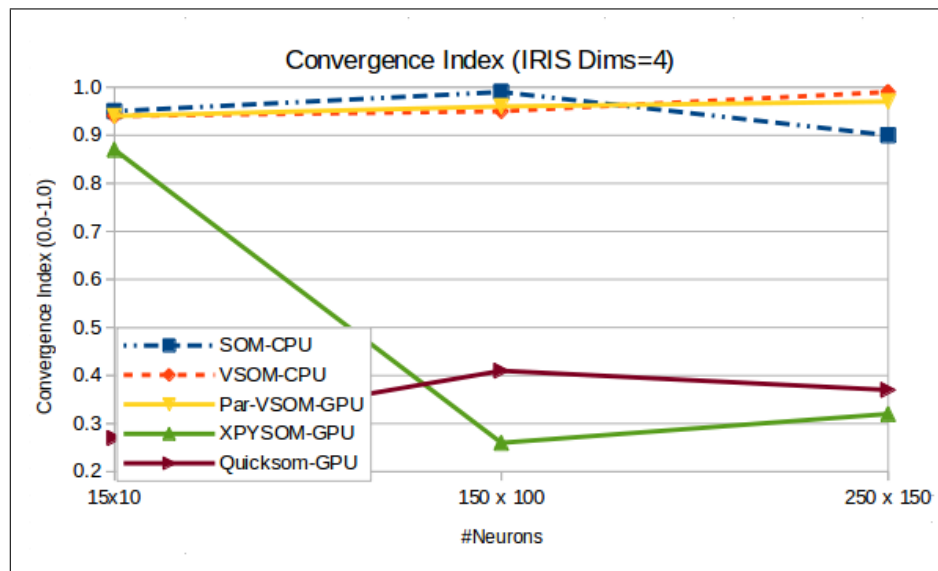


Figure 7: Iris Convergence Index (Map Quality)

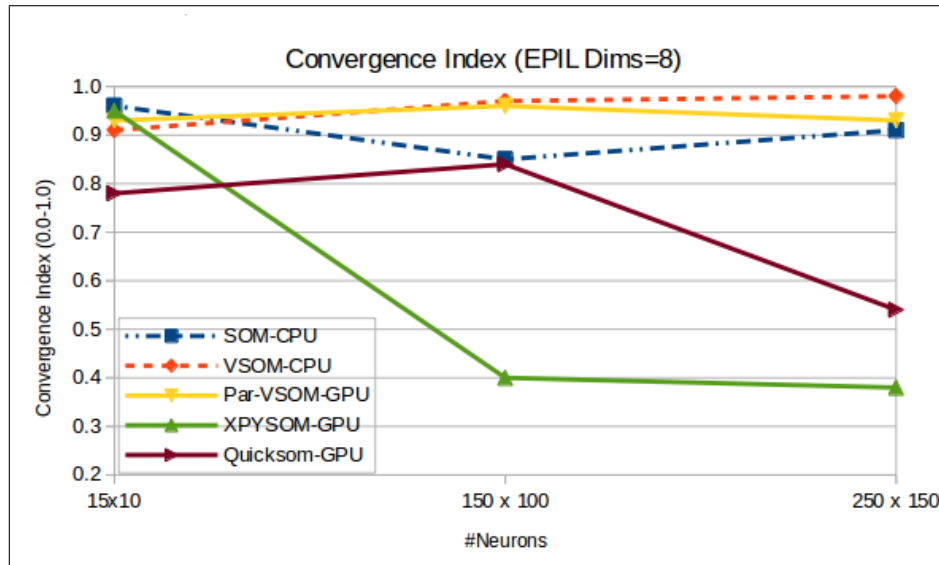


Figure 8: Epil Convergence Index (Map Quality)

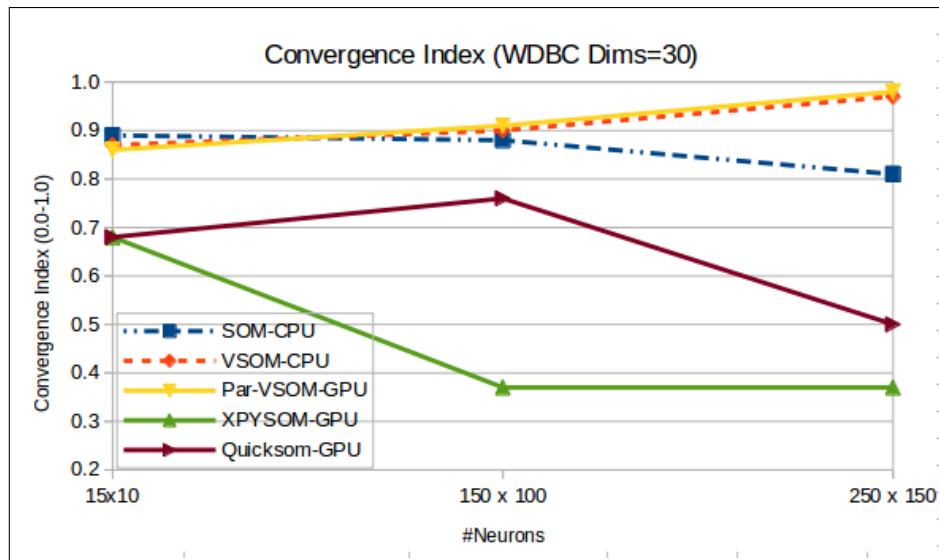


Figure 9: WDBC Convergence Index (Map Quality)

In terms of the quality of the maps, Figure 7 - 9 captures all the algorithm convergence indexes for the three datasets. As illustrated, the Par-VSOM maintains relatively the same quality as the original SOM and the VSOM variants in all the maps. In contrast, the parallel GPU SOM variants (*Xpysom* and *Quicksom*) only obtained good quality indexes with smaller maps (15 x 10). In both of these

parallel packages, the convergence index quality starts decreasing drastically after trying to organized medium and larger SOM maps.

## 1.7 Conclusions

This work introduced the Par-VSOM, a highly parallel, vectorized and matrix-based implementation of stochastic training for self-organizing maps. The novel implementation presented here provides substantial performance increases over Kohonen’s iterative SOM algorithm (up to 67 times faster), the CPU based vectorized VSOM (up to 4 times faster), the GPU *Xpysom* (up to 6.1 times) and *Quicksom*’s GPU (up to 20 times) in large maps environments. The performance gains follow a direct relation with the increment of the map sizes, as shown in Figure 4 - 6. Furthermore, the results obtained by increasing the dimensionality and maps sizes demonstrated the Par-VSOM provides a scalable speed-up performance when the neuronal map size increases. In terms of the quality of the maps, the maps produced by Par-VSOM approximates the high quality values generated by the VSOM iterative algorithms and original Kohonen’s SOM algorithm.

In the proposed design, the Par-VSOM is a multi-threaded algorithm running in a GPU and therefore is an adequate replacement for iterative stochastic training of SOM and parallel SOM variants. We are currently investigating how the Par-VSOM can be implemented in an FPGA and what kind of performance increase we can expect from this type of hardware architecture. Based on our results, the Par-VSOM can be viewed as an alternative to parallel SOM and a new alternative for other parallel algorithms for clustering and pattern recognition. In summary, since the training algorithms results demonstrate the produce maps are roughly the same quality, the Par-VSOM provides a parallel and high-performance alternative to SOM algorithms.

## List of References

- [1] T. Kohonen, *Self-organizing maps*. Springer Berlin, 2001.
- [2] B. Barney, *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory, 2018.
- [3] J. Li, B. M. Chen, and G. H. Lee, “So-net: Self-organizing network for point cloud analysis,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9397–9406.
- [4] M. A. C. Ramos, B. C. C. Leme, L. F. de Almeida, F. C. P. Bizarria, and J. W. P. Bizarria, “Clustering wear particle using computer vision and self-organizing maps,” in *2017 17th International Conference on Control, Automation and Systems (ICCAS)*, 2017, pp. 4–8.
- [5] L. Hamel, *VSOM: Efficient, Stochastic Self-organizing Map Training: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 2*, 01 2019, pp. 805–821.
- [6] G. L. T. C. R. Mancini, A. Rotacco, “Xpysom,” <https://github.com/Manciukic/xpysom>, 2020.
- [7] V. Mallet, M. Nilges, and G. Bouvier, “quicksom: Self-organizing maps on gpus for clustering of molecular dynamics trajectories,” *Bioinformatics*, vol. 37, no. 14, pp. 2064–2065, 2021.
- [8] V. Mallet, M. Nilges, and G. Bouvier, “Quicksom,” <https://github.com/bougui505/quicksom>, 2021.
- [9] A. Morán, J. L. Rosselló, M. Roca, and V. Canals, “Soc kohonen maps based on stochastic computing,” in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–7.
- [10] T. Richardson and E. Winer, “Extending parallelization of the self-organizing map by combining data and network partitioned methods,” *Advances in Engineering Software*, vol. 88, 10 2015.
- [11] M. Abadi, S. Jovanovic, K. Ben Khalifa, S. Weber, and M. Bedoui, “A scalable flexible som noc-based hardware architecture,” 01 2016.
- [12] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.
- [13] B. Silva and N. Marques, “A hybrid parallel som algorithm for large maps in data-mining,” *New Trends in Artificial Intelligence*, 2007.

- [14] P. T. Rauber, Andreas and D. Merkl, “parsom: a parallel implementation of the self-organizing map exploiting cache effects: making the som fit for interactive high-performance data analysis,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000*, vol. 6, 2000.
- [15] S.-J. Sul and A. Tovchigrechko, “Parallelizing blast and som algorithms with mapreduce-mpi library,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 481–489.
- [16] E. S. Schabauer, Hannes and T. Weishaupl, “Solving very large traveling salesman problems by som parallelization on cluster architectures,” in *Sixth Internatioanl Conference on Parallel and Distributed Computer Applications and Technologies PDCAT’ 05*, 2005.
- [17] G. Davidson, “A parallel implementation of the self organising map using opencl,” *University of Glasgow*, 2015.
- [18] F. C. Moraes, S. C. Botelho, N. Duarte Filho, and J. F. O. Gaya, “Parallel high dimensional self organizing maps using cuda,” in *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*. IEEE, 2012, pp. 302–306.
- [19] R. Mancini, A. Ritacco, G. Lanciano, and T. Cucinotta, “Xpysom: high-performance self-organizing maps,” in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 209–216.
- [20] P. Wittek, S. C. Gao, I. S. Lim, and L. Zhao, “Somoclu: An efficient parallel library for self-organizing maps,” *arXiv preprint arXiv:1305.1422*, 2013.
- [21] K.-H. Kim, S.-T. Yun, S. Yu, B.-Y. Choi, M.-J. Kim, and K.-J. Lee, “Geochemical pattern recognitions of deep thermal groundwater in south korea using self-organizing map: Identified pathways of geochemical reaction and mixing,” *Journal of Hydrology*, vol. 589, p. 125202, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0022169420306624>
- [22] T. Li, G. Sun, C. Yang, K. Liang, S. Ma, and L. Huang, “Using self-organizing map for coastal water quality classification: Towards a better understanding of patterns and processes,” *Science of The Total Environment*, vol. 628-629, pp. 1446–1459, 2018. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0048969718305552>
- [23] S. Lokesh, P. M. Kumar, M. R. Devi, P. Parthasarathy, and C. Gokulnath, “An automatic tamil speech recognition system by using bidirectional recurrent neural network with self-organizing map,” *Neural Computing and Applications*, vol. 31, no. 5, pp. 1521–1531, 2019.

- [24] G. Vettigli, “Minisom,” <https://github.com/JustGlowing/minisom>, 2021.
- [25] P. Jaaskelainen, “Task parallelism with opencl: A case study.” *Journal of Signal Processing Systems*, pp. 33–46, 2019.
- [26] L. L. Pilla, “Basics of vectorization for fortran applications,” *Research Report*, vol. RR-9147, pp. 1–9, 2018.
- [27] N. G. Dickson, K. Karimi, and F. Hamze, “Importance of explicit vectorization for cpu and gpu software performance,” *Journal of Computational Physics*, vol. 230, no. 13, pp. 5383–5398, 2011. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0021999111002026>
- [28] Nvidia.com, “Thrust quick start guide,” <https://docs.nvidia.com/cuda/thrust/index.html#abstract>, accessed: 2020-04-30.
- [29] L. Hamel, B. Ott, and G. Breard, *popsom: Functions for Constructing and Evaluating Self-Organizing Maps*, 2016, r package version 4.1.0. [Online]. Available: <https://CRAN.R-project.org/package=popsom>
- [30] L. Hamel, “Som quality measures: An efficient statistical approach,” in *Advances in Self-Organizing Maps and Learning Vector Quantization*. Springer, 2016, pp. 49–59.
- [31] R. A. Fisher, “The use of multiple measurements in taxonomic problems,” *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
- [32] P. F. Thall and S. C. Vail, “Some covariance models for longitudinal count data with overdispersion,” *Biometrics*, pp. 657–671, 1990.
- [33] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, “Nuclear feature extraction for breast tumor diagnosis,” in *IS&T/SPIE’s Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics, 1993, pp. 861–870.

## MANUSCRIPT 2

**High-Level Synthesis Parallelization and Optimization of Vectorized  
Self-Organizing Map**

by

<sup>1</sup>Omar X. Rivera Morales, Lutz Hamel submitted to the 18th International  
Conference on Data Science (ICDATA'22), Nevada, USA, 2022.

---

<sup>1</sup>Omar X. Rivera Morales and Lutz Hamel are with Department of Computer Science and Statistics, University of Rhode Island, Kingston, RI, 02881, Email {oxriveramorales, lutzhamel}@uri.edu.



## 2.1 Abstract

The nature of the Self-Organized Maps (SOM) requires a constant improvement of performance to address the increasing complexity of datasets. These demands have led to high-performance algorithms that run in hardware accelerators such as Graphical Processing Units (GPU) and Field Programmable Gate Array (FPGA). This work introduces a novel High-Level Synthesis (HLS) FPGA implementation for the vectorized SOM algorithm. The proposed algorithm is implemented using HLS parallelization and design optimization techniques available on the Xilinx Alveo FPGA Accelerator Card. This paper introduces the HLS-based algorithm and discusses the pipelining, unrolling, systolic array matrix reduction, and memory transformation techniques to improve the VSOM algorithm performance. Our HLS-VSOM experimental results show a significant performance increase over SOM CPU and parallel GPU variants.

## 2.2 Introduction

The self-organizing map (SOM) is a neural network designed for unsupervised machine learning [1]. After clustering the neurons, the generated maps can be utilized in a diverse range of domains such as atmospheric science, nuclear physics, medical diagnosis, and other data domains [2]. See reference [1] for a more comprehensive literature survey.

This paper demonstrates the performance achievable using various HLS techniques for the VSOM, a highly efficient SOM algorithm published by Hamel [3]. The HLS-VSOM replaces all iterative constructs of the algorithm with a highly optimized kernel running in an FPGA. The HLS kernel provides substantial performance increases over Kohonen’s SOM iterative algorithm, VSOM, and other GPU SOM variants.

The FPGA implementation addresses the increasing demands for high-performance computing and optimization by using various HLS transformations. The HLS optimization can be categorized into three major classes: Pipelining, Scaling, and Memory. Pipelining transformations allow overlapping the instructions from the processor through increasing the execution flow. Scaling are transformations that increase the computational parallelism, and memory transformation increases the read and write efficiency. In addition, we utilized a systolic array matrix reduction using Digital Signal Processors (DSPs) to accelerate some portions of the algorithm.

Our experimental results show that the maps produced by the HLS-VSOM are equivalent in quality to the maps produced by the VSOM and the original SOM iterative algorithm. The current HLS-VSOM model is parallel and highly optimized, therefore, well suited as a replacement for other parallel algorithms to train the self-organizing maps. Since the FPGAs are currently the only hardware

accelerators allowing the use of HLS tools, the algorithm HLS transformations are focused on the context of the FPGA accelerator. The HLS-VSOM implementation presented here is written in OpenCL with Pragmas directives and compiled with the Xilinx Vitis Vivado compiler. The Vitis compiler uses a high-level synthesis to generate traditional hardware design languages like VHDL or Verilog. The HLS connects the hardware and software developments on a single compilation environment and enables basic performance portability [4].

The paper is organized as follows: In Section 2.3 starts our discussion with an overview of the HLS and a brief description of the major stages. Under section 2.4, we included an introduction to the VSOM [3] vectorized rules; this is an implementation of a competitive learning scheme comprised of a competitive step and an update step with vector and matrix training. The relevant details about related research work are included in Section 2.6. As part of Section 2.5, we develop the HLS-VSOM training and examine the instruction pipelining, scaling data level parallelisms, array partitioning, and memory optimization transformations. Under section 2.7, we included the study of the performance of our parallel vectorized training implementation by comparing it to various CPU and GPU SOMs variants. Finally, in Section 2.8, we conclude our discussion with a summary of the observations and some future research ideas under consideration.

### 2.3 High Level Synthesis

The HLS acceleration serves as an answer to address the complex and error-prone hardware design process. The HLS has been known to cope with these losses, obtaining design productivity gains by separating functional system verification, performed from a time-agnostic high-level language, from timed system verification, performed after automatically inferring hardware-specific code [5].

Nowadays, the software and hardware communities are embracing the HLS

tools. The HLS bridges the gap between hardware and software development and enables fundamental performance portability implemented in the compilation system. [4]. Generally, the HLS systems rely on the abstraction and low-level hardware control provided by C/C++ and OpenCL languages.

Companies like Xilinx with the Vivado/Vitis HLS design suite and Intel FPGA SDK offer a structured high-level languages solution for people trying to program configurable hardware, such as FPGAs. However, the HLS approach does not come with some problems. Robattu in [6] listed some of the significant drawbacks of using the HLS.

- Imperative high-level programming languages imperative formulations can not differentiate between iterations over time and iterations over space. This limitation does not translate appropriately to hardware architecture where all the events are occurring in parallel.
- The substantial level of parallelization leads to a "bottleneck" on memory accesses at the implementation level, which immediately leads to a "bottle-neck" on memory accesses [7].

These drawbacks can be circumvented by relying upon so-called applicative or functional languages in which algorithms are described as a (mathematical) composition of side-effect free functions [6]. Another solution is to provide a hardware behavior and software iterations description. The HLS environment allows the programmer to include "Pragmas" directives with a vast amount of functionality encapsulating an instruction of the expected system architecture behavior.

The HLS source to hardware stacks process transforms an imperative code into a hardware design language (HDL) such as Verilog or Vhdl. Here, we provide a sequential description of the major stages based on Johannes [4]:

1. **High-level synthesis** converts an imperative and procedural source code description into functional hardware-level description. This generally translates as converting high level languages with Pragmas directives like C++ or OpenCL into a Hardware Description Language (HDL) such as Verilog or VHDL.
2. **Hardware synthesis** creates a logical mapping between the register level circuits description from the HDL and the physical component available in the target architectures.
3. **Place and Route** maps the hardware logical mapping into the physical components available in the hardware. During this, the system performs target-specific optimization to minimize between registers and cable length. As part of the optimization, the system will configure a hardware environment that increases the best achievable frequency.
4. **Bitstream generation** creates the bitstream image that will be translated into the gate array configuration to form the equivalent to a specific circuit.

## 2.4 Vectorization of Self-Organizing Maps

The origins of the self-organizing maps model can be traced back to the Vector Quantization (VQ) method [1]. The VQ is a signal-approximation algorithm that approximates a finite “codebook” of vectors  $m_i \in R^n, i = 1, 2, \dots, k$  to the distribution of the input data vector  $x \in R^n$ . In the SOM context, the approximated codebook allows us to categorize the nodes and form an “elastic network,” which becomes a meaningful, coordinated map or grid system.

From a computational perspective, the SOM can be described as a mapping of high dimensional input data onto a low dimensional neural network projected as a 2D or three-dimensional (3D) map. The mapping is accomplished by assuming

that the input data set is a real vector such as  $x = [\xi_1, \xi_2, \dots, \xi_n]^T \in R^n$ . The SOM neuronal map can be defined as a model containing the parametric real vector  $m_i = [u_{i1}, u_{i2}, \dots, u_{in}]^T \in R^n$  associated with the neurons' weights. If we consider the distance between the input vector  $x_k$  and the neuron vector  $m_i$  then we can establish an initial minimum distance relation between the input and the neurons by calculating the Euclidean distances. Then, these distances are used to identify the best matching unit (BMU) index with equation (14).

$$c = \mathbf{argmin}_i(\|\mathbf{m}_i - \mathbf{x}_k\|^2) \quad (14)$$

To define the SOM in terms of matrix and vector operations it is assumed that the map's neurons are stored in a  $n \times d$  matrix  $\mathbf{M}$  where each row  $i$  represents the neuron  $\mathbf{m}_i$  with  $d$  components,

$$\mathbf{M}[i, ] = \mathbf{m}_i = (m_1, \dots, m_d)_i, \quad (15)$$

with  $i = 1, \dots, n$ . The training data  $x$  consists of a set  $D = \{\mathbf{x}_1, \dots, \mathbf{x}_l\}$ . The set can be defined as a  $l \times d$  matrix where each row  $k$  represents the training vector  $\mathbf{x}_k$  with  $d$  components,

$$\mathbf{D}[k, ] = \mathbf{x}_k = (x_1, \dots, x_d)_k, \quad (16)$$

with  $k = 1, \dots, l$ .

