electronics

Article

Hardware Architecture for Asynchronous Cellular Self-Organizing Maps

Quentin Berthet *, Joachim Schmidt and Andres Upegui *

inIT, Hepia, University of Applied Sciences and Arts of Western Switzerland, 1202 Geneva, Switzerland; ; joachim.schmidt@hesge.ch
* Correspondence: quentin.berthet@hesge.ch (Q.B.); andres.upegui@hesge.ch (A.U.)

Abstract: Nowadays, one of the main challenges in computer architectures is scalability; indeed, novel processor architectures can include thousands of processing elements on a single chip and using them efficiently remains a big issue. An interesting source of inspiration for handling scalability is the mammalian brain and different works on neuromorphic computation have attempted to address this question. The Self-configurable 3D Cellular Adaptive Platform (SCALP) has been designed with the goal of prototyping such types of systems and has led to the proposal of the Cellular Self-Organizing Maps (CSOM) algorithm. In this paper, we present a hardware architecture for CSOM in the form of interconnected neural units with the specific property of supporting an asynchronous deployment on a multi-FPGA 3D array. The Asynchronous CSOM (ACSOM) algorithm exploits the underlying Network-on-Chip structure to be provided by SCALP in order to overcome the multi-path propagation issue presented by a straightforward CSOM implementation. We explore its behaviour under different map topologies and scalar representations. The results suggest that a larger network size with low precision coding obtains an optimal ratio between algorithm accuracy and FPGA resources.

Keywords: self-organising maps; cellular computing; network-on-chip; bio-inspired hardware; neuromorphic architectures; hardware neural networks

check for updates

Citation: Berthet, Q.; Schmidt, J.; Upegui, A. Hardware Architecture for Asynchronous Cellular Self-Organizing Maps. *Electronics* **2022**, *11*, 215. https://doi.org/10.3390/electronics11020215

Academic Editor: Gemma Piella

Received: 8 October 2021
Accepted: 5 January 2022
Published: 11 January 2022

1. Introduction

The work presented in this paper is part of the Self-Organizing Machine Architecture (SOMA) project in which we propose new architectures based on brain-inspired self-organisation principles applied to digital reconfigurable hardware [1].

Self-Organizing Maps (SOM) are a well-known type of neural network enabling vector quantization using an unsupervised competitive learning technique [2] . During the learning phase, they perform dimensionality reduction and clustering on unlabeled data and, once trained, they can apply the same operations to new data. They find applications in several domains such as voice recognition [3], image compression [4,5] or general data clustering. While their self-organizing and unsupervised properties are very interesting, they are computationally expensive and a lot of efforts have been invested by the community to propose efficient dedicated hardware SOM architectures.

Most of the existing works focus on optimizing the time and power performances by modifying the SOM algorithm to new "hardware-friendly" variants, less expensive in term of hardware resources such as logic and memory elements. The use of Euclidean distance used in the formal definition of SOM and in most of its software implementations are often replaced by a Manhattan distance [6–10] or Chessboard [6] distance functions to avoid multipliers and square root operators. Another common optimization is to replace the exponentiation used to compute the Gaussian neighboring function by approximations such as negative power of two implemented with shift registers [5,6,10,11] or Lookup-Tables [7,9]. In all the previously cited works, vectors are represented with integer or fixed-point arithmetic, with usually 16-bits per vector component. This approach simplifies the arithmetic's unit hardware implementations and reduces the memory requirement compared to the 32 or 64 bits of the single or double precision format commonly used in software implementations. More original vector codings have been proposed by [12] which proposes to encode the vector components as the duty cycle of square waveforms and to use Digital Phase-Locked Loops (DPLL) as parallel computing elements to measure and update the distances between vectors. In [13], it is proposed to code the vectors with tri-state representation and to use Hamming distance as the metric between vectors with probabilistic updates of the weights vectors.

While these optimizations make perfect sense in term of computing time and logic resources reduction, they can introduce deviations from the original SOM algorithm results. As an example, the use of the Manhattan distance instead of Euclidean distance does not yield the same result and this issue is even accentuated with high dimensional vectors. The fixed point representation is also an usual optimization but it requires to carefully select the bit-width and comma position of the format to ensure that intermediate values of the algorithm are correctly represented. This may be a problem in real world applications where the magnitude and precision of the data-set provided to the SOM is not known in advance.

The structure of self-Organizing Maps makes them good candidates for distributed implementations as it is straightforward to parallelize different computation tasks like distance computation and weights update operations over multiple neurons. In the neurons themselves, some implementations process the vectors components simultaneously to speed up computation while some other works serialize the component processing to save resources. The election of a best matching unit (BMU) constitutes a bottleneck for scalable distributivity because a single neuron must be chosen. Fully distributed winner neuron election often relies on a large combinational comparator network which largely limits the maximum operating frequency and the scalability of the system.

