High-Level Synthesis Parallelization and Optimization of Vectorized Self-Organizing Maps

Omar X. Rivera-Morales
Dept. of Computer Science
University of Rhode Island
Kingston, RI USA
omar.riveramorales@my.uri.edu

Lutz Hamel
Dept. of Computer Science
University of Rhode Island
Kingston, RI USA
lutzhamel@uri.edu

***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.**

***Index Terms*—SOM, FPGA, parallel computing, High-Level Synthesis**

I. 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 II starts our discussion with an overview of the HLS and a brief description of the major stages. Under section III, 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 V. As part of Section IV, we develop the HLS-VSOM training and examine the instruction pipelining, scaling data level parallelisms, array partitioning, and memory optimization transformations. Under section VI, 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 VII , we conclude our discussion with a summary of the observations and some future research ideas under consideration.

II. 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 "bottleneck" 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.

III. 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 $m_i \in R^n, i = 1, 2, ..., k$ 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 $x = [\xi_1, \xi_2, ..., \xi_n]^T \in R^n$. The SOM neuronal map can be defined as a model containing the parametric real vector $m_i = [u_{i1}, u_{i2}, ..., u_{in}]^T \in R^n$ 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 (1).

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

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,

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

with $i = 1, \ldots, n$. The training data x consists of a set D= $\{\mathbf{x}_1, \ldots, \mathbf{x}_l\}$. 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,

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

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

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.

A. The VSOM Competitive Step

In the SOM algorithm the competitive step is accomplished in a sequential manner searching for the BMU using Equation(1). 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 X to hold a randomly selected training vector. The matrix X in equation (4) is defined with a component sizes of $n \times d$, where each row is holding the current epoch *training*

vector $\mathbf{x}_k = (x_1, x_2, \ldots, x_d)_k$, which is randomly selected from matrix $\mathbf{D}$,

$$\mathbf{X} = \mathbf{1}^n \otimes \mathbf{x}_k. \tag{4}$$

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

$$\mathbf{1}^n = \underbrace{(1, 1, \ldots, 1)}_{n}^{\mathbf{T}}. \tag{5}$$

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 our epoch training instance matrix X with the duplicated $\mathbf{x}_k$ values, equations (6), (7) and (8) are used to compute the square of the Euclidean distances between the map neurons and the input vector,

$$\mathbf{\Delta} \leftarrow \mathbf{M} - \mathbf{X} \tag{6}$$

$$\mathbf{\Pi} \leftarrow \mathbf{\Delta} \circ \mathbf{\Delta} \tag{7}$$

$$\mathbf{s} \leftarrow \mathbf{\Pi} \times \mathbf{1}^d \tag{8}$$

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 X, 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. In order to find the BMU, we search for the location of the minimum value in vector s.

B. The 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.

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

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

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

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,

$$\mathbf{M} \leftarrow \mathbf{M} - \eta\mathbf{\Delta} \circ \mathbf{\Gamma}_c. \tag{11}$$

Algorithm 1 The HLS-VSOM training algorithm.

Given:
D $\leftarrow$ {training instances, a *l* x *d* matrix}
M $\leftarrow$ {neurons, a *n* x *d* vector of tuples}
$\eta \leftarrow$ {learning rate $0 < \eta < 1$}
$\Gamma(c) \leftarrow$ {neighborhood function for some neuron c}
minIndex(s) $\leftarrow$ {func, returns location of min. val in s}
$\Phi \leftarrow$ {Rowsum reduction using Systolic Array dot product}
$\Omega \leftarrow$ {Pipeline, unrolled loops kernel operations}
$R \leftarrow$ {Random index values list}
$O \leftarrow$ {constant column vector with value of 1's}

Repeat:
/***Select a matrix training instance as vector
for some *k* = $1, ..., l$: ***/
$x_k \leftarrow \mathrm{D}[R_k]$
/***Find the winning neuron using accelerated kernels ***/
$\mathrm{X} \leftarrow \Omega_x(1^n \otimes x_k)$
$\Delta \leftarrow \Omega_\Delta(\mathrm{M} - \mathrm{X})$
$\Pi \leftarrow \Omega_\Pi(\Delta \circ \Delta)$
/***Reduction (Rowsum) Using Systolic Arrays and DSP***/
$s \leftarrow \Phi_s(\Pi \cdot O)$
c = minIndex(s)
/***Update neighborhood with vector operations ***/
$\Gamma_c \leftarrow \Omega_\Gamma(\Gamma(c))$
$M_{new} \leftarrow \Omega_{Mnew}(M_{current} - \eta\Delta \circ \Gamma_c)$
done
return $\mathbf{M_{new}}$

Here, η 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 η 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.