Essential details to consider include (1) the dimensionality  $d$  for the input, and (2) the neuron vectors are required to be the same for well-defined matrix operations.

#### 2.4.1 The SOM and VSOM Competitive Step

In the classic SOM we use an iterative process to find the BMU using 14. Here the  $i = 1, 2, \dots, n$  represents the index of the neurons in the map and  $\mathbf{m}_i$  represents

the neuron in index  $i$ . The *argmin* is a function that returns the minimum value and  $c$  contains the index of the BMU.

In contrast, in the VSOM competitive step, we find the BMU for a particular training instance  $\mathbf{x}_k$  calculating the Euclidean distance as a set of vector and matrix operations. These operations find the  $c$  index associated with the neuron with the minimum distance to the training instance. The BMU  $c$  index corresponds to the neuron in the map with the highest resemblance to the particular  $\mathbf{x}_k$  selected for training during the epoch.

The first step to calculate the BMU requires us to compute a matrix  $\mathbf{X}$  to hold a randomly selected training vector. The matrix  $\mathbf{X}$  in equation (17) is defined with a component sizes of  $n \times d$ , where each row is holding the current epoch *training vector*  $\mathbf{x}_k = (x_1, x_2, \dots, x_d)_k$ , which is randomly selected from matrix  $\mathbf{D}$ ,

$$\mathbf{X} = \mathbf{1}^n \otimes \mathbf{x}_k. \quad (17)$$

Here, the symbol  $\otimes$  represents the outer product and  $\mathbf{1}^n$  is a column vector defined as,

$$\mathbf{1}^n = \underbrace{(1, 1, \dots, 1)}_n^{\mathbf{T}}. \quad (18)$$

Since  $\mathbf{1}^n$  is a column vector and  $\mathbf{x}_k$  is a row vector the operation in (17) is well defined. After populating our epoch training instance matrix  $\mathbf{X}$  with the duplicated  $\mathbf{x}_k$  values, equations (19), (20) and (21) are used to compute the square of the Euclidean distances between the map neurons and the input vector,

$$\mathbf{\Delta} \leftarrow \mathbf{M} - \mathbf{X} \quad (19)$$

$$\mathbf{\Pi} \leftarrow \mathbf{\Delta} \circ \mathbf{\Delta} \quad (20)$$

$$\mathbf{s} \leftarrow \mathbf{\Pi} \times \mathbf{1}^d \quad (21)$$

In equation (19) we calculate the difference between the matrices with an element-by-element matrix subtraction. In equation (20) we use the Hadamard product to

allow us to calculate the  $\mathbf{\Pi}$  matrix, in this context  $\circ$  represents the element-by-element matrix product and  $\mathbf{X}$ ,  $\mathbf{M}$ ,  $\mathbf{\Delta}$  and  $\mathbf{\Pi}$  are all  $n \times d$  matrices.

Lastly, in equation (21) we use a ‘row sum’ matrix reduction to compute the vector  $\mathbf{s}$  of size  $n$ . Here,  $\mathbf{1}^d$  is a column vector similar to (18) with the dimensionality defined by the value of  $d$ . In order to find the BMU, we search for the location of the minimum value in vector  $\mathbf{s}$ .

#### 2.4.2 The SOM and VSOM Update Step

In the classic stochastic SOM, the update step occurs after completing the BMU calculations, the updates to the neuronal weights are accomplished using the training instance  $x_k$  to influence the best matching neuron and its surrounding neighborhood.

$$\mathbf{m}_i \leftarrow \mathbf{m}_i - \eta(\mathbf{m}_i - \mathbf{x}_k)h(c, i) \quad (22)$$

The weights update step in equation (22), affects every neuron inside the neighborhood radius of influence. Here, the learning rate  $\eta$  serves as a scaling factor between 0 and 1. The  $h(c, i)$  acts as the loss function, where  $i = 0, 1, \dots, n$  and it can be defined as,

$$h(c, i) = \begin{cases} 1 & \text{if } i \in \Gamma(c), \\ 0 & \text{otherwise,} \end{cases} \quad (23)$$

where  $\Gamma(c)$  is the neighborhood of the best matching neuron  $\mathbf{m}_c$  with  $c \in \Gamma(c)$ . In the SOM, the learning factor and the loss function both decreased monotonically over time [1].

In the VSOM, the update step also occurs after the BMU calculations but all the neurons update operations are accomplished with matrix operations and is defined as,

$$\mathbf{M} \leftarrow \mathbf{M} - \eta\mathbf{\Delta} \circ \mathbf{\Gamma}_c. \quad (24)$$



Here,  $\eta$  is the learning rate,  $\Delta$  contains the calculations of the difference between the neurons and the selected training instance as computed in (19), and the symbol  $\circ$  represents the Hadamard product. Similarly to the SOM, in the VSOM, the learning rate  $\eta$  is linearly reduced as epochs increase.

The competitive and the update steps are computed during each epoch using the randomly selected training instances until some convergence criterion is fulfilled. After completing multiple learning iterations and updating the neurons weights, every vector will be assigned or clustered to specific neurons in the grid, preserving the neighborhood topology.

## 2.5 High-Level Synthesis VSOM

### 2.5.1 HLS VSOM Algorithm

In the HLS-VSOM, the vector and matrix operations of the original VSOM are executed using a High-Level Synthesis kernel executing in custom FPGA architecture. The HLS kernel allows us to generate parallel operations and obtain performance increase gains by manipulating the algorithm behavior within the FPGA fabric. Algorithm 3 and 4 summarizes the matrix and vector operations required for the parallel HLS-VSOM training. For a more detailed explanation of the SOM and VSOM algorithms, see reference [3].

This work proposes a set of HLS transformations that are imperative to generate an efficient hardware kernel. As part of our HLS algorithm design, we employ three major classes of transformation to improve performance: pipelining, that allows us to improve execution during the for loops within the SOM; scaling to manipulate the instructions parallelism and allow us to execute Single Instruction Multiple Data (SIMD) instructions and memory enhancing transformation to select more efficient memory architectures and access ports settings. Some of the HLS transformation are “Pragma” directives and attribute instruction in-

---

**Algorithm 3** The HLS-VSOM training algorithm.

---

```

1: Given:
2:    $D \leftarrow \{\text{training instances, a } l \times d \text{ matrix}\}$ 
3:    $M \leftarrow \{\text{neurons, a } n \times d \text{ vector of tuples}\}$ 
4:    $\eta \leftarrow \{\text{learning rate } 0 < \eta < 1\}$ 
5:    $\Gamma(c) \leftarrow \{\text{neighborhood function for some neuron } c\}$ 
6:    $\text{minIndex}(s) \leftarrow \{\text{func, returns location of min. val in } s\}$ 
7:    $\Phi \leftarrow \{\text{Rowsum reduction using Systolic Array dot product}\}$ 
8:    $\Omega \leftarrow \{\text{Pipeline, unrolled loops kernel operations}\}$ 
9:    $R \leftarrow \{\text{Random index values list}\}$ 
10:   $O \leftarrow \{\text{constant column vector with value of 1's}\}$ 
11:
12: Repeat:
13: /***/Select a matrix training instance as vector
14: for some  $k = 1, \dots, l$ : ***/
15:
16:    $x_k \leftarrow D[R_k]$ 
17:
18: /***/Find the winning neuron using accelerated kernels ***/
19:    $X \leftarrow \Omega_x(1^n \otimes x_k)$ 
20:    $\Delta \leftarrow \Omega_\Delta(M - X)$ 
21:    $\Pi \leftarrow \Omega_\Pi(\Delta \circ \Delta)$ 
22: /***/Reduction (Rowsum) Using Systolic Arrays and DSP***/
23:    $s \leftarrow \Phi_s(\Pi \cdot O)$ 
24:    $c = \text{minIndex}(s)$ 
25:
26: /***/Update neighborhood with vector operations ***/
27:    $\Gamma_c \leftarrow \Omega_\Gamma(\Gamma(c))$ 
28:    $M_{new} \leftarrow \Omega_{M_{new}}(M_{current} - \eta \Delta \circ \Gamma_c)$ 
29: done
30: return  $M_{new}$ 

```

---

serted in the code and interpreted by the HLS compiler, while others may require adding or modifying the configuration files. Some of the Pragmas utilized in our implementation included the `opencl_unroll_hint(X)`, `xlc_pipeline_loop(X)` and `xlc_array_partition(complete, X)`.

### 2.5.2 Pipelining and Dataflow

The pipeline transformations are an essential aspect to consider during the HLS integration. Pipelining allows to efficiently send data directly from one compu-

---

**Algorithm 4** The HLS-VSOM Neighborhood Function  $\Gamma$ .

---

```

1: given:
2:    $c \leftarrow \{\text{index of winning neuron}\}$ 
3:    $n \leftarrow \{\text{the number of neurons on the map}\}$ 
4:    $nsize \leftarrow \{\text{neighborhood radius}\}$ 
5:    $\mathbf{P} \leftarrow \{\text{an } n \times 2 \text{ matrix with } \mathbf{p}_i = \mathbf{P}[i, ] = (x_i, y_i)\}$ 
6:    $\mathbf{1}^n \leftarrow \{\text{constant column vector with value 1}\}$ 
7:    $\mathbf{0}^n \leftarrow \{\text{constant column vector with value 0}\}$ 
8:    $\Omega \leftarrow \{\text{Pipeline and unrolled loops kernel operations}\}$ 
9:
10:
11:  $P_c \leftarrow \Omega_{pc}(P[c, ])$ 
12:  $\mathbf{C} \leftarrow \Omega_C(\mathbf{1}^n \otimes \mathbf{p}_c)$ 
13:  $\mathbf{\Delta} \leftarrow \Omega_{\Delta}(\mathbf{P} - \mathbf{C})$ 
14:  $\mathbf{\Pi} \leftarrow \Omega_{\Pi}(\mathbf{\Delta} \circ \mathbf{\Delta})$ 
15:
16: /** Perform rowsum matrix reduction
17:  $\mathbf{d} \leftarrow \Omega_d(\mathbf{\Pi}_x + \mathbf{\Pi}_y)$ 
18:  $\mathbf{hood} \leftarrow \Omega_{hood}(\text{ifelse}(\mathbf{d} < (nsize \times 1.5)^2, \mathbf{1}^n, \mathbf{0}^n))$ 
19: return hood

```

---

tational unit to the next, permitting instruction-level parallelism. This technique maximizes the usage of every core available of the processor with some instruction by dividing incoming instructions into a series of sequential steps performed by different processor units with different parts of instructions processed in parallel [8] as shown in Figure 10.

`__attribute__((xcl_pipeline_loop(1)))`

Clock cycle Instr. No.	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Figure 10: Pipelining HLS - [8]

Similarly to Pipelining, the Dataflow optimization allow to send data efficiently but it works between the Kernel functions. In our design, the dataflow Pragma enables the parallel execution between the functions within a kernel.

The pipelining in terms of the HLS-VSOM, improves the iterations within each one of the vectorized loop instruction by overlapping the instructions to compute all the matrix operations shown in the HLS-Vsom Algorithm 3 and 4. In our HLS kernel, we are pipelining all the matrix and vector operations to maximize the execution per clock ratio.

### 2.5.3 HLS VSOM Horizontal Unrolling (Vectorization)

In the VSOM algorithm, the stochastic SOM training is redefined to execute as a set of vector and matrix operations. Utilizing the unrolling HLS transformations to create vectorization for the loop iterations allows the FPGA fabric to create

parallel copies of the body of the loop to increase the algorithm performance. This is the most straightforward way of adding parallelism, as it can often be applied directly to an inner loop without further reordering or drastic changes to the nested loop structure. Vectorization is more powerful in HLS than SIMD operations on load/store architectures, as the unrolled compute units are not required to be homogeneous, and the number of units are not constrained to fixed sizes [4].

In the HLS-VSOM, all the matrix data elements are independent of each other and they can be executed as coarse-grained “embarrassingly parallel” [9] computing units allowing us to exploit the available hardware resources exploit multiple in the target platform .

In the HLS-VSOM context, the vectorization of the calculations can be implemented as vector instructions, or horizontal unrolling similar the SIMD instructions and are a form of Data-Level Parallelism as illustrated in Figure 11. These vector instructions apply the same operation over multiple data elements (like integers and floating-point values) concurrently, given that these items are stored contiguously in vector/SIMD registers [10]. For our implementation using an unrolling Pragma with a factor of 64 provided the best performance gains for our type of map sizes.

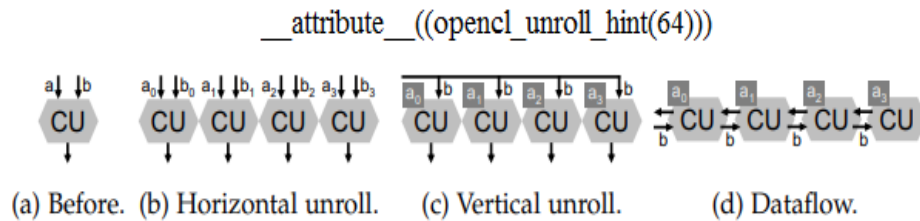


Figure 11: Scalability Transformations HLS - here the rectangles represent buffer space, such as FPGA registers or on chip Ram [4] and the CU refers to computational units.

### 2.5.4 HLS Par-VOM Memory Transformations

In the classic SOM with iterative operations, the operations per column are solved sequentially. This serial dependency results in high overhead and additional latency during every training epoch per memory access request. The HLS memory access transformation allows us to optimize the efficiency of the off-chip memory access, as shown in Figure 12

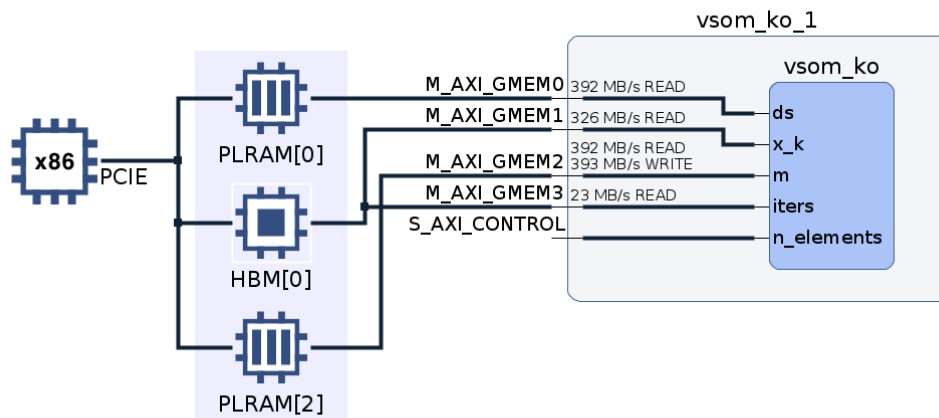


Figure 12: Par-VSOM HLS Memory (Striping) Access Transformations.

In our Xilinx Alveo cards, multiple banks with dedicated channels (e.g. High Bandwidth Memory (HBM) lanes) are available, this allows increasing the arrays bandwidth accessed by a factor equivalent to the number of memory interfaces connected, this is known as memory striping. The HLS environment allows us to explicitly define the striping by indicating the modules and the variables name associated to the data banks as shown in Figure 12. The striping results in parallel read and writes increase the overall bandwidth.

The Alveo accelerator cards contain HBM DRAM and DDR DRAM as external memory resources. In addition, in some accelerator cards, an additional

internal memory resource called PLRAM (UltraRAM and block RAM) is available. In the HLS-VSOM the global M matrix and the buffer containing the Data set are stored in PLRAM space. The less used buffers such as number of iterations and  $X_k$  random index array are allocated in the HBM space. All the other algorithm matrices are stored internally in local memory as part of the Block RAM or in registers.

Accessing the external memory has significant latency; it is recommended to use a burst accesses to global me High Bandwidth Memory (HBM) memory in and from PLRAM memory. Here, PLRAM is small shared memory that consist of UltraRAM/block RAM memory resources available in the FPGA.

As part of our HLS optimization, we also utilized array partitioning for all the internal VSOM vectors. The array partitioning converts the vectors into smaller arrays or separates them into individual registers elements. Since this transforms the elements of the array into registers, it increases the ports for read and write operations and improves the throughput of the design. Therefore, the array partitioning is recommended for smaller arrays since fully partitioning may cause quality and clock delays due to design complexity.

### 2.5.5 HLS Matrix Reduction with Systolic Arrays

In a systolic array, all processing elements, called systolic cells, perform computations simultaneously, while data, such as initial inputs, partial results, and final outputs, is being passed from cell to cell. When partial results are moved between cells, they are computed over these cells in a pipeline fashion. In this case, the computation of each single output is partitioned over these cells [11].

For our systolic array matrix “rowsum” reduction operations illustrated in Figure 13, we use the DSP available in the FPGA as independent Processing Elements (PE); communications between the PEs between and input and output

for the algorithm will take simultaneously achieving high performance.

As part of our HLS algorithm development, we discovered one of the major bottlenecks was the matrix rowsum reduction included in Algorithm 3 line 23. The latency of this instruction is due to the high amount of read and write access requested to the same local BRAM memory locations. Using the systolic array DSP approach allow us to access and execute in multiple PE at the same time alleviating the BRAM traffic and increasing the overall performance.

In the algorithm, we use a dot product  $\Phi_s(\Pi \cdot O)$  with the systolic array. Here  $\Pi$  contains the square of the differences of the distances calculated during the BME step, and  $O$  is a column vector of one. The result is a vector representative of a rowsum matrix reduction.

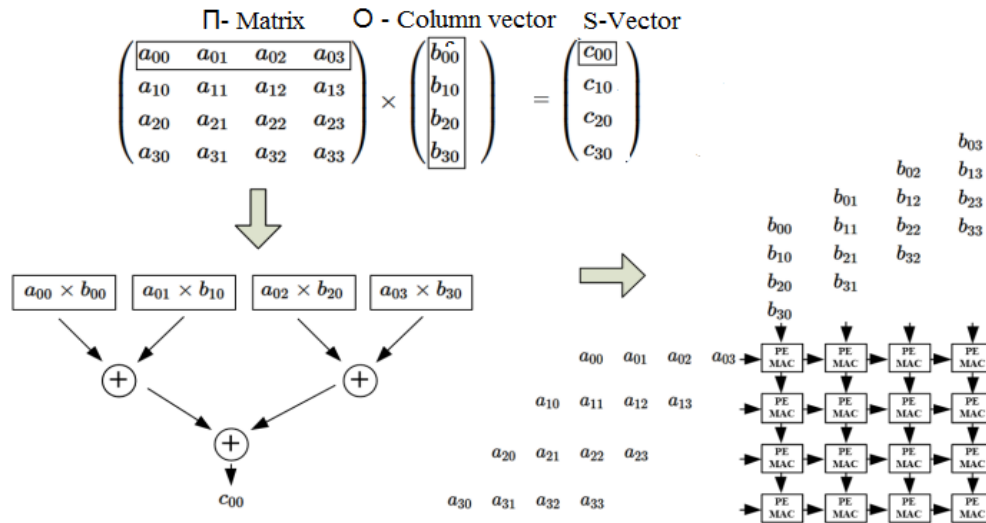


Figure 13: Systolic Array Matrix Multiplication. [12]

## 2.6 Related Work

In this section, we look at prior work related to high-level synthesis and FPGA SOM implementations. The recent research has demonstrated promising improvements using various methodologies associated with reconfiguration hardware meth-



ods. Recent scientific publications on this domain include: using a system on chip (SoC) to generate stochastic SOM [13], SOM Network-on-Chip (NoC) based solution [14], High Level Synthesis (HLS) targeting K-means algorithm [15], achieving high-performance computing applications via High-Level Synthesis [16], and using various types of hardware optimization techniques in FPGAs [17, 18, 19].

In general, all the research publications share the goal of finding optimal speed-up performance facilitating the higher synthesis implementation or using a hardware design language.

### **2.6.1 Stochastic SOM with FPGA SoC**

In his work, Moran proposed a novel System-on-Chip for a stochastic Self-Organizing map implementation. As part of his implementation, he generated several stochastic block design the Winner-Take-All (WTA) similarity check. This map acceleration solution can perform the self-learning and classification task with the same error rate as Matlab and consume 4 times less power consumption 21.5 mW than other Internet of Things (IoT) Devices.

### **2.6.2 A Scalable SOM based on a Sequential Systolic NoC**

Mehdi et al. adapted the NoC for SOM computations. His architecture consisted of a Vector Element Processing block to calculate the distance and update the weights; a Local Winner Search circuit (LWS) which compares the local distances and the received neighbour's distance; an Update Signal Generator (USG). As part of his experiments, he did a performance comparison against Core I7, Parallel FPGA, Systolic Array FPGA and the Noc Sequential systolic FPGA. The proposed NoC Sequential systolic FPGA architecture performs up to 724 MCUPS during the learning and 1168 MCPS in the recall phase for a 32-element input vector and promise a scalable performance by optimizing the architecture pipelining

[13].