In a previous work we proposed a cellular variant of the SOM algorithm in order to cope such scalability limitations [14]. Cellular SOM (CSOM) can achieve better performance than the traditional SOM and is better suited to distributed implementations as their cellular communication properties reduce the need for a central authority and shared resources by favoring neighbor to neighbor communications. CSOM is modelled as a connected network of computing nodes permitting arbitrary and dynamic topologies that can adapt during run-time to the problem thanks to pruning and sprouting mechanisms [15,16].

In another previous work, we presented a Self-configurable 3D Cellular Adaptive Platform (SCALP) [17], which is intended to prototype 3D cellular neuromorphic architectures. It is composed of multiple FPGA-SoC nodes: circuit boards hosting a Xilinx Zynq FPGA SoC, DDR memory and high-speed communication links. The boards are designed so they can be assembled as a 3-dimensional array, as illustrated in Figure 1a. Communication links between the nodes use both LVDS links (4×928 Mb/s bidirectional along X, Y and Z) and high-speed serial interfaces (6.5 Gb/s bidirectional, only on the X-Y plane) as illustrated in Figure 1b. On top of that, an Ethernet connectivity is deployed between all the nodes, and allows access to each individual node from an external PC for the purpose of controlling and monitoring the distributed task's execution. The main purpose of this platform is the study of self-organizing and distributed algorithms as well as underlying Network-On-Chip (NoC) and inter-chip communication protocols. We have also developed SCALPsim [18], a multi-threaded simulator able to model asynchronous computing nodes and heterogeneous inter-node communication links, with the goal of simulating the systems to be deployed on the platform.

The multi-FPGA architecture offered by the SCALP platform provides enough resources to reduce the negative impact of fixed point quantization on accuracy loss. Optimized floating point representation increases system accuracy compared to fixed point representations by still keeping a low resource utilization. An architectural design-space exploration will allow us to obtain the best trade-off between hardware resources versus algorithm accuracy.

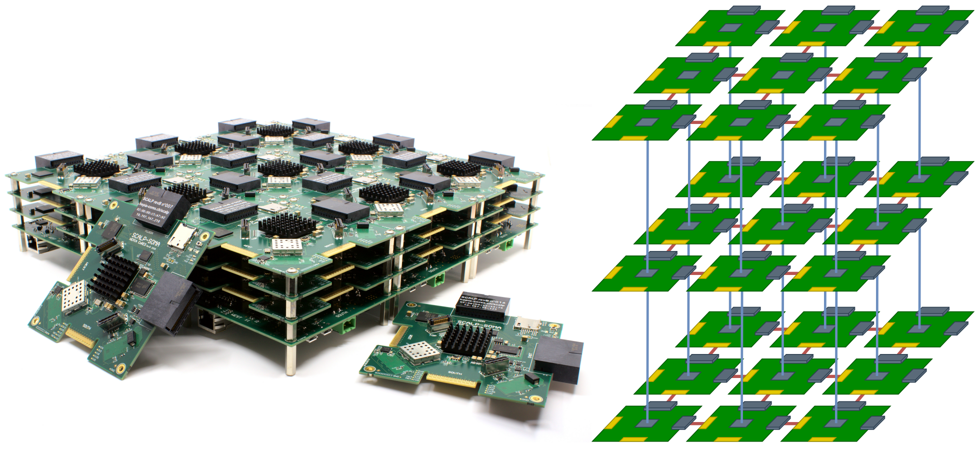

(**a**) (**b**)

Figure 1. Illustration of SCALP board and its network topology. (**a**) Array of SCALP nodes. (**b**) SCALP array communications links. Red lines represent bi-directional high-speed connection. Blue lines show bi-directional differential links.

The purpose of this work is two-fold: first, we propose improvements of the CSOM algorithm in order to better suit a hardware implementation by introducing ACSOM, an asynchronous version of CSOM. This implies taking into account the inter-board communication constraints imposed by the SCALP platform. Second, we explore the quantization error introduced by using various floating point number representations with different ACSOM topologies while avoiding the usual distance and neighboring function simplifications. All this is materialized in the form of a flexible hardware neuron architecture able to perform on-line learning on a synthetic data-set with vectors of floating point values and which has been experimentally tested in FPGA fabric. The underlying NoC layer is not in the scope of the present paper and will be presented in a future work.