IV. High-Level Synthesis VSOM

A. 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 1 and 2 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

Algorithm 2 The HLS-VSOM Neighborhood Function Γ.

given:
$c \leftarrow$ {index of winning neuron}
$n \leftarrow$ {the number of neurons on the map}
$nsize \leftarrow$ {neighborhood radius}
$\mathbf{P} \leftarrow$ {an $n \times 2$ matrix with $\mathbf{p}_i = \mathbf{P}[i,] = (x_i, y_i)$}
$\mathbf{1}^n \leftarrow$ {constant column vector with value 1}
$\mathbf{0}^n \leftarrow$ {constant column vector with value 0}
$\Omega \leftarrow$ {Pipeline and unrolled loops kernel operations}

$P_c \leftarrow \Omega_{pc}(P[c,])$
$\mathbf{C} \leftarrow \Omega_C(\mathbf{1}^n \otimes \mathbf{p}_c)$
$\mathbf{\Delta} \leftarrow \Omega_\Delta(\mathbf{P} - \mathbf{C})$
$\mathbf{\Pi} \leftarrow \Omega_\Pi(\mathbf{\Delta} \circ \mathbf{\Delta})$

/***Perform rowsum matrix reduction
$\mathbf{d} \leftarrow \Omega_d(\mathbf{\Pi_x}+\mathbf{\Pi_y})$
$\mathbf{hood} \leftarrow \Omega_{hood}(\text{ifelse}(\mathbf{d} < (nsize \times 1.5)^2, \mathbf{1}^n, \mathbf{0}^n))$
return hood

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 inserted 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).*

B. 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 computational 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 1.

__attribute__((xcl_pipeline_loop(1)))

Instr. No. \ Clock cycle	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.

Fig. 1: 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 1 and 2. In our HLS kernel, we are pipelining all the matrix and vector operations to maximize the execution per clock ratio.

C. 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 "embarrasingly 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 2. 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.

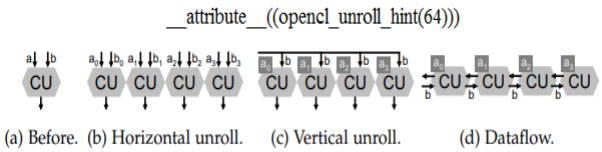


Fig. 2: Scalability Transformations HLS - here the rectangles represent buffer space, such as FPGA registers or on chip Ram [4] and the CU refers to computational units.

D. 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 3

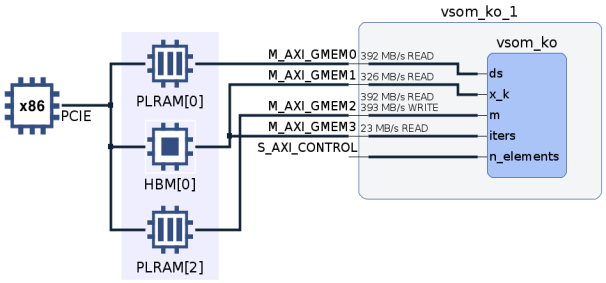


Fig. 3: Par-VSOM HLS Memory (Striping) Access Transformations.

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 3. The stripping 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.

E. 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 4, 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 1 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 Π 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.

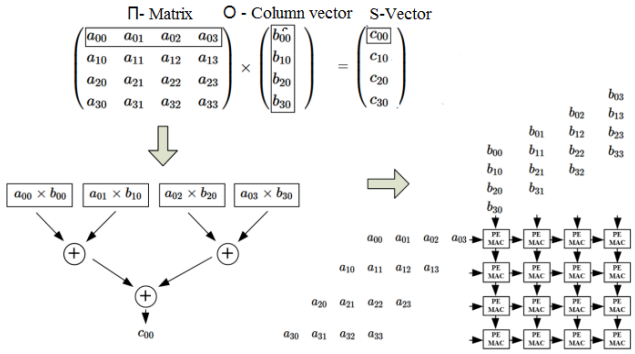


Fig. 4: Systolic Array Matrix Multiplication. [12]

V. 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 methods. 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]–[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.