### 2.6.3 High Level Synthesis (HLS) for K-means algorithm

The research presented by Younes[15] includes an efficient architecture implementation for a K-Nearest Neighbor (KNN) hardware accelerator targeting a modern System-on-Chips (SoCs). This KNN approach revolves in using a HLS design and was implemented on the Xilinx Zynqberry FPGA platform. The results compared with other state-of-the-art implementation indicate the proposed KNN offers between 1.4x and 875x speed and 41% and to 94% of energy consumption. In addition, they enhance the architecture with algorithmic level Approximate Computing Technique (ACTs) and improved the classification performance by 2.3x, loss a 3% percent of accuracy and reduced the energy consumption by 69% on average.

### 2.6.4 High-Performance Computing Applications via High-Level Synthesis

In his paper [16] Muslim presents an OpenCL HLS-based FPGA implementation applicable to K-nearest neighbor, Monte Carlo method for financial models and the Bitonic Sorting algorithm. The paper includes a performance comparison in terms of execution time, energy, and power consumption for some high-end GPUs is performed as well. One of the interesting aspect is, both of the algorithms have been implemented in OpenCL for the GPU and the FPGA. He concluded the FPGAs could surpass the GPU performance with HLS optimization directives. In addition, the FPGA are highly energy-efficient than GPUs in all the considered algorithms.

### 2.6.5 SOMs in GPUs

The GPUs also provide an excellent hardware solution for the parallelization of the SOM. The SOM GPU implementations are a recurrent topic in recent publications. Most of the SOM GPU variants are based on the batch SOM algorithm using new programming languages optimized for parallelism like OpenCL. In his research, Davidson[20] developed a parallel SOM with OpenCL for an Intel i7, AMD, and Nvidia GPU architectures. His research concluded that the parallel OpenCL SOM processing larger maps and running on a GPU could achieve a speed-up factor of more than 10X compared to the run time of SOM PAK run serially.

Among the SOM parallel approaches previously discussed, not too many offer an available open-source repository to validate the research findings or continue with further investigations. In this paper, we decided to compare our proposed HLS implementation with some of the widely available state-of-the-art parallel SOM projects packages. As part of the GPU comparisons, we utilize, XPySom [21] a parallel Batch-SOM variant implemented using the Google Tensorflow 2.0 framework and Python Numpy library. The XPySom package is based on the Minisom[22], a non-parallel, minimalistic and Numpy based widely known implementation of the SOM. The XPySom research paper [21] indicates their parallel variants outperforms the popular SOM GPU package Somoclu by two and three orders of magnitude. In addition, we also compare our HLS-VOM with the PAR-VSOM, our own GPU version of the Parallel VSOM written in CUDA Thrust.

## 2.7 Experiments

### 2.7.1 Hardware setup

The Par-VSOM HLS FPGA experiments used the Xilinx Alveo U50 Data Center accelerator cards to provide the optimized acceleration. The Alveo FPGA

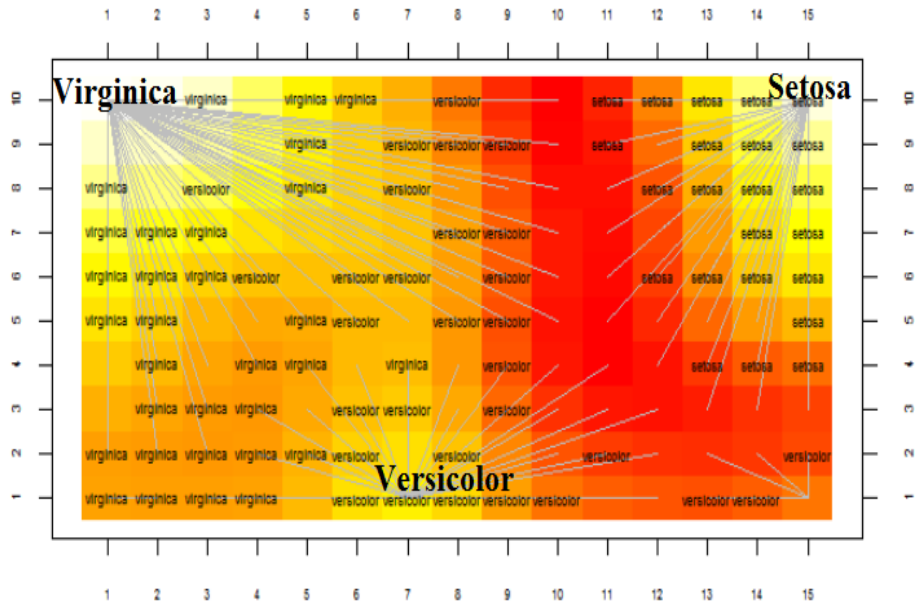


Figure 14: 15 x 10 Self Organizing Map for the Iris Dataset.

includes a Xilinx UltraScale Plus with 8 Gb of HBM memory and the host system included 8 virtual CPUs with a 128 GB of memory. The Par-VSOM and XPysom GPU parallel experiments were performed using the Amazon AWS cloud service instances with Linux and Deep Learning Amazon Machine Images (AMI). The sequential CPU experimental setting included an Intel Xeon E5 2686 running 2.7 GHz/ 3.0 GHz with 18 cores and capable of executing 36 threads. The GPU tests were performed in an AWS P3.2xlarge with 18 virtual Intel Xeon E5 2686 CPU operating at 2.7 GHz/ 3.0 GHz turbo and an NVIDIA Tesla V100. The Tesla V100 contains 5120 NVIDIA Cuda cores with 16 Gb of HBM2 memory. The Tesla V100 memory clock setting was 877 Mhz, with memory graphics clocked at 1530 Mhz.

### 2.7.2 HLS-VSOM setup and Hyper-Parameters

The CPU experimental setup utilized the default values of the SOM and VSOM Popsom [23]. For XPySom[21] package, we maintained the learning rate

constant to obtain higher convergence indexes and tune the hyper-parameters as defined in Table 4. For our map size selection, we followed the method proposed by Vesanto in [24]. That is, the recommended map size should contain approximately not less than  $5 * \sqrt{N}$  neurons where  $N$  is the number of data set observations. For the IRIS dataset that will be 61 neurons, in which case we started testing with  $8 \times 8$  as an approximation but eventually we decided to increase our map size to  $15 \times 10$  larger for more complexity.

Table 4: HLS-VSOM Hyper-Parameters.

**Hyper-Parameter**	**Values**
Training Iterations Range	$1 \times 10^0 \dots 1 \times 10^4$
Learning Rate $\eta$	0.7
Neighborhood Radius	Bubble
Training Data Sets	Iris, Epil, WDBC

Table 5: Times and Speed-up gains of the HLS Par-VSOM for different training algorithms using a  $15 \times 10$  map.

iter	Time SOM(s) CPU R\C	Time VSOM(s) CPU R\Fortran	Time P-VSOM(s) GPU Thrust	Time X-Som(s) CPU-GPU TensorFlow	Time H-VSOM(s) FPGA OpenCL	Speed-up H-VSOM/ SOM	Speed-up H-VSOM/ VSOM	Speed-up H-VSOM/ Par-VSOM	Speed-up H-VSOM/ XPySom
1	0.033	0.017	0.005	0.001	0.000034	961.1	495.1	145.6	29.1
10	0.031	0.017	0.007	0.009	0.000102	303.9	166.7	68.6	88.2
100	0.036	0.018	0.024	0.099	0.000481	74.8	37.4	49.9	205.8
1000	0.075	0.021	0.170	0.986	0.004052	18.5	5.2	41.9	243.3
10000	0.469	0.040	1.548	9.454	0.035312	13.3	1.1	43.8	267.7
*** Iris D=4***									
1	0.040	0.022	0.011	0.001	0.000066	600.0	330.0	162.0	15.0
10	0.041	0.020	0.013	0.010	0.000146	280.8	137.0	89.1	68.5
100	0.043	0.021	0.035	0.102	0.000511	84.0	39.1	68.4	205.2
1000	0.092	0.026	0.221	1.024	0.003811	24.1	6.3	58.0	274.4
10000	0.546	0.063	1.965	10.241	0.033091	16.5	1.9	59.4	309.5
*** Epil D=8***									
1	0.041	0.020	0.018	0.001	0.000143	286.0	139.5	125.6	7.0
10	0.043	0.020	0.024	0.010	0.000260	165.4	76.9	92.3	38.5
100	0.046	0.022	0.063	0.103	0.000815	56.4	28.2	77.7	126.3
1000	0.181	0.034	0.452	1.104	0.005946	30.4	5.7	76.0	185.7
10000	0.925	0.157	4.226	10.130	0.050086	18.5	3.1	84.3	202.3
*** WDBC D=30***									

Table 6: Times and Speed-up gains of the HLS-VSOM compare against a non-accelerated FPGA HLS-VSOM using a  $15 \times 10$  map. Our accelerated HLS-VSOM uses pipelined loops, dataflow, horizontal unrolling, array partitioning and systolic arrays for row sum reductions.

iter	Time HLS-VSOM(ms) FPGA Non-Accel	Time HLS-VSOM(ms) FPGA Accel	Speed-up Accel vs Non-Accel
*** Iris D=4***			
1	0.035	0.034	1.0
10	0.127	0.088	1.5
100	0.591	0.481	1.2
1000	5.074	4.052	1.3
10000	49.796	35.312	1.4
*** Epil D=8***			
1	0.110	0.066	1.6
10	0.524	0.146	3.6
100	4.262	0.511	8.3
1000	41.494	3.811	10.9
10000	413.546	33.091	12.5
*** WDBC D=30***			
1	0.233	0.143	1.6
10	2.033	0.260	7.8
100	18.891	0.815	23.1
1000	187.477	5.946	31.5
10000	1873.383	50.086	37.4

As part of our tests, we compared the performance and the quality of the maps generated by our parallel HLS-VSOM with two CPU SOM and two GPU SOM variants. The quality of the maps is based on the convergence index as defined in [25]. The CPU single-node tests used the SOM and the VSOM algorithms included as part of the R language *Popsom* package. In addition, the GPU parallel comparison was made using the GPU-based SOM packages Tensorflow 2.0 for XPySom and our own Par-VSOM parallel GPU implementation based on NVIDIA

Table 7: Quality of maps using the convergence index [25] produced by the different training algorithms. (VSM=VSOM, P-V=Par-VSOM, X-P=XPySom, H-P=HLS and D=Dimensions)

Iters $10^x$	CI				
	SOM	VSM	P-V	X-P	H-P
*** Iris, D=4***					
1	0.16	0.15	0.09	0.50	0.35
2	0.43	0.45	0.70	0.50	0.50
3	0.92	0.95	0.91	0.88	0.97
4	0.93	0.94	0.91	0.92	0.95
*** Epil, D=8***					
1	0.14	0.17	0.15	0.72	0.17
2	0.56	0.45	0.52	0.49	0.34
3	0.92	0.92	0.94	0.91	0.92
4	0.94	0.92	0.93	0.65	0.92
*** WDBC, D=30***					
1	0.12	0.15	0.11	0.68	0.16
2	0.23	0.40	0.55	0.53	0.47
3	0.90	0.92	0.88	0.68	0.91
4	0.90	0.92	0.93	0.69	0.92

Thrust.

For our experiments, we used three real-world datasets to train our algorithms:

1. Iris [26] - a dataset with 150 instances and 4 attributes that describes three different species of Iris.
2. Epil [27] - a dataset on two-week seizure counts for 59 epileptics. The data consists of 236 observations with 8 attributes. The data set has two classes - placebo and progabide, a drug for epilepsy treatment.
3. Wisconsin Breast Cancer Dataset (wdbc) [28] - a dataset with 30 features and 569 instances related to breast cancer in Wisconsin. The features are



computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe the characteristics of the cell nuclei present in the image. The data set has two classes: malignant and benign.

These datasets are purposely selected to test the algorithm performance by increasing the dimensionality complexity of the input data. As previously mentioned, Iris has four attributes, Epil eight attributes, and WDBC 30 attributes. This provides significant dimensions variability to test the algorithm. To measure the HLS-VSOM performance, we ran each timing test three times and took the average time over these runs. The times reported are the time required for the CPU to perform the calculations, and it is given in CPU seconds. Similarly, the quality tests were done by averaging three quality measurements using the convergence index (CI) explain in detail in [25] and included as part of the R Popsom Package [23]. The CI provides a 0 to 1 numbering scale to measure the maps' quality, with 0 representing the lowest quality and 1 the highest quality. In addition, we trained with various iterations to discover what type of effect a change of training duration had on the implementations.

### 2.7.3 Results

The following tables includes the experimental results obtained for two different map sizes and three data sets.

In the 150 neurons experimental results, included in Table 5, we see the time comparison and the speed-up gains of the algorithm. The optimization achievable by the HLS FPGA significantly boots the performance when compared against the SOM and VSOM CPU variants. In the maps instances under test, the FPGA provides enough computational resource to construct an efficient design without impacting the algorithm performance. However, designs using larger maps (25,000 neurons) demonstrated that optimization could not be achieved due to FPGA

resources limitations (e.g clock don't meet thresholds, routing logic too complex and global iteration problems making the design unable to be completed).

In addition, as part of the result, is it observable that the VSOM in a CPU trains the given maps in a fraction of a second. Achieving performance increase on this scale is important due to the high demand in the military and aerospace industries for acceleration in real-time operating system applications, where the requirements are defined in terms of minimum seconds and milliseconds.

The results illustrate, the HLS-VSOM achieves a speed-up of not less than 6x on average at the convergence iteration (1000) in comparison to the VSOM. The Table 5 results demonstrates, the HLS-VSOM achieves superior speed-up for all the three datasets comparisons, surpassing the speed rates of all the other algorithm implementations. In this map environment, the HLS-VSOM surpassed the SOM with a 30.4x and the VSOM with a 6.3x when reaching the convergence point as summarized in table 8. The comparison with the GPU version demonstrates the GPU versions are not well suited for regular and small size maps due to the low computational workloads. The performance obtained for the GPU variants were 76.0x for the Par-VSOM and 185.7x for the XPysom.

The training time charts included in Figures 15 - 17, capture a generalize representation of the overall results tendencies. The HLS-VSOM offers speedup performance increases for the three datasets. The obtained results allows us to establish a direct relation between the dimentionalty of neuronal maps and better achievable times using the HLS-VSOM. That is, with more dimentionalty complexity in the dataset a better speed up can be achieved making it scalable.

Table 6 results include the times and Speed-up gains of the HLS-VSOM compared against a non-accelerated FPGA HLS-VSOM using an  $15 \times 10$  map. The proposed accelerated HLS-VSOM uses pipelined loops, dataflow, horizontal un-

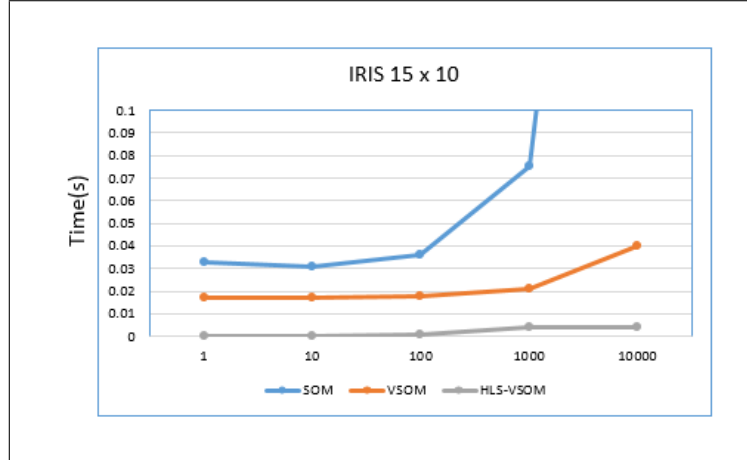


Figure 15: Iris Training Time

rolling, array partitioning, and systolic arrays for row sum reductions allow us to achieved 31.1X performance increase gain when compared with the default Non-accelerated HLS-VSOM. Here, the Non-Accel version refers to running only the default pipeline implemented by the Vitis compiler without any predefined Pragma directives for optimization.

In terms of the quality of the maps, Table 7 captures all the algorithm convergence indexes for the three datasets. As presented, the HLS-VSOM maintains relatively the same quality as the original SOM and the VSOM variants in all the maps.

## 2.8 Conclusions

This work introduced the HLS-VSOM, a high-level synthesis parallel version of the vectorized and matrix-based implementation of stochastic training for self-organizing maps. The novel HLS implementation presented here provides substantial performance increases over Kohonen’s iterative SOM algorithm (up to 30.4X times faster) and the CPU based vectorized VSOM (up to 6.3x times faster). Our comparisons against the GPU variants also demonstrate the optimized FPGA VSOM surpasses the GPU Par-VSOM and XPySom GPUs version by two or three

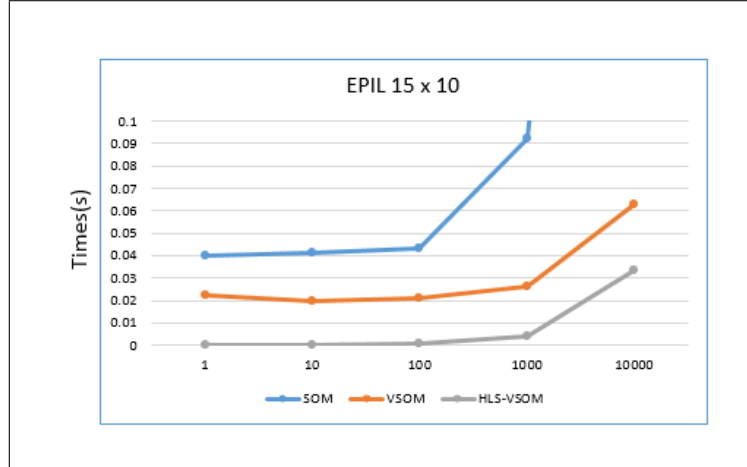


Figure 16: Epil Training Time

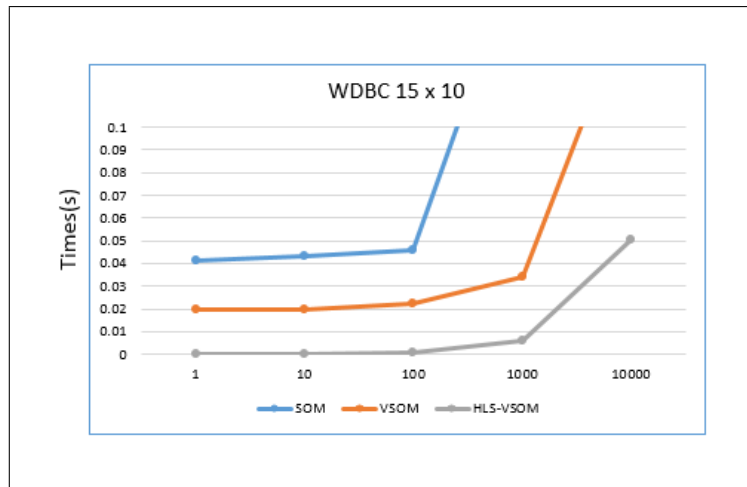


Figure 17: WDBC Training Time

orders of magnitudes of performance in various datasets. The achievable performance gains surpassed all the other architectures implementations and scale exponentially with dimensional increases, as shown in Figure[ 15 - 17 ]. Furthermore, the results demonstrate that the HLS-VSOM provides possibly the best performance SOM currently available. In terms of the quality of the maps, the maps produced by HLS-VSOM approximates the values generated by the VSOM iterative algorithms and original Kohonen's SOM algorithm.

In the proposed design, the HLS-VSOM is a highly optimized algorithm run-

Table 8: HLS-VSOM FPGA Speed-ups Summary

<b>**Dataset**</b>	<b>**MAX**</b>	<b>**@ Convergence**</b>
Speed-up vs SOM-CPU:		
IRIS	961.1	18.5
EPIL	600	24.1
WDBC	286	30.4
<b>**Dataset**</b>	<b>**MAX**</b>	<b>** @ Convergence**</b>
Speed-up vs VSOM-CPU:		
IRIS	495.1	5.2
EPIL	330.0	6.3
WDBC	139.5	5.7
<b>**Dataset**</b>	<b>**MAX**</b>	<b>** @ Convergence**</b>
Speed-up vs Par-V-GPU:		
IRIS	145.6	41.9
EPIL	162.0	58.0
WDBC	125.6	76.0
<b>**Dataset**</b>	<b>**MAX**</b>	<b>** @ Convergence**</b>
Speed-up vs Xpysom-GPU:		
IRIS	267.7	243.3
EPIL	309.5	274.4
WDBC	202.3	185.7

ning in a FPGA Accelerator Card and therefore is an adequate replacement for iterative stochastic training of SOM and parallel SOM variants. Future research on this topic will include investigating how the HLS-VSOM can be implemented in a tensor-core based acceleration environment and what kind of performance increase we can expect from this type of hardware architecture. In the literature, the SOM data partitioning has been used exclusively as the starting point for parallel SOM implementations up to this point, *e.g.* [29, 30]. Given the results reported here, the HLS-VSOM can be viewed as an alternative to parallel SOM and a new alternative starting point for other parallel algorithms for clustering. In summary, since the training algorithms results demonstrate the produce maps are roughly the same

quality, the HLS-VSOM provides a parallel and high-performance alternative to SOM algorithms.