This paper is organized as follows: Section 2 quickly recalls the SOM and CSOM algorithms and presents the ACSOM algorithm. Section 3 describes the neuron architecture in detail. Section 4 presents a comparison between CSOM and ACSOM in terms of behaviour and accuracy and explains the experimental setup used to evaluate the ACSOM hardware architecture. The obtained results are then compared with state-of-the-art implementations. Section 5 discusses the results and proposes several directions for future works.

2. Self-Organizing Maps

2.1. SOM: Self-Organizing Maps

The classical Self-Organising Maps (SOM) initially proposed by Kohonen [2] consists of an ensemble of neurons, each associated with a weight vector, organized in a low-dimensional lattice (Usually from one to three dimensions) and with defined update rules enabling unsupervised quantization of arbitrary dimension input vectors. During the learning phase, the vectors of the input set are successively presented to every neuron of the network. At each new input, the neuron with the closest weight vector to the input vector (according to a distance function) is elected as the winning neuron and called the Best Matching Unit (BMU). Weights of all the surrounding neurons are then updated toward the new input, with decreasing influence, starting from the BMU. This process repeated over the training dataset enables the neurons to learn by mapping the input vector space in a self-organizing manner. At the end of the learning phase, neuron weights are considered the codebook of the input data set and can be used in a recall phase to perform clustering on a new input vector set. An interesting property of this process is the topology preserving behaviour which tends to code similar input vector to neuron close to each other on the map.

Each neuron is represented by a weight vector $\mathbf{m} \in \mathbb{R}^d$ where d is the dimension of the input vectors $\mathbf{x} \in \mathbb{R}^d$. After grid weights initialization (usually random), the learning process can be formalized as follow: at each iteration a new input vector $\mathbf{x}$ from the set is presented to the network. The corresponding BMU is elected by finding the neuron c whose weight vector is closest to the input $\mathbf{x}$ (Using a distance function, usually Euclidean distance). Then all other neurons i update their weight vector according to:

$$\mathbf{m}_i(t+1) = \mathbf{m}_i(t) + \epsilon(t)(\mathbf{x}(t) - \mathbf{m}_i(t))e^{-\frac{\|\mathbf{r}_c - \mathbf{r}_i\|^2}{2\sigma^2(t)}}, \tag{1}$$

where t denotes the iteration, $\mathbf{x}(t)$ is the input vector chosen for iteration t in the input data set, and $\epsilon(t)$ the learning rate at iteration t. The learning rate $\epsilon(t)$ defines the intensity of the adaptation, which is application dependent. Commonly $\epsilon(t)$ is constant or a decreasing scalar function of t. The term $\|\mathbf{r}_c - \mathbf{r}_i\|$ is the distance between neuron i and the winner neuron c, and $\sigma(t)$ is the standard deviation of the Gaussian neighbouring function. This neighbouring function ensures that the update's influence is the strongest for the BMU and is decreasing for other neurons through the map as the distance to the BMU increases. The resulting behaviour is illustrated in Figure 2a for a two-dimensional neurons grid.

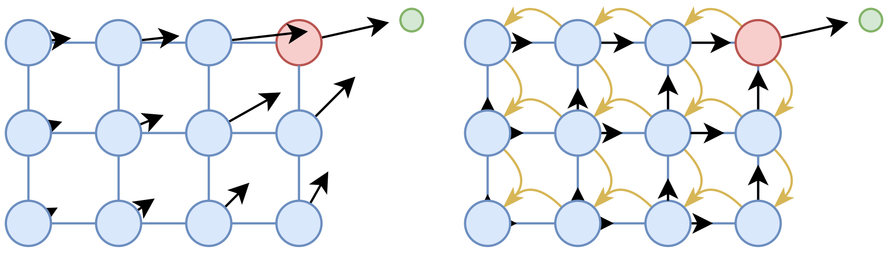


Figure 2. Illustration of the SOM and CSOM update mechanism, with BMU shown in red, new input in green, weight vectors represented as black arrows and update requests shown as yellow arrows. (**a**) SOM weight vectors update. (**b**) CSOM weight vectors update.