A. 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.

B. 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].

C. 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.

D. 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.

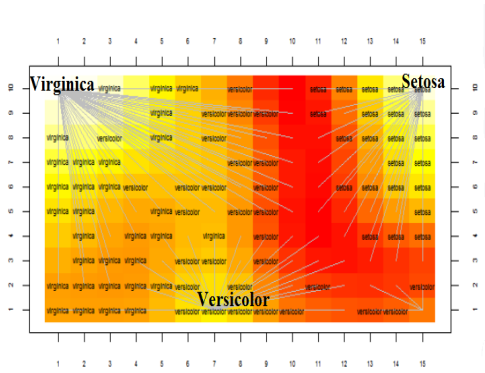


Fig. 5: 15 x 10 Self Organizing Map for the Iris Dataset.

E. SOMs in GPUs

The GPUs also provide an excellent hardware solution for the parallelization of the SOM. The SOM GPUs 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.

VI. Experiments

A. 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 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.

B. 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 I. 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 x 8 as an approximation but eventually we decided to increase our map size to 15 x 10 larger for more complexity.

TABLE III: Times and Speed-up gains of the HLS-VSOM compare against a non-accelerated FPGA HLS-VSOM using a 15×10 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
50	0.337	0.276	1.2
100	0.591	0.481	1.2
500	2.624	2.126	1.2
1000	5.074	4.052	1.3
5000	24.962	18.323	1.4
	*** Epil D=8***		
1	0.073	0.066	1.1
50	2.175	0.282	7.6
100	4.252	0.511	8.3
500	20.809	2.045	10.2
1000	41.449	3.811	10.8
5000	206.850	17.171	12.4
	*** WDBC D=30***		
1	0.233	0.143	1.6
50	5.03	0.260	10.0
100	9.860	0.505	12.1
500	48.257	0.815	15.4
1000	96.099	3.134	16.2
5000	477.866	26.052	18.3

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 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.

C. 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 II, 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).

TABLE I: Par-VSOM Hyper-Parameters.

Hyper-Parameter	**Values**
Training Iterations Range	$1 - 5000$
Learning Rate η	0.7
Neighborhood Radius	Bubble
Training Data Sets	Iris, Epil, WDBC

TABLE II: Times and Speed-up gains of the HLS Par-VSOM for different training algorithms using a 15×10 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
				*** Iris D=4***					
1	0.033	0.017	0.005	0.001	0.000034	961.1	495.1	145.6	29.1
50	0.034	0.018	0.015	0.050	0.000276	123.0	65.1	54.3	180.9
100	0.036	0.018	0.024	0.099	0.000481	74.8	37.4	49.9	205.8
500	0.038	0.019	0.096	0.494	0.002126	17.8	8.9	45.1	232.3
1000	0.075	0.021	0.170	0.986	0.004052	18.5	5.2	41.9	243.3
5000	0.251	0.028	0.794	4.937	0.018323	13.7	1.5	43.3	269.4
				*** Epil D=8***					
1	0.040	0.022	0.011	0.001	0.000066	600.0	330.0	162.0	15.0
50	0.041	0.020	0.023	0.050	0.000282	141.7	67.3	81.5	177.1
100	0.043	0.021	0.035	0.102	0.000511	84.0	39.1	68.4	205.2
500	0.062	0.022	0.115	0.513	0.002040	30.3	10.8	56.2	255.2
1000	0.092	0.026	0.221	1.024	0.003811	24.1	6.3	58.0	274.4
5000	0.287	0.040	1.001	5.162	0.017176	16.7	2.3	58.3	306.3
				*** WDBC D=30***					
1	0.041	0.020	0.018	0.001	0.000143	286.0	139.5	125.6	7.0
50	0.042	0.020	0.040	0.051	0.000505	83.1	39.6	79.1	100.9
100	0.046	0.022	0.063	0.103	0.000815	56.4	28.2	77.7	126.3
500	0.080	0.028	0.236	0.507	0.003136	25.5	8.9	75.3	161.8
1000	0.181	0.034	0.452	1.104	0.005946	30.4	5.7	76.0	185.7
5000	0.489	0.093	2.131	5.068	0.026052	18.8	3.6	81.8	194.5

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 II 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 V. The comparison with the GPU version demonstrate the GPU versions are not well suited for regular size maps with normal 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 6a - 6c, 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 dimentionality of neuronal maps and better achievable times using the HLS-VSOM. That is, with more dimentionality complexity in the dataset a better speed up can be achieved making it scalable.