### List of References

- [1] T. Kohonen, *Self-organizing maps*. Springer Berlin, 2001.
- [2] B. Barney, *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory, 2018.
- [3] L. Hamel, *VSOM: Efficient, Stochastic Self-organizing Map Training: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 2*, 01 2019, pp. 805–821.
- [4] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefer, “Transformations of high-level synthesis codes for high-performance computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2020.
- [5] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, “Design productivity of a high level synthesis compiler versus hdl,” in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 140–147.
- [6] C. Rubattu, F. Palumbo, C. Sau, R. Salvador, J. Sérot, K. Desnos, L. Raffo, and M. Pelcat, “Dataflow-functional high-level synthesis for coarse-grained reconfigurable accelerators,” *IEEE Embedded Systems Letters*, vol. 11, no. 3, pp. 69–72, 2018.
- [7] J. Backus, “Can programming be liberated from the von neumann style? a functional style and its algebra of programs,” *Commun. ACM*, vol. 21, no. 8, p. 613–641, aug 1978. [Online]. Available: <https://doi.org/10.1145/359576.359579>
- [8] wiki.com, “Wiki pipelininig,” [https://en.wikipedia.org/wiki/Instruction\\_pipelining](https://en.wikipedia.org/wiki/Instruction_pipelining), accessed: 2021-11-27.
- [9] P. Jaaskelainen, “Task parallelism with opencl: A case study.” *Journal of Signal Processing Systems*, pp. 33–46, 2019.
- [10] L. L. Pilla, “Basics of vectorization for fortran applications,” *Research Report*, vol. RR-9147, pp. 1–9, 2018.
- [11] H. T. Kung, *Systolic Array*. GBR: John Wiley and Sons Ltd., 2003, p. 1741–1743.

- [12] Z. Yang, L. Wang, D. Ding, X. Zhang, Y. Deng, S. Li, and Q. Dou, “Systolic array based accelerator and algorithm mapping for deep learning algorithms,” in *IFIP International Conference on Network and Parallel Computing*. Springer, 2018, pp. 153–158.
- [13] A. Morán, J. L. Rosselló, M. Roca, and V. Canals, “Soc kohonen maps based on stochastic computing,” in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–7.
- [14] M. Abadi, S. Jovanovic, K. B. Khalifa, S. Weber, and M. H. Bedoui, “A scalable flexible som noc-based hardware architecture,” in *Advances in Self-Organizing Maps and Learning Vector Quantization*. Springer, 2016, pp. 165–175.
- [15] N. Paulino, J. C. Ferreira, and J. M. Cardoso, “Optimizing opencl code for performance on fpga: k-means case study with integer data sets,” *IEEE Access*, vol. 8, pp. 152 286–152 304, 2020.
- [16] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, “Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis,” *IEEE Access*, vol. 5, pp. 2747–2762, 2017.
- [17] R. Li, H. Huang, Z. Wang, Z. Shao, X. Liao, and H. Jin, “Optimizing memory performance of xilinx fpgas under vitis,” *arXiv preprint arXiv:2010.08916*, 2020.
- [18] J. de Fine Licht and T. Hoefler, “hlslib: Software engineering for hardware design,” *arXiv preprint arXiv:1910.04436*, 2019.
- [19] M. Masten, E. Tyurin, K. Mitropoulou, E. Garcia, and H. Saito, “Function/kernel vectorization via loop vectorizer,” in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2018, pp. 39–48.
- [20] G. Davidson, “A parallel implementation of the self organising map using opencl,” *University of Glasgow*, 2015.
- [21] R. Mancini, A. Ritacco, G. Lanciano, and T. Cucinotta, “Xpysom: high-performance self-organizing maps,” in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 209–216.
- [22] G. Vettigli, “Minisom,” <https://github.com/JustGlowing/minisom>, 2021.
- [23] L. Hamel, B. Ott, and G. Breard, *popsom: Functions for Constructing and Evaluating Self-Organizing Maps*, 2016, r package version 4.1.0. [Online]. Available: <https://CRAN.R-project.org/package=popsom>

- [24] J. Vesanto and E. Alhoniemi, “Clustering of the self-organizing map,” *IEEE Transactions on Neural Networks*, vol. 11, no. 3, pp. 586–600, 2000.
- [25] L. Hamel, “Som quality measures: An efficient statistical approach,” in *Advances in Self-Organizing Maps and Learning Vector Quantization*. Springer, 2016, pp. 49–59.
- [26] R. A. Fisher, “The use of multiple measurements in taxonomic problems,” *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
- [27] P. F. Thall and S. C. Vail, “Some covariance models for longitudinal count data with overdispersion,” *Biometrics*, pp. 657–671, 1990.
- [28] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, “Nuclear feature extraction for breast tumor diagnosis,” in *IS&T/SPIE’s Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics, 1993, pp. 861–870.
- [29] R. D. Lawrence, G. S. Almasi, and H. E. Rushmeier, “A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems,” *Data Mining and Knowledge Discovery*, vol. 3, no. 2, pp. 171–195, 1999.
- [30] P. Wittek, S. C. Gao, I. S. Lim, and L. Zhao, “Somoclu: An efficient parallel library for self-organizing maps,” *arXiv preprint arXiv:1305.1422*, 2013.



## APPENDIX A

### Introduction and review of the problem

#### A.1 Introduction

The academic, financial, and industrial sectors are highly interested in finding optimal parallelization of unsupervised machine learning algorithms. Nowadays, scientists in atmospheric sciences, nuclear physics, medical diagnosis, and others are looking for more sophisticated ways of processing large amounts of data in their domains, utilizing various parallel computational approaches [1].

The parallel operations make program execution run faster by performing several computations simultaneously in multiple nodes and minimizing the data dependencies between them [2]. Due to this processing advantage, the parallel implementation of the algorithms seems like the next logical step to solving large and more complex problems. In addition to processing large amounts of data, the parallelization of the algorithms could be highly beneficial because it can minimize system operational costs, and the total energy spent [3].

The results of this research are highly significant because, based on the current SOM research literature review, the proposed parallel method is potentially the fastest parallel SOM implementation known in the scientific community. Furthermore, our experimental research approach allows us to enhance the knowledge of the machine learning community by providing answers to the following questions:

- How the Parallel VSOM algorithm can offers superior performance rates in comparison with previously proposed parallel SOM methods?
- Can we generate good quality maps in less computational time than previously proposed parallel SOM methods?

- Which of the Parallel VSOM algorithm programming languages are more convenient in terms of compilation times, coding complexity, and resources required?
- Which hardware accelerator architecture offers us new upper limits in terms of speed-up rates with the Parallel VSOM algorithms?

## A.2 Review of the Problem

During the last decade, machine learning modeling for big data, pattern recognition, prediction analysis, and other applications has continued evolving, becoming an industry standard for analyzing the sheer amount of data dimensionality complexity investigated [4]. As a result, the machine learning community has been focusing their research efforts on obtaining better computational performance and higher speed-up rates to address the algorithms' demands.

The present work is centered on seeking to improve the SOM algorithms by reducing the computational processing time and memory demands required during the execution of unsupervised machine learning applications. In general, the research revolves around investigating a new parallel approach for Kohonen's self-organizing map (SOM) in multiple hardware accelerators environments and mainly focuses on the development of new parallel versions for the vectorized SOM (VSOM) algorithm proposed by Hamel [5]. The parallel VSOM algorithms (Par-VSOM & HLS-VSOM) replace the vector and matrix operations of the original VSOM algorithm with parallel computational kernels executing in hardware-accelerated architectures.

The primary motivation of this research revolves in reducing the high computational times of complex data sets, which results in exceeding the cost for system-specific operations [6]. In order to address the increasing computational

time problem, this research offers improvements to the SOM training and computational time demands by: reducing the total convergence time of the SOM algorithm by utilizing hardware accelerator architectures for execution; creating a parallel versions of the vectorized SOM algorithm that will provide better performance and speed-up gains; discovering the current state of hardware accelerators for SOMs algorithms and identify their potential benefits and limitations.

### A.2.1 The SOM and VSOM Algorithm

In this research, the SOM algorithm is the population under test. The SOM mapping model is based on neural network interactions derived from Vector Quantization (VQ) algorithm method[7]. The VQ is a signal-approximation algorithm that approximates a finite “codebook” of vectors  $m_i \in R^n, i = 1, 2, \dots, k$  to the distribution of the input data vector  $x \in R^n$ . In the SOM context, the approximated codebook allows us to categorize the nodes and form an “elastic network,” which becomes a meaningful, coordinated map or grid system.

From a computational science perspective, the SOM can be described as a mapping of high dimensional input data onto a low dimensional neural network projected as a 2D or 3D map. The mapping is accomplished by assuming the set of input data is a real vector such as  $x = [\xi_1, \xi_2, \dots, \xi_n]^T \in R^n$ . The SOM neuronal map can be defined as a model containing the parametric real vector  $m_i = [u_{i1}, u_{i2}, \dots, u_{in}]^T \in R^n$  associated with the original weights or locations of the neurons. If we consider the distance between the input vector  $x$  and the neuron vector  $m_i$  then we can establish an initial minimum distance relation between the input and the neurons by calculating the euclidean distances and identifying the best matching unit (BMU) array  $m_c$  with

$$c = \operatorname{argmin}\{d(x, m_i)\} \qquad \text{Best Matching Unit}$$

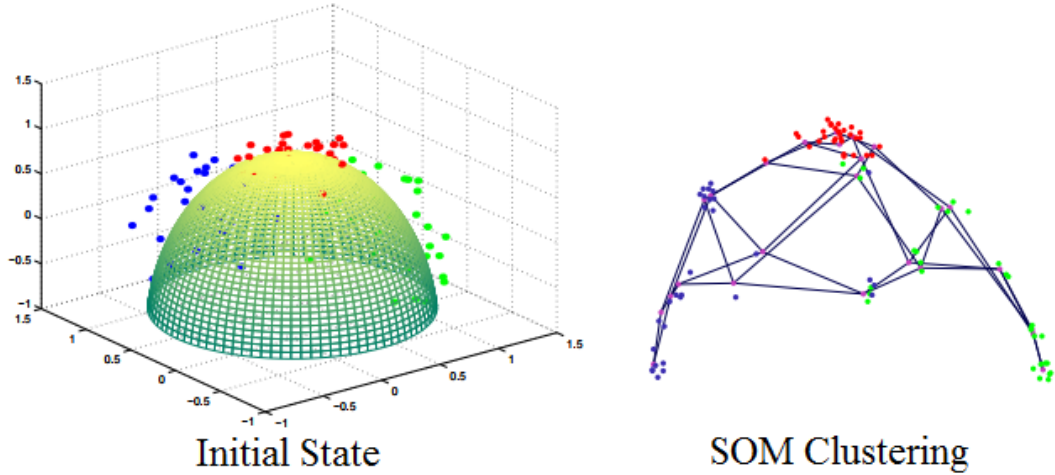


Figure A.18: SOM preserving the neighborhood topology [5]

or

$$\|x - m_c\| = \min\{\|x - m_i\|\}$$

After initializing the neuronal model, we create data neighborhoods  $N_i$  around the neurons  $m_i$  by associating every input  $x(t)$  into a sublist of every  $m_i$  neuron vector. Then, using the  $N_i$  sublist values, we can identify the generalized median  $\bar{X}_i$  or the arithmetic mean of the sublist with the closest distance.

The next step of the algorithm calculates the  $\bar{X}_i$  for the neighborhood and replace the old  $m_i$  with the calculated  $\bar{X}_i$ . This realigns the neuronal map based on the new  $\bar{X}_i$  and counts as the first iteration or epoch. After finishing the first iteration, the algorithm continues the iterative process updating the calculating the distances, learning rate parameters, and generating the  $\bar{X}_i$  until it asymptotically converges. After multiple iterations of the neurons location or weights readjustments, every vector will be assigned or clustered to a specific neuron in the grid, preserving the neighborhood topology as shown in figure 2.

In order to identify how far or close the SOM will be selecting the neighborhood topology boundaries, we used a smoothing kernel definition over the shape of the neuronal network. In the basic SOM, we use an on-line weight method to

update each neuron weight, one by one every time. Where time  $t = 0, 1, \dots$  is an integer,  $x(t)$  is the input vector given to the network,  $h_{ci}(t)$  acts as the neighborhood function and  $\alpha(t)$  is the adaptation gain or learning-rate factor between 0 and 1 ( $0 < \alpha(t) < 1$ ). The learning factor and the neighborhood function radius  $\sigma$  both decreased monotonically over time.

$$m_i(t + 1) = m_i(t) + h_{ci}(t)[x(t) - m_i(t)] \quad \text{Update Step}$$

In contrast, for a parallel SOM implementation, it is preferred to use a batch mode (BatchSOM) learning. In the BatchSOM variant, the whole set of input vectors is known in advance (constant), and the weights values are updated based on this input. Due to the importance of timing when running a parallel instance, the update step times can now be expressed in terms of epochs by  $t = eT + t'$  where  $e = 0, 1, \dots$  is the epoch number,  $T$  is the epoch length, and  $0 < t' < T$  the time running inside an epoch [2]. Here we maintain, the  $h_{ci}$  constant during the epoch, and the weight update for the batch update can be expressed as:

$$m_i(e + 1)T = m_i(eT) + h_{ci} \left( \sum_{t'=0}^{T-1} \Gamma(x(t') - m_i(eT)) \right)$$

BatchSOM update with time terms

where  $\Gamma$  is the smoothing kernel that classifies whether the input vector  $X_i$  belongs to the winner neighborhood or not. In practice, a list of winners and input vectors has to be maintained. “Other modifications exist, in which on-line and batch algorithms are combined for the election of winners or weight updates [2].”

Recent research has introduced more efficient implementations of the SOMs. The vectorized SOM (VSOM) results prove that additional performance gains are obtainable by replacing the classic stochastic SOM iterative construct with vector and matrix operations[13]. In the VSOM constructs, the BMU can be determined

by representing the neuron weights using a  $n \times d$  matrix  $M$  as  $M[i,] = (m_1, \dots, m_d)_i$  and the input data as a  $l \times d$  matrix  $D$  as  $D[k,] = (x_1, \dots, x_d)_k$ . These matrices can be used to generate a vector  $s$  that holds the square of Euclidean distance of each neuron from the data vectors following this equation:

$$\begin{aligned}
 s[i] &= \Delta[i,] \cdot \Delta[i,] \\
 &= \| \Delta[i,] \|^2 \\
 &= \| M[i,] - X[i,] \|^2 \\
 &= \| m_i - x_k \|^2
 \end{aligned}
 \tag{VSOM BMU}$$

In order to make the matrix operations well defined, the  $D$  matrix is used as the basis to generate the  $X$  matrix by using the computation of the outer product of  $X$  with a column vector  $1^n$  as:

$$X = 1^n \otimes X_k$$

and

$$1^n = \underbrace{(1, 1, \dots, 1)^T}_n$$

The VSOM vectorized neighborhood update step also takes advantage of the matrix and vector operations, redefining the update rules as :

$$M[i,]_{new} \leftarrow M[i,]_{current} - \eta \Delta[i,] \circ \Gamma_c \tag{VSOM Update Step}$$

Utilizing the Hadamard product between  $\eta \Delta[i,]$  and  $\Gamma_c$ ; and defining  $\Gamma_c$  as:

$$\Gamma_c[i,] = \begin{cases} 1^{d'} & \text{if } \Gamma(c)[i] = 1, \text{ in neighborhood} \\ 0^{d'} & \text{otherwise } \text{notinneighborhood} \end{cases}$$

---

**Algorithm 5** Stochastic SOM training algorithm.

---

```

1: given:
2:    $\mathbf{m}_i \leftarrow \{\text{neurons for } i = 1, \dots, n\}$ 
3:    $\mathbf{x}_k \leftarrow \{\text{training instances for } k = 1, \dots, l\}$ 
4:    $\eta \leftarrow \{\text{learning rate, } 0 < \eta < 1\}$ 
5:    $h(c, i) \leftarrow \begin{cases} 1 & \text{if } i \in \Gamma(c), \\ 0 & \text{otherwise,} \end{cases}$ 
6:
7: repeat
8:   /*** Select a training instance ***/
9:    $\mathbf{x}_k$  for some  $k = 1, \dots, l$  :
10:
11:  /*** Find the winning neuron ***/
12:   $c \leftarrow 1$ 
13:   $v \leftarrow \|\mathbf{m}_c - \mathbf{x}_k\|^2$ 
14:  for  $i = 2, n$  do
15:     $d \leftarrow \|\mathbf{m}_i - \mathbf{x}_k\|^2$ 
16:    if  $d < v$  then
17:       $c \leftarrow i$ 
18:       $v \leftarrow d$ 
19:    end if
20:  end for
21:
22:  /*** Update neighborhood ***/
23:  for  $i = 1, n$  do
24:     $\mathbf{m}_i \leftarrow \mathbf{m}_i - \eta(\mathbf{m}_i - \mathbf{x}_k)h(c, i)$ 
25:  end for
26: until done
27: return  $\mathbf{m}_i$  for all  $i = 1, \dots, n$ 

```

---



---

**Algorithm 6** The Neighborhood Function  $\Gamma$ .

---

```

1: given:
2:    $c \leftarrow \{\text{index of winning neuron}\}$ 
3:    $n \leftarrow \{\text{the number of neurons on the map}\}$ 
4:    $nsize \leftarrow \{\text{neighborhood radius}\}$ 
5:    $\mathbf{p}_i \leftarrow \{\text{position of } \mathbf{m}_i \text{ on the map with } \mathbf{p}_i = (x_i, y_i)\}$ 
6:    $\mathbf{p}_c \leftarrow \{\text{position of } \mathbf{m}_c \text{ on the map with } \mathbf{p}_c = (x_c, y_c)\}$ 
7:
8: hood  $\leftarrow \{\}$ 
9:
10: for  $i$  in  $1, 2, \dots, n$  do
11:    $d \leftarrow \|\mathbf{p}_i - \mathbf{p}_c\|$ 
12:   if  $d < nsize \times 1.5$  then
13:     hood  $\leftarrow \text{hood} \cup \{i\}$ 
14:   end if
15: end for
16: return hood

```

---

---

**Algorithm 7** The VSOM training algorithm.

---

```

1: given:
2:    $\mathbf{D} \leftarrow \{\text{training instances, a } l \times d \text{ matrix}\}$ 
3:    $\mathbf{M} \leftarrow \{\text{neurons, a } n \times d \text{ matrix}\}$ 
4:    $\eta \leftarrow \{\text{learning rate, } 0 < \eta < 1\}$ 
5:    $\Gamma(c) \leftarrow \{\text{neighborhood function for some neuron } c\}$ 
6:    $\text{minloc}(\mathbf{s}) \leftarrow \{\text{func. returns location of min. val. in } \mathbf{s}\}$ 
7:
8: repeat
9:   /** Select a training instance */
10:   $\mathbf{x}_k \leftarrow \mathbf{D}[k, ]$  for some  $k = 1, \dots, l$ :
11:
12:  /** Find the winning neuron */
13:   $\mathbf{X} \leftarrow \mathbf{1}^n \otimes \mathbf{x}_k$ 
14:   $\mathbf{\Delta} \leftarrow \mathbf{M} - \mathbf{X}$ 
15:   $\mathbf{\Pi} \leftarrow \mathbf{\Delta} \circ \mathbf{\Delta}$ 
16:   $\mathbf{s} \leftarrow \mathbf{\Pi} \times \mathbf{1}^d$ 
17:   $c = \text{minloc}(\mathbf{s})$ 
18:
19:  /** Update neighborhood */
20:   $\mathbf{\Gamma}_c \leftarrow \Gamma(c) \otimes \mathbf{1}^{d'}$ 
21:   $\mathbf{M} \leftarrow \mathbf{M} - \eta \mathbf{\Delta} \circ \mathbf{\Gamma}_c$ 
22: until done
23: return  $\mathbf{M}$ 

```

---



---

**Algorithm 8** The Vectorized Neighborhood Function  $\Gamma$ .

---

```

1: given:
2:    $c \leftarrow \{\text{index of winning neuron}\}$ 
3:    $n \leftarrow \{\text{the number of neurons on the map}\}$ 
4:    $nsize \leftarrow \{\text{neighborhood radius}\}$ 
5:    $\mathbf{P} \leftarrow \{\text{an } n \times 2 \text{ matrix with } \mathbf{p}_i = \mathbf{P}[i, ] = (x_i, y_i)\}$ 
6:    $\mathbf{1}^n \leftarrow \{\text{constant column vector with value 1}\}$ 
7:    $\mathbf{0}^n \leftarrow \{\text{constant column vector with value 0}\}$ 
8:
9:    $\mathbf{p}_c \leftarrow \mathbf{P}[c, ]$ 
10:   $\mathbf{C} \leftarrow \mathbf{1}^n \otimes \mathbf{p}_c$ 
11:   $\mathbf{\Delta} \leftarrow \mathbf{P} - \mathbf{C}$ 
12:   $\mathbf{\Pi} \leftarrow \mathbf{\Delta} \circ \mathbf{\Delta}$ 
13:   $\mathbf{d} \leftarrow \mathbf{\Pi} \times \mathbf{1}^{2'}$ 
14:   $\text{hood} \leftarrow \text{ifelse}(\mathbf{d} < (nsize \times 1.5)^2, \mathbf{1}^n, \mathbf{0}^n)$ 
15: return  $\text{hood}$ 

```

---



### A.2.2 Parallel SOM

The classic parallel SOM model follows an approach similar to the training set parallelism proposed in the classical parallel models, as demonstrated by Hamalinen in [2]. The model will split the total amount of input vectors evenly and distribute its elements across the available computational nodes during the BMU unit calculation for better performance gains and calculate the neuron competitive step in a similar manner.