2.2. *CSOM: Cellular Self-Organizing Maps*

The Cellular Self-Organizing Maps, introduced in [14], aims to offer a cellular variant of the SOM algorithm with similar input set mapping properties but more suited to distributed applications. In a cellular environment, local interactions are privileged and neurons communicate only with their direct neighbors (Input vectors distribution and BMUs election being exceptions to this rule). These constraints raise a number of challenges for the implementation of the algorithm, but in return allow to distribute the algorithm without being affected by shared resources such as communication channels or central memory. One of the main differences between SOM and CSOM is in the update Equation (1) which is replaced by Equation (2) for the BMU and Equation (3) for the rest of the neurons of the network. Considering $\mathbf{m}_c(t)$ as the winning neuron for input $\mathbf{x}(t)$ of iteration t, the BMU update equation becomes:

$$\mathbf{m}_c(t+1) = \mathbf{m}_c(t) + \alpha(t)(\mathbf{x}(t) - \mathbf{m}_c(t)) \tag{2}$$

where $\alpha(t)$ is the learning rate parameter at iteration t. Once the BMU weights are updated, an update request is propagated from the BMU to neighboring units. These update requests will reach connected neurons, which will update their weight vectors towards the sender of the request and in turn send update request to their neighbors. This process is then repeated until the full network has been updated. Each non-BMU neuron i uses the following equation:

$$\mathbf{m}_i = \mathbf{m}_i + \alpha(t)\frac{1}{\#Infl(i)}\sum_{j\in Infl(i)}(\mathbf{m}_j - \mathbf{m}_i)e^{(-\frac{\sqrt{d}}{\eta(t)}\frac{hops(c,i)}{||\mathbf{m}_i-\mathbf{m}_j||})}, \tag{3}$$

where $hops(c,i)$ is the number of propagation hops necessary to reach neuron i from the BMU through synaptic connections, $Infl(i)$ is the set of the influential neurons of neuron i, and $\#Infl(i)$ is the number of influential neurons for neuron i. An influential neuron of neuron i is defined as the neuron from which neuron i received the update request. For a neuron with $hops(c,i) = h$, the influential neuron(s) will be every neuron j connected to i with $hops(c,j) = h - 1$. The parameter $\alpha(t)$ is the learning rate at time t, and $\eta(t)$ is the elasticity of the network at time t. The latter is modulated by $\sqrt{d}$ so as to take into account the range $[0, \sqrt{d}]$ of euclidean distances in dimension d. This process is illustrated in Figure 2 for a two-dimensional neurons grid. Average quantization error of the CSOM algorithm implemented in Pyhton can be found in [14] and results obtained on the SCALPsim simulator can be found in [18]. Further tests of CSOM applied to dynamic problems presented in [15] illustrate that the topological preserving clustering behaviour observed in SOM is also present in CSOM.

2.3. *ACSOM: Asynchronous Cellular Self-Organizing Maps*

In the original definition of the CSOM algorithm, it is considered that the update of the grid is starting from the BMU and is propagated from neuron to neuron in a cellular manner, each neuron updating its weight vector toward the one having requested the update, as illustrated in Figure 2b by the yellow arrows. This is the major difference with the SOM algorithm, where all neuron weights are updated toward the input vector.

As can be observed in Figure 2b, the non-BMU neuron i can receive several update requests from their neighbours, up to the number of dimension of the network. To account for the multiple influences, the weight update Equation (3) defined in [14] uses the set of influential neurons $Infl(i)$ (sum of the multiple influences) and then normalizes the result by the size of this set to compensate for the multiple influences that neuron i will receive from neighbors.

Given that the SCALP platform is composed of a set of asynchronous nodes and does not have a global time-base, a straightforward implementation of Equation (3) in hardware is not trivial because it requires to synchronize the update request from the set of influential neurons. It would require an ad-hoc mechanism able to detect two update requests related to the same input, and conversely to wait for future updates coming from different directions once an update request is received.

In order to overcome this problem, in this work we propose to introduce an asynchronous variant of CSOM, refered as ACSOM, by considering update requests from a single influential neighbor for a given weigh update. The CSOM BMU update Equation (2) stays the same, and the network update Equation (3) is replaced by Equation (4) through the removal of the influential neurons:

$$\mathbf{m}_i = \mathbf{m}_i + \alpha(t)(\mathbf{m}_j - \mathbf{m}_i)e^{(-\frac{\sqrt{d}}{\eta(t)}\frac{hops(c,i)}{||\mathbf{m}_i-\mathbf{m}_j||})} \tag{4}$$