Table III results include the times and Speed-up gains of the HLS-VSOM compared against a non-accelerated FPGA HLS-VSOM using an 15 × 10 map. The proposed accelerated HLS-VSOM uses pipelined loops, dataflow, horizontal unrolling, array partitioning, and systolic arrays for row sum reductions allow us to achieved 18.3X 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 Pragmas directives for optimization.

In terms of the quality of the maps, Table IV 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.

TABLE IV: 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)

Total iters	SOM	VSM	CI P-V	X-P	H-P
		*** Iris, D=4***			
50	0.41	0.42	0.34	0.50	0.33
100	0.43	0.45	0.70	0.50	0.50
500	0.42	0.79	0.71	0.83	0.85
1000	0.92	0.95	0.91	0.88	0.97
5000	0.96	0.96	0.94	0.93	0.97
		*** Epil, D=8***			
50	0.35	0.33	0.49	0.46	0.47
100	0.56	0.45	0.52	0.49	0.34
500	0.43	0.61	0.70	0.86	0.80
1000	0.92	0.92	0.94	0.91	0.92
5000	0.90	0.91	0.93	0.90	0.94
		*** WDBC, D=30***			
50	0.28	0.19	0.50	0.62	0.40
100	0.23	0.40	0.55	0.53	0.47
500	0.32	0.74	0.72	0.62	0.84
1000	0.90	0.92	0.88	0.68	0.91
5000	0.92	0.93	0.94	0.71	0.91

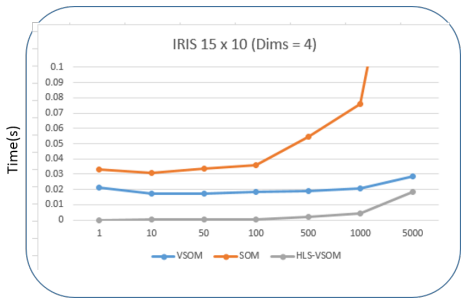
TABLE V: HLS-VSOM FPGA Speed-ups Summary

Dataset Speed-up vs SOM-CPU:	**MAX**	**@ Convergence*** 1000 iters
IRIS	961.1	18.5
EPIL	600	24.1
WDBC	286	30.4
Dataset Speed-up vs VSOM-CPU:	**MAX**	** @ Convergence*** 1000 iters
IRIS	495.1	5.2
EPIL	330.0	6.3
WDBC	139.5	5.7
Dataset Speed-up vs Par-V-GPU:	**MAX**	** @ Convergence*** 1000 iters
IRIS	145.6	43.3
EPIL	162.0	58.0
WDBC	125.6	76.0
Dataset Speed-up vs Xpysom-GPU:	**MAX**	** @ Convergence*** 1000 iters
IRIS	269.4	243.3
EPIL	306.3	274.4
WDBC	194.5	185.7

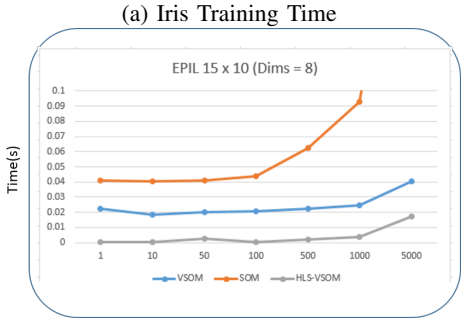
VII. Conclusion

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 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[6a - 6c]. 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.

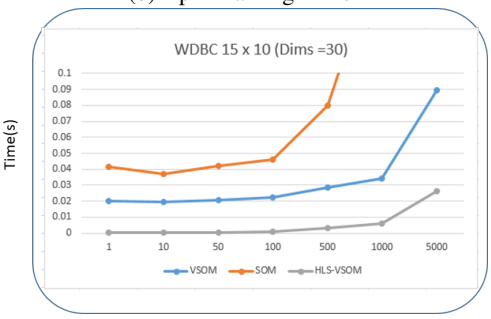
In the proposed design, the HLS-VSOM is a highly optimized algorithm running in a FPGA Accelerator Card and therefore is an adequate replacement for iterative stochastic training of SOM and parallel SOM variants. Future research on this topic will include investigating how the HLS-VSOM can be implemented 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.



(a) Iris Training Time



(b) Epil Training Time



(c) WDBC Training Time

Fig. 6: Total Training Time for all datasets with multiple map sizes.

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. Hoefler, "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, 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.