The design of the BMU portion of the implemented parallel algorithm includes a two-stage parallel reduction. The first goal of the parallel BMU is to find all the distances between neurons and the input vector  $\{d(x, m_i)\}$  and, secondly, identify the BMU in parallel stages. In order to accomplish this, the algorithm acquires the total number of input vectors and the amount of computational units to calculate the parallel chunk size specific for the hardware environment. The parallel chunk-size is calculated using the following equation:

$$PCS = \frac{\text{Vector Units}(Vu)}{\text{ComputationalUnits}(Cu)} \quad \text{Parallel Chunk Size}$$

During the execution, the number of vector units is equivalent to the total of input vectors that will be connected to the neurons on the SOM map and the  $Cu$  represent the computational nodes available on the type of acceleration device. The  $Cu$  should have an equally balanced amount of vector units to process during the parallel iterations. The PCS calculation is utilized to select the correct size of data that will be distributed through each one of the computational nodes available by parallel threads or with parallelization of matrix operations (depending on version of the algorithm under test) in order to acquire the optimal workload for each processing element available.

The Parallel Chunk Size is used to determine the parallelization schema; the

parallel BMU kernel calculates the values for the distance map between all the inputs vectors and the neuron map; and proceeds to find the best matching units (BMU) in parallel as described in Algorithm 9.

---

**Algorithm 9** Parallel Distance Calculation and BMU - SOM
 

---

1.1

```

1: Given:
2:  $Vu \leftarrow$  Amount of Vector Units
3:  $Cu \leftarrow$  Amount of Computational Units
4:  $PCS \leftarrow Vu/Cu$  ▷ Parallel Chunk Size

5: Parallel Threads Instances Size Calculations:
6: /**Max and minimum input vector indexes for parallel instances */
7: for  $pi = 1, Cu$  do
8:   if  $pi > 1$  then
9:      $t_{(pi,min)} \leftarrow t_{(pi-1,max)} + 1$ 
10:     $t_{(pi,max)} \leftarrow pi * PCS$ 
11:   else
12:      $t_{(pi,min)} \leftarrow 1$ 
13:      $t_{(pi,max)} \leftarrow pi * PCS$ 
14:   end if
15: end if
16: end for
17: end for

18: Parallel Distance and BMU:
19: /** Select Parallel Training Instance per Thread */
20:  $X_t$  for some  $t = t_{min}, \dots, t_{max}$ 

21: /** Finding Distance and BMU on each Parallel Instance */
22:  $c \leftarrow 1$  (Init index of BMU)
23:  $v \leftarrow \|x_t - m_c\|^2$ 
24: for  $i = 2, n$  do (In Parallel)
25:    $d \leftarrow \|x_t - m_i\|^2$ 
26:   if  $d < v$  then ▷ Update the BMU index if the distance is less
27:      $c \leftarrow i$ 
28:      $v \leftarrow d$ 
29:   end if
30: end if
31: end for
32: end for

```

---

Additional parallel instructions can be introduced during the SOM calculation to modify the weight of the vectors for the neuron neighborhood locations . This part of the parallel algorithm updates the weight vectors associated to each SOM neuron location using the original update weight and predetermined learning rate. The Algorithm 10 describes the parallel update neighborhood process.

---

**Algorithm 10** Parallel Update Function - SOM

---

```

1
1: Given:
2:  $c \leftarrow$  Index of BMU
3: /** Parallel Instances Max and Min Input Vector Indexes**/
4:  $t \leftarrow$  Max and Min indexes

5: Update Neighborhood:
6: for  $i = 1, n$  do (In Parallel)
7:    $m_i \leftarrow m_i - \eta(m_i - x_t)h(c, i)$ 
8: end for
9: end for

```

---

By providing an initial amount of parallel instances or the numeric quantity of the parallel threads for our architecture, we can generate a parallel function to optimize the neuronal updates calculations. The parallel version of the update step is very similar to the sequential counterpart of the original SOM. The parallel implementation will require the minimum and maximum values of the input vectors for each instance and a parallelization loop instruction to distribute the instances across the processors or nodes for execution.

### A.2.3 Hardware Architectures for Parallel SOMs

Multiple hardware acceleration architectures were utilized during the data collection process. As previously mentioned, the main goal of the research is to establish a comparison between the three acceleration environments and reach a conclusion in terms of SOM's optimal performance and speed-up. For our experimental setup, we utilized a Graphics Processor Unit (GPU) and a Field Pro-

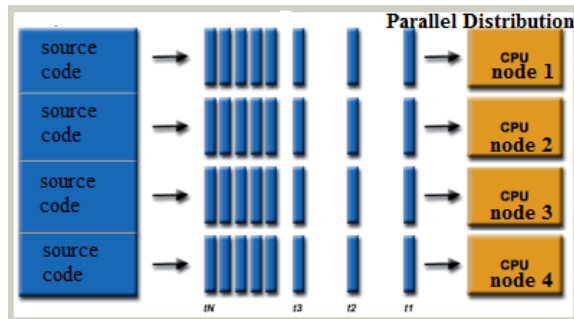


Figure A.19: Parallel Program execution[6]

programmable Gate Array (FPGA) to test our parallel algorithms implementations.

The first type of environment involves the utilization of the high amount of processing cores available in a GPU to obtain significant parallelization and improve the overall algorithm performance. This type of environment allows computer scientists and engineers to execute programs in multiple system threads by splitting (See Figure 3 A.19) Single Instruction, Multiple Data (SIMD) programs and distributing them among the available system nodes for faster processing capability. The GPU architecture (see figure reffig:GPU Grid is a collection of interconnected multiprocessors with high parallelism potential. Each one of the nodes can process its own data independently. The GPU graphic processors and Central Processing Unit (CPU) communicate using a globally shared memory mechanism, allowing them to read, write and modify the same memory space as illustrate in Figure A.21. The shared global memory provides us with more flexibility for process synchronization between the nodes by sharing the same memory windows and scheduling of processing events.

The programming of the processes running in the GPU can be accomplished using the CUDA Application Programming Interface (API). CUDA provides the users with C extensions, hiding the complexity of managing the C language threads and facilitating the parallelization of sequential behavior by using simpler CUDA

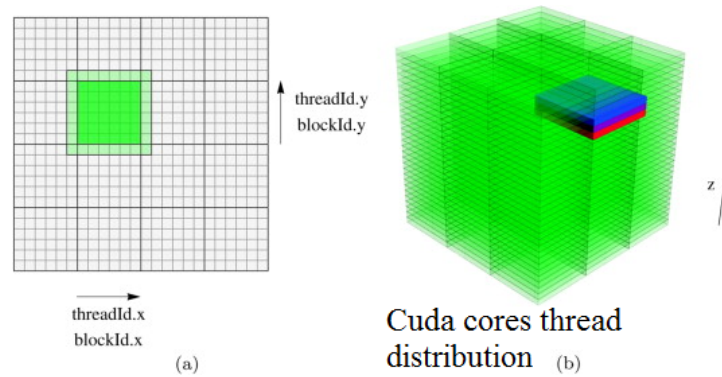


Figure A.20: GPU Grid Diagram [8]

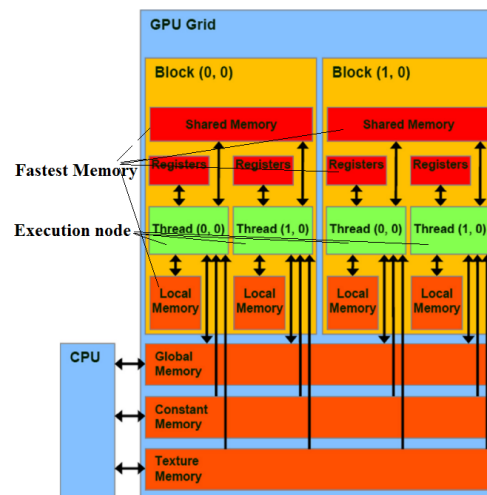


Figure A.21: GPU memory diagram [4]

kernels and CUDA threads function calls. However, the amount of CUDA threads is directly dependent on the specific GPU chip architecture. The programmers should experiment accordingly to obtain peak performance and efficient memory access based on their algorithm design requirements.

The second accelerator environment is the FPGA. The FPGA is a reconfigurable hardware device composed of highly customizable programmable logic blocks (LU), LU Interconnects and reconfigurable I/O Blocks (see Figure ]A.22 FPGA Block Diagram). Industry enthusiasts and academic researchers have

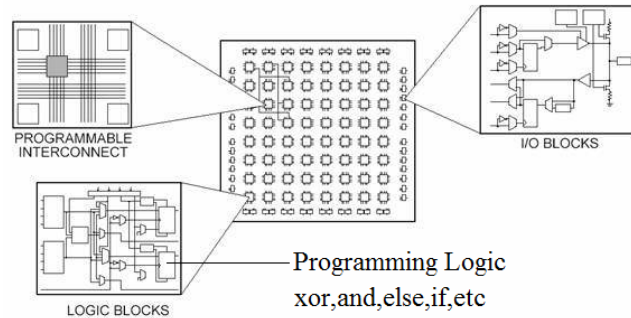


Figure A.22: [FPGA Block Diagram[16]]

demonstrated how the FPGA could implement the requirements of any application using a high-performance ratio (performance to power ratio)[15].

Compared to the CPU and GPUs acceleration environments, the FPGA is a more complex system to program due to the lack of a microprocessor architecture, shared-memory-less architecture, and lower abstraction level perspective. This lower abstraction level of customization allows us to optimize the algorithm processing, data movements, memory access, and system power consumption. Due to the high configuration flexibility available, the FPGA has become one of the most widely used target systems for parallel research, algorithm speed-up, and applications performance in the machine learning field.

#### A.2.4 Parallel Vectorized SOM

As part of our Parallel VSOM implementation testing, we experimented with various types of performance- increasing techniques to obtain optimal results. The techniques included network partitioning, data partitioning [4], manipulating the types of memory used by the kernels, efficient memory management, HLS techniques and asynchronous kernel execution.

The data partitioning technique can be implemented is possible by using multiple processors or SIMD (single instruction multiple data)[9]. Using hardware ac-

celerators with SIMD capability, we can distribute each matrix and vector components into individual threads allowing us to operate on the components in parallel. This vectorization provides performance increases when compared with sequential execution, as illustrated in Figure 3

In addition to data partitioning, we manipulated the type of memory that was used during the kernel execution. By using memory types with higher transfer rates, we can achieve higher speed-up rates and improve the overall performance of the algorithm.

The Parallel VSOM model executes the vectors and matrices operations in the accelerated architecture using kernels. The kernels identified as  $\Phi$  in Algorithm 1 and  $\Omega$  in 3 ,distributes all the components of the vectors and matrices into independent threads for a parallel execution during each training epoch. A secondary set of parallel kernel instructions are introduced to the VSOM algorithm during the neuron neighborhood calculation in the  $\Phi_{\Gamma}$ . This part of the algorithm is executed in a similar manner, using kernels instead of functions and operating in all the matrices components in parallel.

### A.2.5 High Level Synthesis for Parallel SOMs

The Vitis™ HLS is a high-level synthesis tool that allows C, C++, and OpenCL™ functions to become hardwired onto the device logic fabric and RAM/DSP blocks as illustrated in figure A.23. Vitis HLS implements hardware kernels in the Vitis application acceleration development flow and uses C/C++ code for developing RTL IP for Xilinx® device designs in the Vivado® Design Suite [10].

The HLS acceleration serves as an answer to address the complex and error-prone hardware design process. The HLS has been known to cope with these losses, obtaining design productivity gains by separating functional system veri-



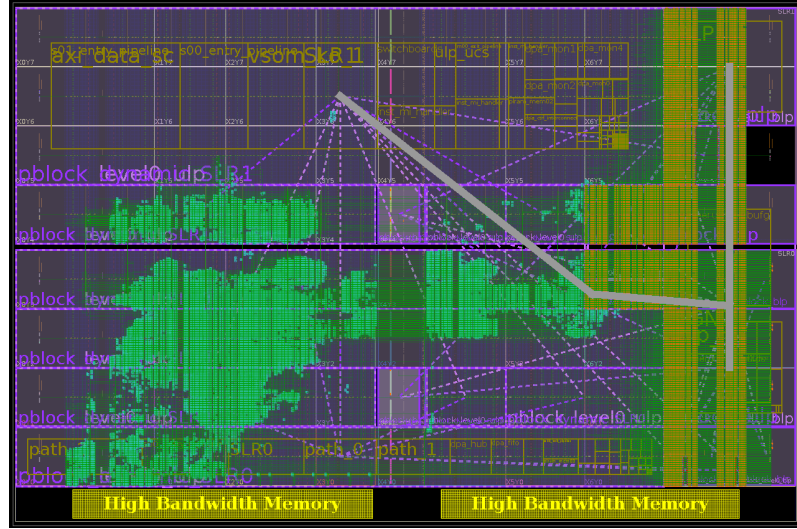


Figure A.23: HLS VSOM in Alveo FPGA

fication, performed from a time-agnostic high-level language, from timed system verification, performed after automatically inferring hardware-specific code [11].

Imperative high-level programming languages imperative formulations can not differentiate between iterations over time and iterations over space. This limitation does not translate appropriately to hardware architecture where all the event are occurring in parallel.

### A.2.6 Systolic Array with HLS

The systolic array is a composed of multiple data processing units (DPUs) called cells or nodes. Each node or DPU operates and execute partial result of the algorithm functions using the data received from the neighbours DPU. In order to maintain the partial result the DPU stores the data within itself and passes to the next DPUs. A multiple networked DPUs executing in parallel can greatly increase the performance of certain math matrix function by executing using multiple element in parallel and store partial results. When partial results are moved between cells, they are computed over these cells in a pipeline fashion.

In this case, the computation of each single output is partitioned over these cells [12].

In our HLS-VSOM algorithm we employ a systolic array approach to execute a matrix “rowsum” reduction operation. The Xilinx compiler transforms the HLS instructions into a Digital Signal Processors (DSP) operations. The Alveo FPGA provides DSPs than act the as independent Processing Elements (PE); communications between the PEs between and input and output for the algorithm will take simultaneously achieving high performance.

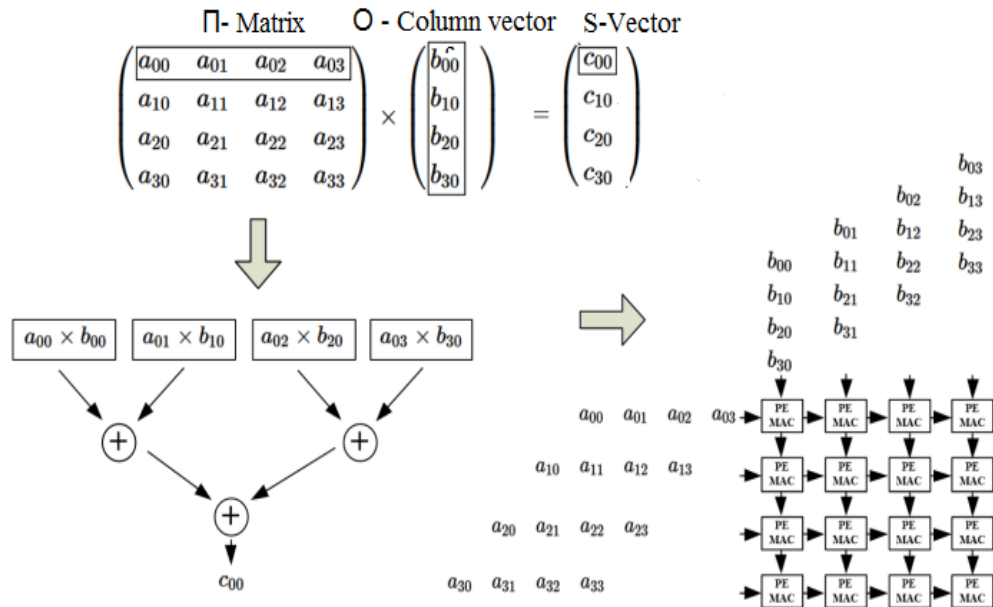


Figure A.24: Systolic Array Matrix Multiplication. [12]

As part of our HLS algorithm development, we discovered one of the major bottleneck was the matrix rowsum reduction included in Algorithm 3 line 23. The latency of this instruction is due to the high amount of read and write access requested to the same memory local BRAM memory locations. Using the systolic Digital Signal Processor (DSP) approach, allow us to access and execute in multiple PE at the same time alleviating the BRAM traffic and increasing the overall

performance. In the algorithm, we use a dot product  $s(\cdot \cdot O)$ . Here contains the square of the differences of the distances calculated during the BME step and  $O$  is a column vector of one. The result is a vector representative of a rowsum matrix reduction.

## List of References

- [1] B. Barney, *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory, 2018.
- [2] Haskell, “Parallelism vs. concurrency,” [https://wiki.haskell.org/index.php?title=Parallelism\\_vs\\_Concurrency&oldid=62377](https://wiki.haskell.org/index.php?title=Parallelism_vs_Concurrency&oldid=62377), accessed: 2018-03-38.
- [3] J. Zhang, *Parallel Computing*. University of Kentucky, 2017, <https://www.cs.uky.edu/jzhang/CSC621/chapter7.pdf>(visited 2016-01-01).
- [4] T. Richardson and E. Winer, “Extending parallelization of the self-organizing map by combining data and network partitioned methods,” *Advances in Engineering Software*, vol. 88, 10 2015.
- [5] L. Hamel, *VSOM: Efficient, Stochastic Self-organizing Map Training: Proceedings of the 2018 Intelligent Systems Conference (IntelliSys) Volume 2*, 01 2019, pp. 805–821.
- [6] E. S. Schabauer, Hannes and T. Weishaupl, “Solving very large traveling salesman problems by som parallelization on cluster architectures,” in *Sixth International Conference on Parallel and Distributed Computer Applications and Technologies PDCAT’ 05*, 2005.
- [7] T. Kohonen, *Self-organizing maps*. Springer Berlin, 2001.
- [8] “Thread Hierarchy in Cuda Programming.”
- [9] P. T. Rauber, Andreas and D. Merkl, “parsom: a parallel implementation of the self-organizing map exploiting cache effects: making the som fit for interactive high-performance data analysis,” in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000*, vol. 6, 2000.
- [10] Xilinx, “Vitis unified software platform documentation application acceleration development ug1393 (v2021.1) july 19, 2021,” [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2021\\_1/ug1393-vitis-application-acceleration.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1393-vitis-application-acceleration.pdf), accessed: 2022-03-10.

- [11] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry, “Design productivity of a high level synthesis compiler versus hdl,” in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. IEEE, 2016, pp. 140–147.
- [12] Z. Yang, L. Wang, D. Ding, X. Zhang, Y. Deng, S. Li, and Q. Dou, “Systolic array based accelerator and algorithm mapping for deep learning algorithms,” in *IFIP International Conference on Network and Parallel Computing*. Springer, 2018, pp. 153–158.

## APPENDIX B

### Methodology and Source code

#### B.1 Methodology

Our research methodology followed a quantitative process to generate multiple numerical comparisons for analysis. The principal methods of data acquisition and measurement included the speed-up and total time to converge.

The speed-up definition is given by Amdahl's Law [1] and it allow us to measure the improvement in execution speed of the Parallel VSOM algorithm in various architectures.

$$S(p_a) = \frac{t_s + t_p}{t_s + \frac{t_p}{c_1 * P_{an} + c_2 * P_{a(n+1)+...}}} \quad \text{Speed-Up}$$

In this equation  $n$  is considered a natural number,  $p_{an}$  represents an identifier of the used processors,  $t_s$  is the estimated amount of time spent by serial operations in one (single serial) processor and  $t_p$  is the estimated amount of time running parallel parts on a single processor. The  $c_n$  are employment coefficients representing the amount of the usage of the processors utilized during the parallel processes. For our test comparison, dividing the amount of time of algorithm execution in architecture A by the time of execution in architecture B, will suffice to provide the speed-up rate for our further analysis. In addition, we utilized the total time of the algorithm during execution but excluded the time of writing the neuron file in the system (hard drive write).