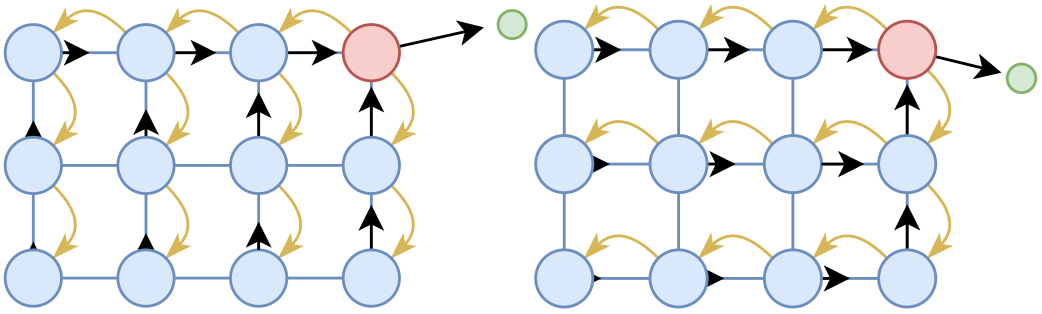
While avoiding the aforementioned complexity of multiple update requests per input, this simplification introduces a structural bias which results from the propagation algorithm used for the update request. If a neuron always receives the update request from the same neighbor, it will always update toward this neighbor. To mitigate this effect, it is proposed to tweak the broadcast algorithm used to propagate the update through the network over time. Over many iterations the resulting average update will be equivalent to the original CSOM update rule.

For each of the one, two and three dimensions network case, the units can be arranged to form a regular lattice for which it is possible to use the dimension-order routing algorithm [19], also referred as XY-routing in the two dimensional case. This routing algorithm considers the dimensions sequentially and always route the message in the direction having a non-null offset between the source and the destination for the current dimension. Once a dimension is null, the next one is considered and so on until the message reach its final destination. This simple algorithm has the advantage to be easy to implement in a fully distributed manner and can also be extended by the use of the broadcast algorithm defined in [20] which can propagate a message to all nodes in the network non-redundantly.

We propose to extend this broadcast algorithm by defining mechanisms for permuting the order in which the dimensions will be processed. At the emission, it is then possible to flag a broadcast message with a specific routing strategy which will be executed by the routing units. For the broadcast to be implemented in a distributed manner, each routing node needs to make a decision on which direction to propagate a message based on the arrival direction and on its mechanism flag. The direction can be formalized as {East, West} for the X dimension, {North, South} for the Y and {Up, Down} for the Z. Using this formalism, Table 1 gives the required propagation directions for the **XY** and **YX** mechanisms of the two dimensional case and Figure 3 illustrates their behaviours.

Table 1. Two dimensional update request propagation: output directions depend on the input direction and the current propagation mechanism.

	Incoming Direction			
	East	**West**	**North**	**South**
XY	W,N,S	E,N,S	S	N
YX	W	E	E,W,S	E,W,N



(**a**) (**b**)

Figure 3. Successive ACSOM weight vectors updates, with BMU shown in red, new inputs in green, weight vectors represented as black arrows and update requests shown as yellow arrows. (**a**) Input $\mathbf{x}(t)$ with $t_{BMU} \equiv 0 \mod 2 : \rho = \mathbf{XY}$. (**b**) Input $\mathbf{x}(t)$ with $t_{BMU} \equiv 1 \mod 2 : \rho = \mathbf{YX}$.

Table 2 shows the required propagation directions for the six mechanism permutations of the three dimensional case. Figure 4 illustrates the propagation for the mechanism **XZY**. Figure 5 shows the six possible paths followed by a broadcast message sent by the unit $(1,1,1)$ to unit $(3,3,3)$, highlighting only the nodes involved in the propagation. Note that the destination unit receives a message twice from each of the direction facing the emitter, ensuring a minimization of the structural bias over multiple updates in the case of ACSOM.

Table 2. Three dimensional update request propagation: output directions depends on the input direction and the current propagation mechanism.

	Update Request Incoming Direction					
	East	**West**	**North**	**South**	**Up**	**Down**
XYZ	W,N,S,U,D	E,N,S,U,D	S,U,D	N,U,D	D	U
YXZ	W,U,D	E,U,D	E,W,S,U,D	E,W,N,U,D	D	U
ZXY	W,N,S	E,N,S	S	N	E,W,N,S,D	E,W,N,S,U
XZY	W,N,S,U,D	E,N,S,U,D	S	N	N,S,D	N,S,U
YZX	W	E	E,W,S,U,D	E,W,N,U,D	E,W,D	E,W,U
ZYX	W	E	E,W,S	E,W,N	E,W,N,S,D	E,W,N,S,U