Additional quantitative evaluation methods were used to determine the quality of the maps generated with our proposed algorithm. In our evaluations, we considered the maps' embedding accuracy and the topological quality by using the equations proposed by Hamel in [2]. The embedding accuracy is a measurement

of how closely the neurons appear to be drawn from the same distribution as the training instances. From our perspective, we define a feature as being embedded if its variance and mean appear to be drawn from the same distribution for both the training data and the neurons. The topographic accuracy is a statistical approach used to measure the quality or organization of the neurons on the map. This evaluation method utilizes the BMU and the second BMU for each data instance to quantify how well the map neurons are organized after finishing the neuronal training.

### B.1.1 Research Design

The research design included a group of comparative components focusing on the behavior of different parallel algorithms in multiple hardware platforms. The control group includes the SOM algorithm running in a standard non-parallel single node environment, the VSOM running in CPU, and the Batch SOM running on a GPU establishing the baseline for our two VSOM experimental comparisons.

The experimental group is composed of a set of VSOM parallel algorithms executing in different hardware architecture environments. Table 1 summarizes our proposed research design for multiple experimental settings and our consideration of success criteria.

\* The proposed success criteria expect the Parallel VSOM to provide better  $\overline{Bt}$  and  $\overline{Nt}$  performance results, and superior speed-up rates when compared against the baseline algorithms executing in various hardware architectures. Furthermore, to validate the quality of the produced maps, the topographic  $ta'$  and estimated embedded accuracy  $ea$  [2] of the maps will be calculated using the convergence index  $ci$  for each one of the experimental settings. The Parallel VSOM will be tested using a small, medium, and large maps sizes in addition to training the maps with various datasets to reduce the possibility of datasets bias.

Parallel-VSOM Research Design			
Control Group	Experimental 1	Experimental 2	Success Criteria*
(Baseline) Sequential SOM Intel I7	Parallel VSOM GPU	Parallel VSOM FPGA	$Speedup > 1.0X;$ $\{Exp1_t, Exp2_t\} < Baseline Time$ $ci \approx Baseline Map Quality$
(Baseline) Vectorized SOM Intel I7	Parallel VSOM GPU	Parallel VSOM FPGA	$Speedup > 1.0X;$ $\{Exp1_t, Exp2_t\} < Baseline Time$ $ci \approx Baseline Map Quality$
(Baseline) Parallel BatchSOM in (GPU)	Parallel VSOM GPU	Parallel VSOM FPGA	$Speedup > 1.0X;$ $\{Exp1_t, Exp2_t\} < Baseline Time$ $ci \approx Baseline Map Quality$

Table 1: Parallel-VSOM research design

### B.1.2 Data Sets

As part of the research, two different sample types will be considered:(1) The data sets selection used for the Parallel-VSOM experimentation and (2) the architectural environments used to test the algorithms (See Hardware Architecture for Parallel SOMs section for details).

The possibility of data sampling bias will be reduced by selecting datasets from multiple domains. The selection will be representative of datasets commonly used by the academic community for machine learning clustering experiments. The following list provides details of the dataset selection that will be utilized for our experimental tests:

- Iris [3]-a dataset with 150 instances and 4 attributes that describe three

different species of iris.

- Epil [4]-a dataset on 2-week seizure counts for 59 epileptics. The data consists of 236 observations with 8 attributes. The dataset has two classes: placebo and progabide, a drug for epilepsy treatment.
- Wisconsin Breast Cancer Dataset (wdbc) [5]-a dataset with 30 features and 569 instances related to breast cancer in Wisconsin. The features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe the characteristics of the cell nuclei present in the image. The data set has two classes: malignant and benign.

## B.2 Readme File

The following Readme file contains information related to the experimentation and tools utilized during the Par-VSOM and HLS-VSOM testing. The files illustrate examples and screen capture of the script and the parameters used.

The Parallel-Vectorized Self-Organizing Maps is a software implementing Kohonen’s self-organizing maps with a number of distinguishing features: A very efficient, parallel, stochastic training algorithm based on ideas from tensor algebra. The new algorithm is implemented using parallel kernels on GPU hardware accelerators. It provides performance increases over the original VSOM algorithm, Py-Torch Quicksom parallel version, Tensorflow Xpysom parallel variant, as well as Kohonen’s classic SOM iterative implementation. In this research we develop the algorithm in some detail and then demonstrate its performance on several real-world datasets. We also demonstrate that our new algorithm does not sacrifice map quality for speed using the convergence index quality assessment.

I. Datasets Information: For our experiments we used three real-world datasets to train our algorithms:



1. Iris - a dataset with 150 instances and 4 attributes that describes three different species of Iris.
2. Epil - a dataset on two-week seizure counts for 59 epileptics. The data consists of 236 observations with 8 attributes. The dataset has two classes - placebo and progabide, a drug for epilepsy treatment.
3. Wisconsin Breast Cancer Dataset (wdbc) – a dataset with 30 features and 569 instances related to breast cancer in Wisconsin, for our experiment we generated a random normalized sample of 100 instances. The dataset has two classes: malignant and benign.

The datasets are available in the following links:

1. Iris:<https://archive.ics.uci.edu/ml/datasets/iris>
2. Epil:<https://stat.ethz.ch/R-manual/R-devel/library/MASS/html/epil.html>
3. WDBC: [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic))

II. External Software Packages: This research utilized various software package and programming language to test and generate our experimental results. The following list provides brief details and the links to find additional information or download the open source packages.

1. R Popsom 4.0.1– For the CPU SOM, VSOM and Convergence Index experiments we utilized the Popsom package version 4.0.1. Using the 4.0.1 version is very important because it allow us to maintain a constant learning through the training iterations. During our experiment, we confirmed the constant learning rate allows the medium and large sizes maps to reach

higher convergence indexes. Note: we included an script to generate vsom and som in each one of the dataset folders. The R package is available @:<https://github.com/lutzhanel/popsom/releases/tag/4.0.1>

2. Xpysom- A high performance Self-Organizing Maps implemented in Python/TensorFlow. This package uses the batch SOM parallel to gain performances in a GPU environment. The Xpysom package is available @:<https://github.com/Manciukic/xpysom>
3. Quicksom-A GPU implementation of the Self-Organizing Maps running in Python/PyTorch The Quicksom package is available @:<https://github.com/bougui505/quicksom>

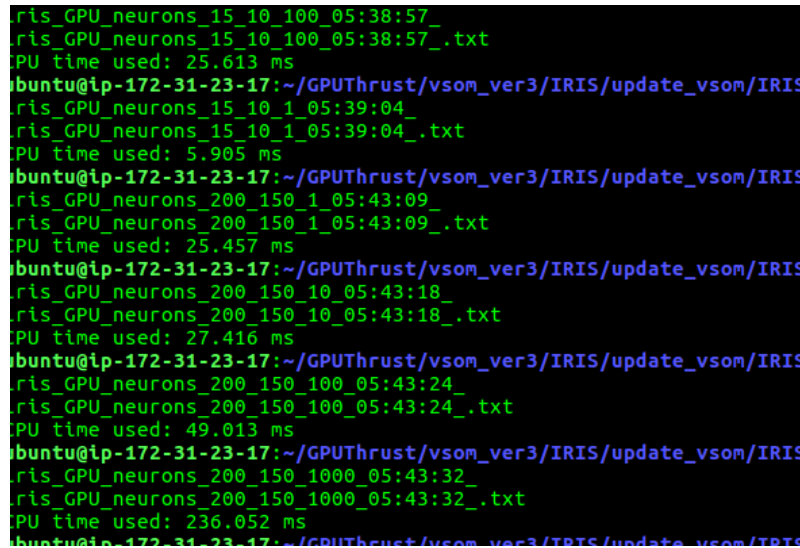
III. Par-VSOM Source code details: The source code for the Par-VSOM experiments is included under the datasets folder structure. The Cuda/Thrust source code for the Par-VSOM and the R-scripts for the convergence index analysis are included in their respective folders (IRIS, EPIL, WDBC).

#### IV. Experimental Environment Setup:

1. System Setup Details: For all of our experiment we set the NVIDIA GPU clocks to the following to the following setting (need sudo access and nvidia-smi installed)
  - a. NVIDIA V100 clock setup : `sudo nvidia-smi -ac 877,1530 -i 0`
2. How to Compile- Par VSOM in CUDA Compilation in AWS GPU system:
  - a. First, navigate to the directory containing the Par-vsom source code e.g. `/home/ubuntu/GPUCodeDirectory/`
  - b. Execute the nvidia compiler using the following command in the shell : `nvcc -o par_vsom par_vsom_source.cu` (that is `nvcc -o executable source.cu`)

3. How to run experiments command: a. Run the Par-vsom training with the corresponding arguments (size of x, size of y , number of iterations) :

```
./par_vsom 15 10 10000
```

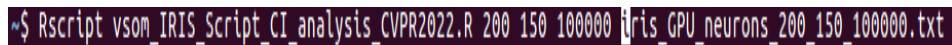


```
iris_GPU_neurons_15_10_100_05:38:57_
iris_GPU_neurons_15_10_100_05:38:57_.txt
GPU time used: 25.613 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
iris_GPU_neurons_15_10_1_05:39:04_
iris_GPU_neurons_15_10_1_05:39:04_.txt
GPU time used: 5.905 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
iris_GPU_neurons_200_150_1_05:43:09_
iris_GPU_neurons_200_150_1_05:43:09_.txt
GPU time used: 25.457 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
iris_GPU_neurons_200_150_10_05:43:18_
iris_GPU_neurons_200_150_10_05:43:18_.txt
GPU time used: 27.416 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
iris_GPU_neurons_200_150_100_05:43:24_
iris_GPU_neurons_200_150_100_05:43:24_.txt
GPU time used: 49.013 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
iris_GPU_neurons_200_150_1000_05:43:32_
iris_GPU_neurons_200_150_1000_05:43:32_.txt
GPU time used: 236.052 ms
ubuntu@ip-172-31-23-17:~/GPUThrust/vsom_ver3/IRIS/update_vsom/IRIS
```

Par-VSOM Executing in Shell screenshot

This means, execute the par\_vsom with a grid of 15x10 with 10000 iterations. At the end of the run, the screen will display the total run time and the name of the map (neuronal weights) file created. Figure 1- Executing Par-VSOM command with arguments

4. How to run convergence index (CI) quality test: These tests require to have the R language installed, the R-script command available in the shell and the Popsom package 4.0.1 to run all the library files as expected. a. The script can be executed using an R-script running the following command



```
~$ Rscript vsom_IRIS_Script_CI_analysis_CVPR2022.R 200 150 100000 iris_GPU_neurons_200_150_100000.txt
```

Par-VSOM Executing in R screenshot

Figure 2- Rscript command with arguments The format of the command is : Rscript “name of the script” arg1 arg2 arg3 arg4 For our application: arg1

= X size arg2 = Y size arg3 = number of iterations arg4 = name of the file to be analyzed. If everything works as intended, the output will contain the CI for CPU and CI for GPU. The idea of the script is to generate a CPU CI that we can compare with our GPU CI to validate quality.

```
[1] "X: 200 Y: 150 Iters: 100000"
[1] "Total map Embedding in CPU is 1"
[1] "Total map Accuracy in CPU is 0.98"
[1] "Total map convergence in CPU is 1"
[1] "Total accuracy in GPU is 0.986666666666667"
[1] "Total embedding in GPU is 1"
[1] "The map Convergence Index is in GPU : 0.993333333333335"
```

### Par-VSOM Results

Figure 3- R-script Convergence Index output

5. How to configure the Xpysom commands: These tests require to have Python and Tensorflow available in the system environment prior to run. Xpysom also needs to be previously installed using the Python PIP tool. Since Xpysom runs on top of Python, we need to specify the argument to be used during the script execution.

```
#Som parameters
som = XPySom(200, 150, 4, learning_rate=0.7, learning_rateN=0.7, decay_function='linear', neighborhood_function='bubble')
```

### Xpysom Parameters

Figure 4- XPySom SOM creation with parameters Here, we are creating a 200 x 150 map, with 4 dimensions, constant learning rate, linear decay and bubble neighborhood. The script includes the rest of the python code to generate a map, with this types of argument. The XpySom function provides additional flexibility and offer multiple neighborhood functions to create maps.

6. How to run Quicksom commands: Quicksom require to have Python and Py-torch available in the system environment prior to running. Also, Quicksom

can be invoked as a Python library. Since Quicksom runs on top of Python, we need to specify the arguments that will be used during the script execution. The Quicksom package interface, does not permit the users to manipulate the learning rate. Due to our experimental setup, we were required to modify the included SOM library to set the learning rate to constant inside som.py. The fix included the following modification: From : `alpha_op = self.alpha * learning_rate_op` To: `alpha_op = self.alpha` In addition, to configure the Quicksom parameters for the som, we specify the values inside the python script:

```
# Create SOM object and train it,
m, n = 15, 10
alpha = 0.7
print(X.shape)
dim = X.shape[1]
niter = 1
batch_size = 150
```

Quicksom Parameters

Figure 5 - Quicksom SOM configuration.

7. How to run som/vsom commands: The vsom and som are generated using R scripts. Each one of the dataset folders includes a script to generate a vsom or som (e.g. vsom\_or\_som\_iris.R). Replacing the algorithm parameter “vsom,som”, allows the user to select the type of som. The script allows the users to modify the properties of the algorithm:

Table B.10: Algorithm Parameters.

<b>**code**</b>
ms = map.build(data,..
Learning Rate $\eta$
xdim = 15,
ydim = 10,
alpha = 0.7,
train = 100000,
algorithm="vsom")
end_time = Sys.time()
delta_time = end_time - start_time
print(delta_time)
print(map.convergence(ms,verb=TRUE))

### B.3 Source Code

Under this section we provide an example of the source code for the PAR-VSOM CUDA Thrust source and the HLS-VSOM Opencl Driver, Kernels and configuration files.

#### B.3.1 Par-VSOM Cuda Kernel

The Par-VSOM CUDA Thrust Code

```
// iris-par-vsom.cu/
// version 3.0
// Author(s):Omar X. Rivera Morales
//
// This file constitutes a set of routines which are useful in constructing
// and evaluating self-organizing maps (SOMs)in a GPU environment.

// The application allows the user to define the size of the maps and the number
//iterations as part of the arguments:

//Usage: vsom_executable.exe [X_size]... [Y_size]... [Number_iter]...
//assuming we want a 150 x 100 amps with a 100 iterations we run:

//vsom_executable.exe 150 100 1000

// License
```

```

// This program is free software; you can redistribute it and/or modify it under
// the terms of the GNU General Public License as published by the Free Software
// Foundation.

// This program is distributed in the hope that it will be useful, but WITHOUT ANY
// WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public License for more details.
//
// A copy of the GNU General Public License is available at
// https://www.gnu.org/licenses

//Load Libraries
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <thrust/iterator/zip_iterator.h>
#include <thrust/random.h>
#include <thrust/functional.h>
#include <iostream>
#include <fstream>      // std::ifstream
#include <string>
#include <sstream>
#include <stdio.h>
#include <stdlib.h>     /* srand, rand */
#include <time.h>      /* time */
#include <ctime>
#include <cmath>
#include <algorithm>   // std::max
#include <vector>
#include <omp.h>

//For random sample
#include <random>
#include <algorithm>
#include <iterator>

/// Prototypes////////////////////////////////////

// We'll use a 2-tuple to store our 2d vector coordinates types
typedef thrust::tuple<float ,float> Float2;

// This functor implements the RowSum for 2d vectors
struct RowSum_2d : public thrust::unary_function<Float2 ,float>
{
    __host__ __device__
    float operator()(const Float2& a) const
    {

```

```

        return thrust::get<0>(a) + thrust::get<1>(a);
    }
};

// This functor implements hood function
struct hood_func
{
    int nei_size;
    hood_func(int n_s) : nei_size(n_s) {};
    __host__ __device__
    float operator()(float x) const
    {
        if ( sqrt(x) < nei_size * 1.5)
            {return 1;}
        else
            {return 0;}
    }
};

//This functor implements the update M (updates the neuron weights)
struct update_m_functor
{
    template <typename Tuple>
    __host__ __device__
    void operator()(Tuple t)
    {
        thrust::get<3>(t) = thrust::get<3>(t) - (thrust::get<1>(t) *
            (thrust::get<0>(t))) * thrust::get<2>(t) ;
    }
};

//This functor implements find the winning neuron (BMU)
struct find_bmu_functor
{
    template <typename Tuple>
    __host__ __device__
    void operator()(Tuple t)
    {
        thrust::get<2>(t) = pow(thrust::get<0>(t) - thrust::get<1>(t),2) ;
    }
};

// We'll use a 4-tuple to store our 4d vector type
typedef thrust::tuple<float ,float ,float ,float> Float4;
// This functor implements the RowSum for 4d vectors
struct RowSum_4d : public thrust::unary_function<Float4 ,float>
{
    __host__ __device__

```



```

    float operator()(const Float4& a) const
    {
        return thrust::get<0>(a) + thrust::get<1>(a) +
            thrust::get<2>(a) + thrust::get<3>(a) ;
    }
};

//This functor executes the PI and Delta transforming the coordinates
struct find_nei_functor
{
    template <typename Tuple>
    __host__ __device__
    void operator()(Tuple t)
    {
        thrust::get<2>(t) = pow(thrust::get<0>(t) - thrust::get<1>(t),2) ;
    }
};

//Function Definitions//////////////////////////////////////

// Return a host vector with random values in the range (0,1) for the Initial
//weights
thrust::host_vector<float> random_vector(const size_t N,
    unsigned int seed = thrust::default_random_engine::default_seed)
{
    thrust::default_random_engine rng(seed);
    thrust::uniform_real_distribution<float> u01(0.0f, 1.0f);
    thrust::host_vector<float> temp(N);
    for(size_t i = 0; i < N; i++) {
        temp[i] = u01(rng);
    }
    return temp;
}

////Main program sequence of the Par-VSOM

//the executable receives 3 argumentes (x size , y size , number of iters)

int main(int argc, char *argv[])
{
    int x_val;
    x_val=atoi(argv[1]);
    int y_val;
    y_val=atoi(argv[2]);
    int iters_val;
    iters_val=atoi(argv[3]);

    int N;
    int new_n = x_val * y_val;
    N = new_n;

    //Init the GPU before start processing

```

```

cudaFree(0);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Init Clocks to measure total time-elapsed
std::clock_t c_start = std::clock();
std::clock_t c_end ;

//Define device (GPU) vector to hold training data (4 columns)
thrust::device_vector< float > file_d_0;
thrust::device_vector< float > file_d_1;
thrust::device_vector< float > file_d_2;
thrust::device_vector< float > file_d_3;

//Define device iterators to create tuple (4 columns matrix)
typedef thrust::device_vector<float >::iterator      FloatIterator_d;
typedef thrust::
tuple<FloatIterator_d, FloatIterator_d, FloatIterator_d, FloatIterator_d>
FloatIteratorTuple_d;
typedef thrust::zip_iterator<FloatIteratorTuple_d>      Float4Iterator_d;

srand((unsigned)time(NULL));

//Vectors declaration:

//Trainig vector D (l x d)
thrust::host_vector< float > file_vector_d;

//Neuron vector M (n x d)
thrust::device_vector<float> M_vector_d = random_vector(N*4,rand() % 100000 + 1);

//Vector X (Holds the selected training instance)
// 'X' vector N*4=size, and init to 0
thrust::device_vector<float> X_vector_d(N*4,0);

//Vector Delta 'D'
// 'D' vector 'Delta' N*4=size, and init to 0
thrust::device_vector<float> D_vector_d(N*4,0);

//Vector PI 'P'
// vector 'P' vector N*4=size, and init to 0
thrust::device_vector<float> P_vector_d(N*4,0);

//The 'S' vector N=size, and init to 0
thrust::device_vector<float> S(N,0);

//Vector Pc 'P-coordinates' for gamma function
//vector Pc N*2, coordinates for BMU x and y in one vector
thrust::device_vector<float> P_coors(N*2,0);

//vector is a merger of all coordinate x and y vector N*2 =size, and init to 0
thrust::device_vector<float> P_coors_all(N*2,0);

```

```

//Vector PI in coordinates (gamma)
//vector to contain the PI coordinates
thrust::device_vector<float> PI_coors(N*2,0);

//Vector D-coor-d (holds all the distances)
// n_0 components of the 'D-coor' vector N=size, and init to 0
thrust::device_vector<float> D-coor-d(N,0);

//Vector hood to nei (gamma)
//vector to contain the nei (1,0)
thrust::device_vector<float> hood_vec-d(N,0);

//vector to contain the nei (1,0)
thrust::device_vector<float> hood_vec-d-dim(N*4,0);

//Mask vector in GPU
thrust::device_vector<float> mask_vec-d(N,0);

//Neighborhood cache (Huge vector containing the nei for each instance)
thrust::device_vector<float> nei_cache(new_n * N,0);

//Vector for Eta values
thrust::device_vector<float> Eta_d(N*4,0);
// Fill Eta vector with eta constant value
thrust::fill(Eta_d.begin(), Eta_d.end(), 0.7);

//Create P Matrix vectors ( holds the coordinates)
// n_0 (x) components of the 'P-coor' vector N=size, and init to 0
thrust::device_vector<float> P-coor0(N,0);
// n_1 (y) components of the 'P-coor' vector N=size, and init to 0
thrust::device_vector<float> P-coor1(N,0);

//Init P Matrix with sequence for calculations (holds the coordinates)
thrust::sequence(P-coor0.begin(), P-coor0.end());
thrust::sequence(P-coor1.begin(), P-coor1.end());

//Create Operator Matrix for coordinate calculation
// fill with x size values to calculate coordinate system
thrust::device_vector<float> Oper_row(N);
thrust::fill(Oper_row.begin(), Oper_row.end(), x_val);

thrust::device_vector<float> Oper_col(N);
thrust::fill(Oper_col.begin(), Oper_col.end(), x_val);

//Calculate X coordinates
thrust::transform(P-coor0.begin(), P-coor0.end(), Oper_row.begin(), P-coor0.begin(),
thrust::divides<int>());

//Calculate y coordinates
thrust::transform(P-coor1.begin(), P-coor1.end(), Oper_col.begin(), P-coor1.begin(),
thrust::modulus<int>());

```

```

//Init of vector for coordnate merge in big vector
P_coors_all.resize(P_coors.size());
thrust::copy(P_coor0.begin(),P_coor0.end(),P_coors_all.begin());
thrust::copy(P_coor1.begin(),P_coor1.end(),P_coors_all.begin()+N);

//Vectors for C Matrix
// n_0 components of the 'C' vector N=size, and init to 0
thrust::device_vector<float> C0_coor(N,0);
// n_1 components of the 'C' vector N=size, and init to 0
thrust::device_vector<float> C1_coor(N,0);

//Define device iterators to create tuple coordinates
typedef thrust::device_vector<float>::iterator
FloatIterator_c;
typedef thrust::tuple<FloatIterator_c, FloatIterator_c>
FloatIteratorTuple_c;
typedef thrust::zip_iterator<FloatIteratorTuple_c>
Float2Iterator_c;

//Iterator for Row2 coordinates
Float2Iterator_c first_PI_coor = thrust::make_zip_iterator
(thrust::make_tuple(PI_coors.begin(),
PI_coors.begin() + N));
Float2Iterator_c last_PI_coor = thrust::make_zip_iterator
(thrust::make_tuple(PI_coors.begin()
+ N,PI_coors.end()));

// Create iterator for C Matrix (type Float2Iterator)
Float2Iterator_c first_C = thrust::make_zip_iterator
(thrust::make_tuple(C0_coor.begin(),C1_coor.begin()));
Float2Iterator_c last_C = thrust::make_zip_iterator
(thrust::make_tuple(C0_coor.end(),C1_coor.end()));

//Create Delta_coor Matrix //////////////////////////////////////
////////////////////////////////////
// n_0 components of the 'C' vector N=size, and init to 0
thrust::device_vector<float> D0_coor(N,0);
// n_1 components of the 'C' vector N=size, and init to 0
thrust::device_vector<float> D1_coor(N,0);

// Create iterator for D Matrix (type Float2Iterator)
Float2Iterator_c first_D_coor = thrust::make_zip_iterator
(thrust::make_tuple(D0_coor.begin(),D1_coor.begin()));
Float2Iterator_c last_D_coor = thrust::make_zip_iterator
(thrust::make_tuple(D0_coor.end(),D1_coor.end()));

//Create PI_coor Matrix //////////////////////////////////////
////////////////////////////////////
// n_0 components of the 'Pi_coor' vector N=size, and init to 0
thrust::device_vector<float> P0_coor(N,0);
// n_1 components of the 'Pi_coor' vector N=size, and init to 0
thrust::device_vector<float> P1_coor(N,0);

```

```

Float2Iterator_c first_P_coor = thrust::make_zip_iterator
(thrust::make_tuple(P0_coor.begin(),P1_coor.begin()));
Float2Iterator_c last_P_coor = thrust::make_zip_iterator(
thrust::make_tuple(P0_coor.end(),P1_coor.end()));

//Create D_coor Matrix ////////////////////////////////////////
//////////////////////////////////////
// n_0 components of the 'D_coor' vector N=size, and init to 0
thrust::device_vector<float> D_coor(N,0);

//Create hood vector
// n_0 components of the 'D_coor' vector N=size, and init to 0
thrust::device_vector<float> hood_vec(N,0);

//Iterator of M Matrix (Random Neuron Weights) for debug
Float4Iterator_d first_M1= thrust::make_zip_iterator(thrust::make_tuple
(M_vector.d.begin(),M_vector.d.begin() + (N * 1),M_vector.d.begin() +
(N * 2),M_vector.d.begin() + (N*3)));
Float4 m_1 = first_M1[0];

//std::cout << "Print out M Matrix (Random Weights)" << std::endl;
//for(size_t i = 0; i < N; i++)
// {
//     m_1 = first_M1[i];
//     std::cout << "(" << thrust::get<0>(m_1) << ","
<< thrust::get<1>(m_1) << "," << thrust::get<2>(m_1) << ","
<< thrust::get<3>(m_1) << ")" << std::endl;
// }

//Random variable selection for Xk (use size of dataset + 1)
int v1 = rand() % 149 + 1;

//Initialize the Matrix size to read Matrix D Data
float file_mat[150][4];

//read Data file and load into cpu array
std::ifstream reader("iris.data");
if (!reader)
std::cerr << "Error_opening_file";
else
{
    for (int i = 0; i < 150; i++)
    { for (int j = 0; j < 4; j++)
        {
            reader >> file_mat[i][j];
            reader.ignore();
        }
        file_d_0.push_back(file_mat[i][0]);
        file_d_1.push_back(file_mat[i][1]);
        file_d_2.push_back(file_mat[i][2]);
        file_d_3.push_back(file_mat[i][3]);
    }
}

```

```

    }

    //load data from cpu array into device (GPU) vector
    for ( int j = 0; j < 4; j++)
    { for (int i = 0; i < 150; i++)
      {
        file_vector_d.push_back(file_mat[i][j]);
      }
    }

//Define neurons file and other variables before epocs loop
std::ofstream neuronsFile;

//Init train iterations
int train = iters_val;

//Init index of c as 0
int c_index = 0;

//Declare x,y as int
int x,y;

//Declare tuple
thrust::pair<thrust::device_vector<float>::iterator ,thrust::device_vector<float>::
iterator> tuple_v;

//Variables for neighborhood calculations
int max_val = max(x_val ,y_val);
int nei_size = max_val + 1;
float temp_val = (float)train/nei_size;
int nei_step = ceil((float)temp_val);
int nei_counter = 0;

if (nei_step == 0)
{
  nei_step = 1;
}

//Declate vector D iterators for Rowsum reduction
Float4Iterator_d vector_first_D = thrust::make_zip_iterator(thrust::
make_tuple(P_vector_d.begin(),P_vector_d.begin() + (N * 1),P_vector_d.begin()
+ (N * 2),P_vector_d.begin() + (N*3)));
Float4Iterator_d vector_last_D = thrust::make_zip_iterator(thrust::
make_tuple(P_vector_d.begin() + (N * 1), P_vector_d.begin() + (N * 2) ,P_vector_d.begin()
+ (N * 3),P_vector_d.begin() + (N*4) ));

//Par-VSOM Main training "Epocs" loops////////////////////////////////////
for (int epocs = 0 ; epocs < train; epocs++)
{

```

```

//Verify if we need to reduce neighborhood size//////////
nei_counter = nei_counter + 1;
if (nei_counter == nei_step)
{
    nei_counter = 0;
    nei_size = nei_size - 1;

    //Clear the masking cache array
    thrust::fill(mask_vec_d.begin(), mask_vec_d.end(), 0);
}
//////////

//Start of Best Matching Units Search (BMU)//////////
//Select random training instance index
v1 = rand() % 149 + 1;

//Fill vector X with Xk values  $X \leftarrow 1^n * Xk$ 
thrust::fill(X_vector_d.begin(), X_vector_d.begin() + (N * 1), file_vector_d[v1]);
thrust::fill(X_vector_d.begin() + (N * 1), X_vector_d.begin() +
(N * 2), file_vector_d[150 * 1 + v1]);
thrust::fill(X_vector_d.begin() + (N * 2), X_vector_d.begin() +
(N * 3), file_vector_d[150 * 2 + v1]);
thrust::fill(X_vector_d.begin() + (N * 3), X_vector_d.begin() +
(N * 4), file_vector_d[150 * 3 + v1]);

//Calculate delta vector :
thrust::transform(M_vector_d.begin(), M_vector_d.end(), X_vector_d.begin(),
D_vector_d.begin(), thrust::minus<float>());

//Calculate PI vector:
thrust::transform(D_vector_d.begin(), D_vector_d.end(), D_vector_d.begin(),
P_vector_d.begin(), thrust::multiplies<float>());

//Calculate S Vector with Rowsum reduction and store in S vector
thrust::transform(vector_first_D, vector_last_D, S.begin(), RowSum_4d());

//Find minimum
tuple_v = thrust::minmax_element(S.begin(), S.end());
c_index = (tuple_v.first - S.begin());

//Done with BMU//////////

//Check if neighborhood is cached
if (mask_vec_d[c_index] == 1)
{
    thrust::copy(nei_cache.begin() + (new_n * c_index), nei_cache.begin() +
(new_n * c_index) + N, hood_vec_d.begin());
}

```

```

//if not in cache Calculate the nei
else
{
    mask_vec_d[c_index] = 1;

//Start of Update neighborhood ////////////////////////////////////////

//Gamma Function /////

//Init PC x
x = P_coor0[c_index];

//Init PC y
y = P_coor1[c_index];

//Fill Matrix C with index coordinates  $X \leftarrow 1^m * P_c$ 
thrust::fill(P_coors.begin(), P_coors.begin()+ N, x);
thrust::fill(P_coors.begin()+ N, P_coors.end(), y);

//Apply the find coordinates neighborhood functor transformation (PI and Delta) to
the coordinates vector (vector with X and Y)
thrust::for_each(thrust::make_zip_iterator(thrust::make_tuple(P_coors_all.begin(),
P_coors.begin(), PI_coors.begin())),
                thrust::make_zip_iterator(thrust::make_tuple(P_coors_all.end(),
P_coors.end(), PI_coors.end())),
                find_nei_functor());

//Finally, we pass the zip_iterators into transform() as if they
// were 'normal' iterators for a device_vector<Float2>.
thrust::transform(first_PI_coor, last_PI_coor, D_coor_d.begin(), RowSum_2d());

//Calculate the hood using the hood_func
thrust::transform(D_coor_d.begin(), D_coor_d.end(), hood_vec_d.begin(), hood_func(nei_size));

//Copy calculated hood in the cache
thrust::copy(hood_vec_d.begin(), hood_vec_d.end(), nei_cache.begin() + (new_n * c_index));

}

//End of Gamma Function

//New M Matrix Calculation
//Delta * Hood stored in Delta
thrust::transform(D_vector_d.begin(), D_vector_d.begin() + (N * 1),
hood_vec_d.begin(), D_vector_d.begin(), thrust::multiplies<float>());
thrust::transform(D_vector_d.begin() + (N * 1), D_vector_d.begin()
+ (N * 2), hood_vec_d.begin(), D_vector_d.begin()+ (N * 1), thrust::multiplies<float>());
thrust::transform(D_vector_d.begin() + (N * 2), D_vector_d.begin()
+ (N * 3), hood_vec_d.begin(), D_vector_d.begin()+ (N * 2), thrust::multiplies<float>());
thrust::transform(D_vector_d.begin() + (N * 3), D_vector_d.end(),
hood_vec_d.begin(), D_vector_d.begin()+ (N * 3), thrust::multiplies<float>());

```



```

//Constant learning rate eta value = 0.7
thrust::fill(Eta_d.begin(), Eta_d.end(), 0.7);

//Delta * eta stores in Delta
thrust::transform(D_vector_d.begin(), D_vector_d.end(), Eta_d.begin(),
D_vector_d.begin(), thrust::multiplies<float>());

//M - Delta stores in M
thrust::transform(M_vector_d.begin(), M_vector_d.end(), D_vector_d.begin(),
M_vector_d.begin(), thrust::minus<float>());

//Write to File the neurons weight if we are in the last iteration
if (epocs == train-1)
{
    using namespace std;

    //Stop clock
    c_end = std::clock();

    //File Name buffer and formatting
    char filename [] = "iris-GPU-neurons";
    char underScore [] = "_";
    char txt [] = ".txt";
    char x_dim [33];
    char y_dim [33];
    char iter_v [33];

    snprintf(x_dim, sizeof(x_dim), "%d", x_val);
    snprintf(y_dim, sizeof(y_dim), "%d", y_val);
    snprintf(iter_v, sizeof(iter_v), "%d", iters_val);
    time_t t = time(0);    // get time now

    struct tm * now = localtime( & t );
    char time_buffer [120];
    strftime (time_buffer,120,"%X",now);

    strcat (filename, underScore);
    strcat (filename, x_dim);
    strcat (filename, underScore);
    strcat (filename, y_dim);
    strcat (filename, underScore);
    strcat (filename, iter_v);
    strcat (filename, underScore);
    strcat (filename, time_buffer);
    strcat (filename, underScore);
    std::cout << filename << std::endl;
    strcat (filename, txt);
    std::cout<<filename<<std::endl;

    neuronsFile.open(filename);

    //Iterator to writo to file
    Float4Iterator_d first_M1= thrust::make_zip_iterator(thrust::
make_tuple(M_vector_d.begin(),M_vector_d.begin() + (N * 1),M_vector_d.begin()

```



```

#include <string>
#include <fstream>
#include <math.h>
#include <vector>
#include <sstream>
#include <typeinfo>
#include <algorithm>
#include <random>
#include <chrono>

//Const Definitions
static const int DS_ROWS = 150; //Rows in the dataset
static const int DIMS_SIZE = 4; //Dimensionality of Dataset and Neurons
static const int DS_SIZE = DS_ROWS * DIMS_SIZE; //Based on dataset
static const int NEURONS_MAP_X = 8; //Neuron Map size X
static const int NEURONS_MAP_Y = 8; //Neuron Map size Y
static const int NEURONS_W_SIZE = (NEURONS_MAP_X * NEURONS_MAP_Y) * DIMS_SIZE;
//Based on Neuron Map
static const int COOR_SIZE = NEURONS_MAP_X * NEURONS_MAP_Y;
//Based on Neuron Map e.g 15 x 10

using std::default_random_engine;
using std::generate;
using std::uniform_real_distribution;
using std::uniform_int_distribution;
using std::vector;

float gen_random_float() {
    //static default_random_engine e; //turn this on for same randomw values
    static default_random_engine e( std::random_device{}() ); //turn this on for different randow values
    static uniform_real_distribution<float> dist(0.0, 1.0);
    return dist(e);
}

float gen_random_int() {
    //static default_random_engine e; //turn this on for same randomw values
    static default_random_engine e( std::random_device{}() ); //turn this on for different randow values
    static uniform_int_distribution<int> dist(0, DS_ROWS - 1); //random selection of training instace
    return dist(e);
}

static const std::string error_message =
    "Error: \_Result\_mismatch: \n"
    "i\_=%d\_CPU\_result \_=%d\_Device\_result \_=%d\n";

// This example illustrates the very simple OpenCL example that performs
// an addition on two vectors
int main(int argc, char** argv) {

```

```

if (argc != 2) {
    std::cout << "Usage:_" << argv[0] << "_<XCLBIN_File>" << std::endl;
    return EXIT_FAILURE;
}

std::string binaryFile = argv[1];
// compute the size of array in bytes

size_t size_in_bytes_ds      = DS_SIZE * sizeof(float);
size_t size_in_bytes_ds_row = 100000 * sizeof(int);
//Holds the random training vector indexes
size_t size_in_bytes_neur    = NEURONS_W_SIZE * sizeof(float);
size_t size_in_bytes_coor    = COOR_SIZE * sizeof(float);

cl_int err;
cl::CommandQueue q;
cl::Kernel krnl_vsom;
cl::Context context;

//create Matrix for dataset
float file_mat[DS_ROWS][DIMS_SIZE];
float file_mat_1d[NEURONS_W_SIZE];

//Create File for output
std::ofstream neurons_file ("iris_neurons.txt");
int counter = 0;

//Define and init vector X
vector<float, aligned_allocator<int> > source_x(NEURONS_W_SIZE, 0);

printf("\n_Print_out_File_Matrix:>>_:_\n\n");
    //storing map data in map_tiles
std::ifstream reader("iris.csv");
if (!reader)
    std::cerr << "Error_opening_file";
else
    {
        for (int i = 0; i < DS_ROWS; i++)
            {
                for (int j = 0; j < DIMS_SIZE; j++)
                    {
                        reader >> file_mat[i][j];
                        file_mat_1d[counter] = file_mat[i][j];
                        counter++;
                        reader.ignore();
                    }
            }
    }

//displaying map
//when n = amount of tiles on x axis, create a new line for the next set
counter = 0;
for (int i = 0; i < DS_ROWS; i++)

```

```

{
  for (int j = 0; j < DIMS.SIZE; j++)
  {
    std::cout << "[" << counter << "]" << file_mat[i][j] << "_";

    if (j == 3)
      std::cout << "\n";
  }
  counter++;
}
for (int i = 0; i < 100; i++)
{

  std::cout << "[" << i << "]" << file_mat_1d[i] << "_";
  std::cout << "\n";
}
std::cout << std::endl << "End_of_File_Matrix_Print" << std::endl;

srand(time(0));

//Define an Data vector D to hold data set
vector<float, aligned_allocator<int> > source_ds;

for (int i = 0; i < 150 * 4 ; i++)
{
  source_ds.push_back(file_mat_1d[i]);
}

std::cout << "The_content_of_the_ds_vector_is:" << std::endl;
for (int i=0; i < source_ds.size(); i++)
{
  if (i % 4 == 0){
    std::cout << "\n";
  }
  std::cout << source_ds.at(i) << '_';
}

//Define an x_k init vector (holds the random)
vector<int, aligned_allocator<int> > source_x_k(100000,0);

cl_int iters = 1;

vector<int, aligned_allocator<int> > source_iters(10,0);
//Ask for number of iterations
std::cout << "How_many_iters?" << std::endl;
std::cin >> iters;
source_iters[0] = iters;
source_x_k.resize(iters);

generate(begin(source_x_k), end(source_x_k), gen_random_int);

std::cout << "The_x_k_vector_contains:_ " <<std::endl;

```

```

for(int i=0; i < source_x_k.size(); i++)
{
    {
        std::cout << source_x_k.at(i) << '\n';
    }
}

std::cout<<std::endl;
//Define an init vector M
vector<float, aligned_allocator<int> > source_m(NEURONS_W_SIZE, 0);
generate(begin(source_m), end(source_m), gen_random_float);

//Input Randomm Neuron Weights
printf("Init _m_ before _kernel=_\n");
for (int i = 0; i < 600; i++) {
    printf("%f\n", source_m[i]);
    if (((i + 1) % 4) == 0) printf("\n");
}

//Define and init vector DELTA
vector<float, aligned_allocator<int> > source_delta(NEURONS_W_SIZE, 0);

//Define and init vector PI
vector<float, aligned_allocator<int> > source_p_ne(NEURONS_W_SIZE, 0);

//Define an init vector S
vector<float, aligned_allocator<int> > source_s(COOR_SIZE, 0);

//Define vector to hold minimum
vector<int, aligned_allocator<int> > source_s.min(COOR_SIZE, 0);

//Create Gamma_c Matrix to hold Neighborhood matrix
vector<float, aligned_allocator<int> > nei_vector(COOR_SIZE, 0);

//Create a P vector to hold coordinate P matrix
vector<float, aligned_allocator<int> > source_p(COOR_SIZE * 2, 0);

//Create a C vector to hold coordinate C matrix
vector<float, aligned_allocator<int> > source_c(COOR_SIZE * 2, 0);

//Create a D vector to hold coordinate Delta matrix
vector<float, aligned_allocator<int> > source_de(COOR_SIZE * 2, 0);

//Create a PI vector to hold coordinate PI matrix
vector<float, aligned_allocator<int> > source_pi(COOR_SIZE * 2, 0);

//Create a d vector to hold coordinate distance matrix
vector<float, aligned_allocator<int> > source_dis(COOR_SIZE, 0);

//Create a hood vector to hold coordinate hood matrix
vector<float, aligned_allocator<int> > source_hood(NEURONS_W_SIZE, 0);