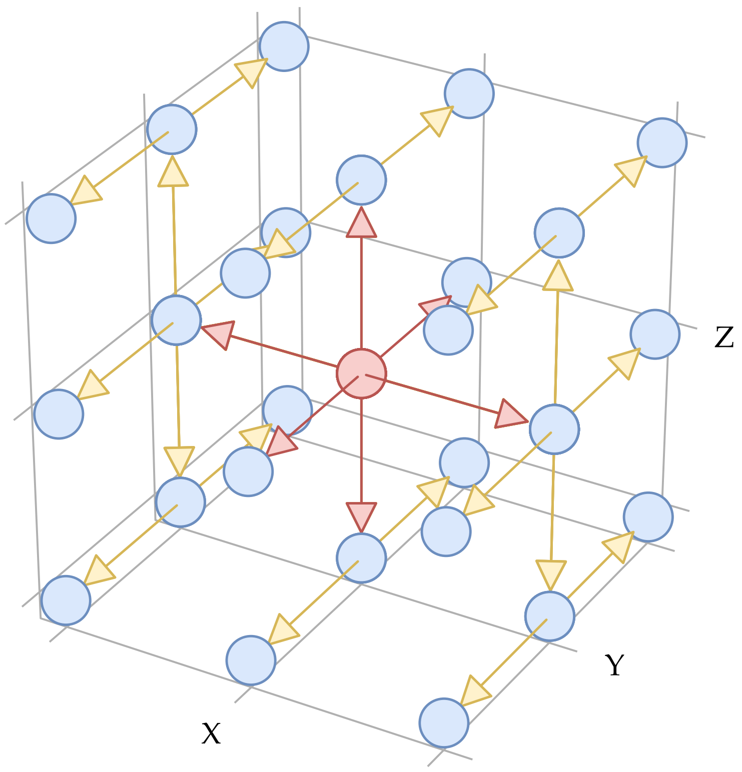


Figure 4. **XZY** broadcast mechanism example from the BMU at position $(2, 2, 2)$ shown in red, to all others neurons (in blue) of a $3 \times 3 \times 3$ network. Red arrows represent the BMU broadcast special case and yellow arrows represent the paths followed by **XZY** broadcast mechanism.

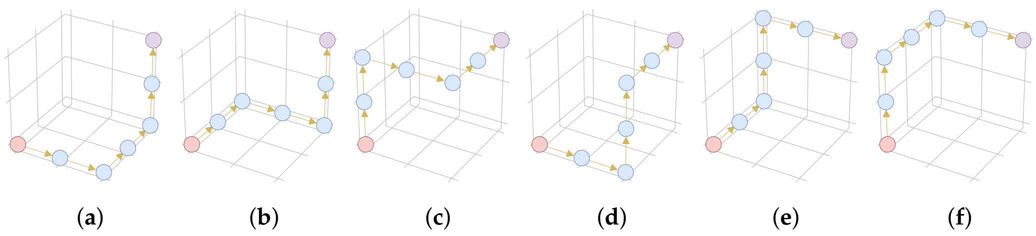


Figure 5. Illustration of the paths from unit $(1, 1, 1)$ (in red) to unit $(3, 3, 3)$ (in purple) with the 6 permutations of the broadcast mechanisms. (**a**) **XYZ**. (**b**) **YXZ**. (**c**) **ZXY**. (**d**) **XZY**. (**e**) **YZX**. (**f**) **ZYX**.

To select the broadcast mechanism ρ used for an update request propagation, each unit keeps track of the number of times t_{BMU} it has been elected as BMU. Once a neuron is designed as the BMU, it increments its own t_{BMU} and uses it to select the next mechanism in a round-robin manner as described in Equation (5) for two-dimensional cases and Equation (6) for three-dimensional cases.

$$\text{2D routing mechanism } \rho(t_{BMU}) = \begin{cases} \mathbf{XY}, & \text{if } t_{BMU} \equiv 0 \mod 2 \\ \mathbf{YX}, & \text{if } t_{BMU} \equiv 1 \mod 2 \end{cases} \tag{5}$$

$$\text{3D routing mechanism } \rho(t_{BMU}) = \begin{cases} \mathbf{XYZ}, & \text{if } t_{BMU} \equiv 0 \mod 6 \\ \mathbf{YXZ}, & \text{if } t_{BMU} \equiv 1 \mod 6 \\ \mathbf{ZXY}, & \text{if } t_{BMU} \equiv 2 \mod 6 \\ \mathbf{XZY}, & \text{if } t_{BMU} \equiv 3 \mod 6 \\ \mathbf{YZX}, & \text{if } t_{BMU} \equiv 4 \mod 6 \\ \mathbf{ZYX}, & \text{if } t_{BMU} \equiv 5 \mod 6 \end{cases} \tag{6}$$

The BMU then sends an update request flagged with the selected routing mechanism to **all** of its neighbors (BMU is a special case in which update request is performed to all directions). Then, all others units receiving the update request will perform their weight update and forward the update request while preserving the routing mechanism ρ elected by the BMU. Figure 3 illustrates two consecutive updates of a 2D network. In Figure 3a, the **XY** mechanism is used to propagate the update requests (in yellow), and in Figure 3b **YX** mechanism is used.

Figure 6a–e illustrate the behavior of a 6×6 neurons ACSOM network during the learning of 5000 random vectors uniformly distributed over a circle of radius 0.25 with center on $\{0.75, 0.75\}$, initial weights of the network being randomly initialized within the $[0, 0.3]$ interval. The learning rate α and the network elasticity η have been kept constant during the test. From Figure 6e, we can observe that topology preserving nature of traditional SOM is still present with ACSOM.

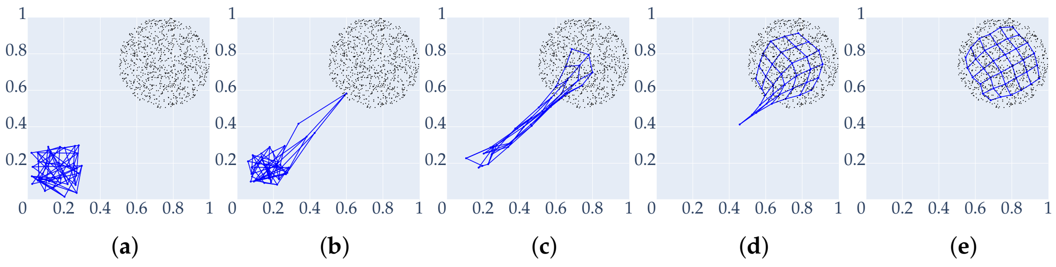


Figure 6. Evolution of a $6 \times 6 \times 1$ ACSOM network during training over 5000 input vectors, with neurons shown as blue dots, neuronal connections represented as blue lines and input population shown as black dots. (**a**) $t = 0$. (**b**) $t = 10$. (**c**) $t = 100$. (**d**) $t = 1000$. (**e**) $t = 5000$.

3. Architecture

This section presents the architecture of the floating point vector processing units, representing a single neuron of an ACSOM network. The unit is able to perform the distance and weight update computations required by ACSOM algorithm described in Section 2.3.

In the context of the SOMA project, these units are ultimately intended to be used over a Network on Chip (NoC) layer, spanning over multiple FPGA in three dimensions as illustrated in Figure 7a,b.

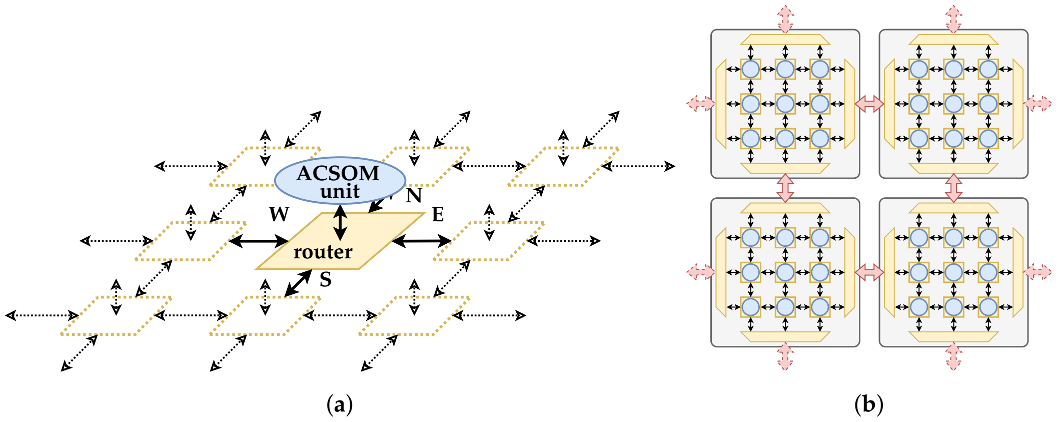


Figure 7. Example of a two-dimensional integration of ACSOM architecture in a SCALP array. (**a**) SCALP Network-On-Chip topology. (**b**) Multi-FPGA architecture. Black arrows represent AXI stream links, red arrows denote inter-FPGA high-speed serial links.

The NoC layer includes the equivalent of routing + link + physical layers in the OSI model, it guarantees link integrity between two neighbor nodes, and can redirect packets to neighbors in order to permit remote node communications without using intermediate computation nodes. The routing unit, illustrated in Figure 8, is composed of a routing decision logic and a set of multiplexers to supports the packet transmission. The AXI stream bus is used between two routing nodes, which enable easy encapsulation of the streams on high-speed serial link for inter-FPGA communication. This NoC is responsible for the propagation of the update requests as explained in Section 2.3, for inter-neuron communication and for BMU election trough a min-reduce algorithm. A more detailed of the packet types planned is presented below:

- Inputs propagation packets: contain the new input vector, routed as broadcast communication to all the units in the network.
- BMU election packets: contain the distance to the new input vector, routed using min-reduce algorithm from all nodes to the sender of the related input propagation packet.
- BMU notification packets: contains the input vector that the BMU won, routed as point to point.
- Update propagation packets: contain the weight of the unit asking for the update, routed as neighbor to neighbor packet using the propagation algorithm elected by the BMU as explained in Section 2.3.