```

```

//Create a neig vector to hold gamma function nei matrix
vector<float, aligned_allocator<int> > source_nei(COOR.SIZE, 0);

// The get_xil_devices will return vector of Xilinx Devices
auto devices = xcl::get_xil_devices();

// read_binary_file() is a utility API which will load the binaryFile
// and will return the pointer to file buffer.
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
bool valid_device = false;
for (unsigned int i = 0; i < devices.size(); i++) {
    auto device = devices[i];
    // Creating Context and Command Queue for selected Device
    OCLCHECK(err, context = cl::Context(device, nullptr, nullptr, &err));
    OCLCHECK(err, q = cl::CommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying_to_program_device[" << i << "]:_"
    << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    cl::Program program(context, {device}, bins, nullptr, &err);
    if (err != CL_SUCCESS) {
        std::cout << "Failed_to_program_device[" << i << "]:_with_xclbin_file!\n";
    } else {
        std::cout << "Device[" << i << "]:_program_successful!\n";
        // This call will extract a kernel out of the program we loaded in the
        // previous line. A kernel is an OpenCL function that is executed on the
        // FPGA. This function is defined in the src/vsom.k.cl file.
        OCLCHECK(err, krnl_vsom = cl::Kernel(program, "vsom.ko", &err));
        valid_device = true;
        break; // we break because we found a valid device
    }
}
if (!valid_device) {
    std::cout << "Failed_to_program_any_device_found,_exit!\n";
    exit(EXIT_FAILURE);
}

// These commands will allocate memory on the FPGA. The cl::Buffer objects can
// be used to reference the memory locations on the device. The cl::Buffer
// object cannot be referenced directly and must be passed to other OpenCL
// functions.

OCLCHECK(err, cl::Buffer buffer_ds(context, CL_MEM_USE_HOST_PTR |
CL_MEM_READ_WRITE, size_in_bytes_ds, source_ds.data(),
&err));
OCLCHECK(err, cl::Buffer buffer_x_k(context, CL_MEM_USE_HOST_PTR |
CL_MEM_READ_WRITE, size_in_bytes_ds_row, source_x_k.data(),
&err));
OCLCHECK(err, cl::Buffer buffer_m(context, CL_MEM_USE_HOST_PTR |
CL_MEM_READ_WRITE, size_in_bytes_neur, source_m.data(),
&err));

OCLCHECK(err, cl::Buffer buffer_iters(context, CL_MEM_USE_HOST_PTR |

```

```

CLMEM_READ_WRITE , sizeof(cl_int) * 10, source_iters.data(), &err));

// set the kernel Arguments
int narg = 0;

OCLCHECK(err, err = krnl_vsom.setArg(narg++, buffer_ds));
OCLCHECK(err, err = krnl_vsom.setArg(narg++, buffer_x_k));
OCLCHECK(err, err = krnl_vsom.setArg(narg++, buffer_m));
OCLCHECK(err, err = krnl_vsom.setArg(narg++, buffer_iters));
OCLCHECK(err, err = krnl_vsom.setArg(narg++, NEURONS_W_SIZE));

// These commands will load the source vectors from the host
// application and into the buffer_a and buffer_b cl::Buffer objects. The data
// will be transferred from system memory over PCIe to the FPGA on-board
// DDR memory.
OCLCHECK(err, err = q.enqueueMigrateMemObjects({buffer_ds, buffer_x_k, buffer_m,
buffer_iters}, 0 /* 0 means from host*/));

using namespace std::chrono;
//Timing kernel
auto start = high_resolution_clock::now();

// Launch the Kernel
OCLCHECK(err, err = q.enqueueTask(krnl_vsom));

// The result of the previous kernel execution will need to be retrieved in
// order to view the results. This call will write the data from the
// buffer_result cl_mem object to the source_results vector

OCLCHECK(err, err = q.enqueueMigrateMemObjects({buffer_m}, CLMIGRATE_MEM_OBJECT_HOST));
q.finish();

auto stop = high_resolution_clock::now();

//q.finish();

int match = 0;

printf("\n_New_M_Matrix_-_\n_\n");
for(int i=0; i<NEURONS_W_SIZE ; i++)
{
    if (neurons_file.is_open())
    {
        printf("%4.8f_", source_m[i]);
    }
}

```



```

        std::cout << std::setprecision(10);
        neurons_file << "_" << std::setprecision(10) << source.m[i] ;
        if (((i + 1) % DIMS_SIZE) == 0)
        { printf("\n");
          neurons_file << "\n";
        }
    }
}

auto duration = duration_cast<microseconds>(stop - start);

    // To get the value of duration use the count()
    // member function on the duration object
std::cout << "\nThe duration in microseconds was:" << duration.count() << std::endl;

std::cout << "TEST_" << (match ? "FAILED" : "PASSED") << std::endl;
return (match ? EXIT_FAILURE : EXIT_SUCCESS);
}

```

## The HLS-VSOM Openl Code Kernel (FPGA)

```

// iris-kernel.cl/
// version 1.0
// Author(s):Omar X. Rivera Morales
//
// This file constitutes a set of routines which are useful in constructing
// and evaluating self-organizing maps (SOMs)in a FPGA environment.

// Usage: The application generates the FPGA kernel

//Note: Requires Makefile to run and Vitis compiler

// This function represents an OpenCL kernel. The kernel will be call from
// host application using the xcl_run_kernels call. The pointers in kernel
// parameters with the global keyword represents cl_mem objects on the FPGA
// HBM memory.
//
#define DIM_SIZE 4
#define DS_ROWS 150
#define DS_SIZE (DS_ROWS * DIM_SIZE)
#define NEURONS_X_DIM 15
#define NEURONS_Y_DIM 10
#define NEURONS_W_SIZE (NEURONS_X_DIM * NEURONS_Y_DIM)
#define MAX_ITERS 100000

//For systolic
// Maximum Array Size
#define MAX_SIZE 150
#define MAX_SIZE_2 1

// TRIPCOUNT indentifier
//--constant uint c_size = BUFFER_SIZE;
kernel __attribute__((reqd_work_group_size(1,1,1)))
__attribute__((xcl_dataflow))
void vsom_ko(global float* ds, global int* x_k, global float* m,
global int* iters, const int n_elements) {

    //Local memory is implemented as BRAM memory blocks
    //Local BRAM Memory
    int temp_min_index = 0;
    float temp_min_val = 0.0;
    float eta = 0.7;
    // int count = 0;
    int n_iters = iters[0];

////////////////////////////////////
    int max_val = max(NEURONS_X_DIM,NEURONS_Y_DIM);
    //Fix the nei_size
    float nei_size = max_val + 1;

```

```

float temp_val = (float)n_iters/nei_size;
int counter = 0;
int nei_step = ceil((float)temp_val);
int nei_counter = 0;
if (nei_step == 0)
{
    nei_step = 1;
}

//init ds_local (Dataset)////////////////////////////////////
float ds_local[DS_ROWS][DIM_SIZE]; // __attribute__((xcl_array_partition(complete, 2)));

//Burst reads on input matrices from global Memory to local memory for competitive step
//Burst read for matrix ds contains the data set
read_ds:

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for (int itr = 0, i = 0, j = 0; itr < (DS_ROWS * DIM_SIZE); itr++, j++)
    {
        if(j == DIM_SIZE) {
            j = 0;
            i++;
        }
        ds_local[i][j] = ds[itr];
    }

//init M_local (Random neurons weights)////////////////////////////////////
float m_local[NEURONS.W_SIZE][DIM_SIZE] __attribute__((xcl_array_partition(complete, 2)));

//Burst reads on input matrices from global Memory
//Burst read for matrix m (Randomw Weights)

read_m:

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for (int itr = 0, i = 0, j = 0; itr < (NEURONS.W_SIZE * DIM_SIZE); itr++, j++)
    {
        if(j == DIM_SIZE) {
            j = 0;
            i++;
        }
        m_local[i][j] = m[itr];
    }

//init x_local
float x_local[NEURONS.W_SIZE][DIM_SIZE] __attribute__((xcl_array_partition(complete, 2)));

//init x_k_local
int x_k_local[MAX_ITERS]; //Holds all the randomw selection indexes

//Burst reads on input matrices from global Memory
//Burst read for matrix x

```

```

read_x_k:

//--attribute--((opencl_unroll_hint(2)))
--attribute--((xcl_pipeline_loop(1)))
    for (int itr = 0; itr < n_iters; itr++)
    {
        x_k_local[itr] = x_k[itr];
    }
//init d_local (Delta matrix)
float delta_local[NEURONS_W_SIZE][DIM_SIZE]
--attribute--((xcl_array_partition(complete, 2)));
float delta_local_copy[NEURONS_W_SIZE][DIM_SIZE]
--attribute--((xcl_array_partition(complete, 2)));
//init pi_local (pi matrix)

local float pi_local_copy[NEURONS_W_SIZE][DIM_SIZE]
--attribute--((xcl_array_partition(complete, 0)));
local float pi_local[NEURONS_W_SIZE][DIM_SIZE]
--attribute--((xcl_array_partition(complete, 0)));
//init s_local (s vector)
local float s_local[NEURONS_W_SIZE]
--attribute--((xcl_array_partition(complete, 1)));
local float s_local_temp[NEURONS_W_SIZE]
--attribute--((xcl_array_partition(complete, 1)));

//init pc_local (pi matrix)
float pc_local[NEURONS_W_SIZE][2]
--attribute--((xcl_array_partition(complete, 2)));

//Caching Arrays
//This array will have the flag indicating if the neig is cached or not
int mask_vec_d[NEURONS_W_SIZE] --attribute--((xcl_array_partition(complete, 1)));

// float [NEURONS_W_SIZE * NEURONS_W_SIZE];
// e.g 150 neruons will need 150 neurom measurmeent (all instances)
int nei_cache[NEURONS_W_SIZE][NEURONS_W_SIZE]
--attribute--((xcl_array_partition(complete, 2)));

read_Pc:

//--attribute--((opencl_unroll_hint(2)))
--attribute--((xcl_pipeline_loop(1)))
    for (int i = 0; i < NEURONS_W_SIZE; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            if(j == 0)
            {
                pc_local[i][j] = i / NEURONS_X_DIM ;
            }
        }
    }

```

```

        if(j == 1)
        {
            pc_local[i][j] = i % NEURONS_X_DIM ;
        }
    }
}

float c_coor_local[NEURONS_W_SIZE][2]
__attribute__((xcl_array_partition(complete, 2))); //Winner coordinates
float delta_coor_local[NEURONS_W_SIZE][2]
__attribute__((xcl_array_partition(complete, 2))); //Delta coor local
float pi_coor_local[NEURONS_W_SIZE][2]
__attribute__((xcl_array_partition(complete, 2))); //pi coor local
float dis_coor_local[NEURONS_W_SIZE]
__attribute__((xcl_array_partition(complete, 1))); //Distance to winner
int hood_local[NEURONS_W_SIZE]
__attribute__((xcl_array_partition(complete, 1))); //neighborhood local for Gamma function
int gamma_local[NEURONS_W_SIZE][DIM_SIZE]
__attribute__((xcl_array_partition(complete, 2))); // Holds neighborhood for M update
int temp_i;

//Systolic Magic ////////////////////////////////////////

int a_row = 150;
int a_col = 4;
int b_col = 1;

int b_row = a_col;
int c_row = a_row;
int c_col = b_col;

// Local memory to store input and output matrices

float localA[150][4] __attribute__((xcl_array_partition(complete, 1)));

float localB[150][1] __attribute__((xcl_array_partition(complete, 2)));

float localC[150][1] __attribute__((xcl_array_partition(complete, 0)));

float a[NEURONS_W_SIZE * DIM_SIZE];
/Cant do array partition, Max complete partition size is 1024

//Init localB with ones for systolic sum reduction
__attribute__((xcl_pipeline_loop(1)))
for (int itr = 0, i = 0, j = 0; itr < (DS_ROWS * DIM_SIZE); itr++, j++)
{
    if(j == DIM_SIZE) {
        j = 0;
        i++;
    }
    localB[i][j] = 1;
}

```

```

//Main VSOM loop
vsom_loop:
  for (int epoc = 0; epoc < n_iters; epoc++)

  {

    temp_i = x_k.local[epoc];

    //update_neighborhood size radius
    nei_counter = nei_counter + 1;
    if (nei_counter == nei_step)
    {
      nei_counter = 0;
      nei_size = nei_size - 1;

      //clear the masking cache array (flag for neighborhood available)

      __attribute__((opencl_unroll_hint(64)))
      __attribute__((xcl_pipeline_loop(1)))
      for (int i = 0; i < NEURONS_W_SIZE; i++)
      {
        mask_vec_d[i] = 0;
      }

    }

    /*** Updating X Matrix ***/

    iris_x:

    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for (int i = 0; i < NEURONS_W_SIZE; i++)
    {
      for (int j = 0; j < DIM_SIZE; j++)
      {
        x_local[i][j] = ds_local[temp_i][j]; //here is the training data point
      }
    }

    /*** Finding the winning neuron ***/

    /*** Finding the winning neuron ***/

    iris_delta_l :

    //for local
    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for (int i = 0; i < NEURONS_W_SIZE; i++)
    {
      for (int j = 0; j < DIM_SIZE; j++)
      {

```

```

        delta_local[i][j] = m_local[i][j] - x_local[i][j];

    }

}

iris_delta_2:
    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            delta_local_copy[i][j] = delta_local[i][j];
        }
    }

iris_delta_3:
    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            float delta_val_1 = delta_local_copy[i][j];
            float delta_val_2 = delta_local[i][j];
            pi_local_copy[i][j] = delta_val_1 * delta_val_2;
        }
    }

iris_delta_4:

    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            pi_local[i][j] = pi_local_copy[i][j];
        }
    }

clear_s_vector:

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
for ( int i = 0; i < NEURONS.W_SIZE; i++)
{

```

```

        s_local[i] = 0;
    }

__attribute__((xcl_pipeline_loop(1))) systolic1 : for (int k = 0; k < a_col; k++) {
    systolic2 : for (int i = 0; i < MAX_SIZE; i++) {
        systolic3 : for (int j = 0; j < MAX_SIZE_2; j++) {
            // Get previous sum
            float last = (k == 0) ? 0 : localC[i][j];

            // Update current sum
            // Handle boundary conditions
            float a_val = (i < a_row && k < a_col) ? pi_local[i][k] : 0;
            float b_val = (k < b_row && j < b_col) ? localB[k][j] : 0;
            float result = last + a_val * b_val;

            // Write back results
            localC[i][j] = result;
        }
    }
}

update_s_vector:
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
for (int i = 0; i < NEURONS_W_SIZE; i++)
{
    s_local[i] = localC[i][0];
}

iris_s_min:
temp_min_val = s_local[0];

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
for (int j = 0; j < NEURONS_W_SIZE; j++)
{
    if (s_local[j] < temp_min_val)
    {
        temp_min_index = j;
        temp_min_val = s_local[j];
    }
}

////////////////////////////////////

if (mask_vec_d[temp_min_index] == 1)
{check_cache_nei:
    __attribute__((opencl_unroll_hint(64)))

```



```

__attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W.SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            gamma_local[i][j] = nei_cache[temp_min_index][i];
        }
    }

    counter = 0;
}

//else //////////////////////////////////////enter creater nei
else {

    mask_vec_d[temp_min_index] = 1;

    iris_p_coord:

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for(int i=0; i < NEURONS.W.SIZE; i++) {

//Local Mode
        c_coord_local[i][0] = pc_local[temp_min_index][0];
        c_coord_local[i][1] = pc_local[temp_min_index][1];

    }

    iris_coord_delta_pi_distance_1 : //Delta , PI and d coordinate calculations
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
        for (int i = 0; i < NEURONS.W.SIZE; i++) {

            delta_coord_local[i][0] = pc_local[i][0] - c_coord_local[i][0];
        }

    iris_coord_delta_pi_distance_2 : //Delta , PI and d coordinate calculations
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
        for (int i = 0; i < NEURONS.W.SIZE; i++) {

            delta_coord_local[i][1] = pc_local[i][1] - c_coord_local[i][1];

        }

    iris_coord_delta_pi_distance_3 : //Delta , PI and d coordinate calculations
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
        for (int i = 0; i < NEURONS.W.SIZE; i++) {

```

```

        pi-coor-local[i][0] = delta-coor-local[i][0] * delta-coor-local[i][0];

    }

iris-coor-delta-pi-distance-4 : //Delta , PI and d coordinate calculations
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for (int i = 0; i < NEURONS.W_SIZE; i++) {

        pi-coor-local[i][1] = delta-coor-local[i][1] * delta-coor-local[i][1];

    }

iris-coor-delta-pi-distance-5 : //Delta , PI and d coordinate calculations
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for (int i = 0; i < NEURONS.W_SIZE; i++) {

        dis-coor-local[i] = pi-coor-local[i][0] + pi-coor-local[i][1];

    }

iris_hood:

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        if (sqrt(dis-coor-local[i]) < nei_size * 1.5)
        {
            hood_local[i] = 1;
        }
        else
        {
            hood_local[i] = 0;
        }
    }

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            gamma_local[i][j] = hood_local[i];
        }
    }

copy_cache_nei:
__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        nei_cache[temp_min_index][i] = hood_local[i];
    }

```

```

    } //end of nei catching if-else

    gamma-vec:

// //////////////////////////////////////

__attribute__((opencl_unroll_hint(64)))
__attribute__((xcl_pipeline_loop(1)))
    for ( int i = 0; i < NEURONS.W_SIZE; i++)
    {
        for ( int j = 0; j < DIM.SIZE; j++)
        {
            m_local[i][j] = m_local[i][j] - eta * delta_local[i][j] * gamma_local[i][j];
        }
    }

} //end of training loop

//check writeback

write_back_global-4: //Write back Matrix Global M

    __attribute__((opencl_unroll_hint(64)))
    __attribute__((xcl_pipeline_loop(1)))
    for (int itr = 0, i = 0, j = 0; itr < (NEURONS.W_SIZE * DIM.SIZE); itr++, j++) {
        if (j == 4) {
            j = 0;
            i++;
        }
        m[itr] = m_local[i][j];
    }
}

////////////////////////////////////End of Kernel

```

## List of References

- [1] U. Seiffert and B. Michaelis, “Multi-dimensional self-organizing maps on massively parallel hardware,” in *Advances in Self-Organising Maps*. Springer, 2001, pp. 160–166.
- [2] L. Hamel, “Som quality measures: An efficient statistical approach,” in *Advances in Self-Organizing Maps and Learning Vector Quantization*. Springer, 2016, pp. 49–59.
- [3] R. A. Fisher, “The use of multiple measurements in taxonomic problems,” *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

- [4] P. F. Thall and S. C. Vail, “Some covariance models for longitudinal count data with overdispersion,” *Biometrics*, pp. 657–671, 1990.
- [5] W. N. Street, W. H. Wolberg, and O. L. Mangasarian, “Nuclear feature extraction for breast tumor diagnosis,” in *IS&T/SPIE’s Symposium on Electronic Imaging: Science and Technology*. International Society for Optics and Photonics, 1993, pp. 861–870.

## APPENDIX C

### Conclusion

This research aimed to identify and generate superior speed-up and performance strategies for parallel self-organizing maps. Based on a quantitative and qualitative analysis of parallel SOMs experimentation in GPUs and FPGAs hardware accelerators, it can be concluded that our vectorized GPU and FPGA implementations of the SOMs do provide a superior alternative for parallel SOMs. The results indicate the current hardware accelerators are a good alternative for implementing the parallelization of the vectorized SOMs. Furthermore, our findings demonstrate that the vectorized SOM in GPUs and FPGAs offers superior performance and speed-up gains than the other available parallel implementation and can generate the same quality of maps as SOMs in the CPUs environment.

The GPU implementation (Par-VSOM) obtained substantial performance increases over Kohonen’s iterative SOM algorithm (up to 67 times faster), the CPU based vectorized VSOM (up to 4 times faster), the GPU *Xpysom* (up to 6.1 times) and *Quicksom’s* GPU (up to 20 times) in large maps environments. The results obtained by increasing the dimensionality and map sizes demonstrated that the Par-VSOM provides scalable speed-up performance when the neuronal map size increases.

The HLS-VSOM was developed for an FPGA architecture for our second hardware accelerator environment. The HLS-VSOM is a high-level synthesis parallel version of the vectorized and matrix-based implementation of stochastic training for self-organizing maps. The HLS variant also offers significant performance gains over Kohonen’s iterative SOM algorithm (up to 30.4X times faster) and the CPU-based vectorized VSOM (up to 6.3x times faster). Our comparisons with the GPU

variants also demonstrate that the optimized FPGA VSOM surpasses the GPU Par-VSOM and XPySom GPUs version by two or three orders of performance in various datasets using regulars size maps environment. The results obtained with the HLS-VSOM demonstrated it is the best performance parallel SOM currently available.

This research clearly illustrates the superior speed-up gains achievable with parallel vectorized SOMs. However, it also raises the question of how can we make the GPU variant work efficiently with smaller maps. In contrast, the HLS-VSOM offers a great superior performance gains alternative with regular size maps, but it does have limitations with larger maps due to increasing memory access, logic resource limitation, and highly complex routing schemes.

Based on our results, the Par-VSOM and the HLS-VSOM can be viewed as an alternative to parallel SOMs and a new alternative for other parallel algorithms for clustering. To better understand the implications of these results, future studies could research the implementation of the VSOM using the Tensor-cores available in newer NVIDIA GPU chip architectures. Another alternative will be using Google's Tensor Processing Unit (TPU) AI accelerator application-specific integrated circuit (ASIC). Both of these architectures can provide additional performance gains and novel research discoveries using Tensor cores technologies for the SOMs.