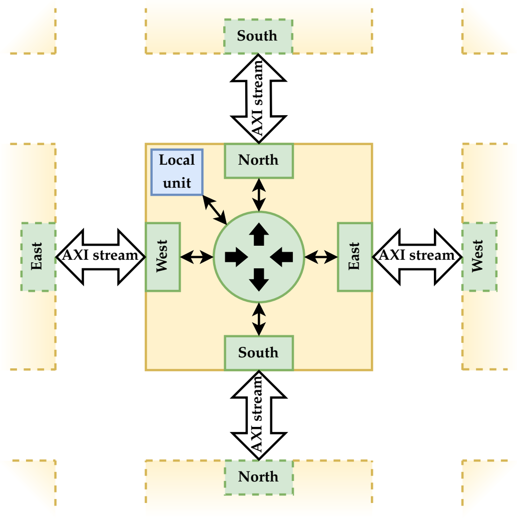


Figure 8. Two-dimensional SCALP Network-On-Chip router.

In the present work, the NoC is not further discussed and is implemented for the validation as a combinational routing layer providing the same routing mechanisms.

3.1. Arithmetic Operators

Most hardware SOM architectures found in literature usually propose a lot of simplifications for the implementation on the algorithm and an important quantization on data encoding with fixed point representations (Use of Manhattan distance instead of Euclidean distance, simplification of the neighborhood function by use of power of $1/2$ implemented with right-shifts). Such simplifications aim to reduce hardware resources at the cost of SOM accuracy loss, but in many cases they can result in very poor accuracy making the SOM useless for real-world applications. The presented architecture does not use such simplifications and instead, it is built around floating point operators generated by the Flopoco project [21]. Flopoco provides the flexibility to modify the number representation at the synthesis phase to be able to closely match the software model during the development phase, and to then reduce the precision for the exploitation to closely match the requirement of the application.

The Flopoco operator generator allows the creation of a VHDL description of pipelined arithmetic operators for arbitrary floating point mantissa and exponent sizes. The Flopoco floating point format is inspired from the IEEE-754 format, with some adaptations to simplify its implementation in FPGA. The two main differences are the encoding of special cases (infinities, zeroes and Not a Number (NaN)) in a separate bit-field, and the absence of subnormal numbers. In IEEE-754 format, a special case is used to represent very small numbers in which the fractional part is not implicitly prefixed by 1. Subnormal handling requires additional logic which can be spared using the Flopoco format and the difference in representation can be easily be alleviated by increasing the exponent size.

3.2. ACSOM Unit

The ACSOM unit is the hardware module representing a single neuron. It performs the following tasks:

1. Storing the weight vector associated with the unit.
2. Computing the euclidean distance to a provided input vector.
3. Updating its own weight vector when designated as BMU.
4. Initiating weight update mechanism after update as a BMU.
5. Updating of its own weight vector toward a neighbor when requested by said neighbor.

The number d of components in the input vector set is application dependent (The architecture can be configured at synthesis time for a specific component count) and it can span from few tens [14] to hundreds [22]. With such a high number, a fully parallel implementation of vector processing would be extremely expensive in term of hardware resources. Because of this, the proposed ACSOM unit architecture illustrated in Figure 9 process the input vector serially, in a pipelined manner in order to limit resource requirements. Pipelining is generated by Flopoco and its depth depends on the operator complexity, the floating point format and on the targeted operating frequency. After Flopoco operators generation, the resulting pipeline lengths are automatically taken into the control logic of the ACSOM unit. Input and output vector ports (used for input vectors, neighbor communication and weight initialisation) are of the width of the full vector for fast communication and registering. The $\mathbf{V}_{in}$ and $\mathbf{V}_{out}$ ports serve this purpose and allow initialization and retrieval of the unit weight vector easily. The x_{valid} signal notifies the unit when a new input vector is present in the $\mathbf{V}_{in}$ port and triggers a distance computation. The nm_{valid} indicates that an neighbor has requested an update and that its weight and grid coordinates are present on the $\mathbf{V}_{in}$, c_x, c_y and c_z ports. The m_{valid} input is used to initialize the unit's weight vector with the value presented on the $\mathbf{V}_{in}$ port.

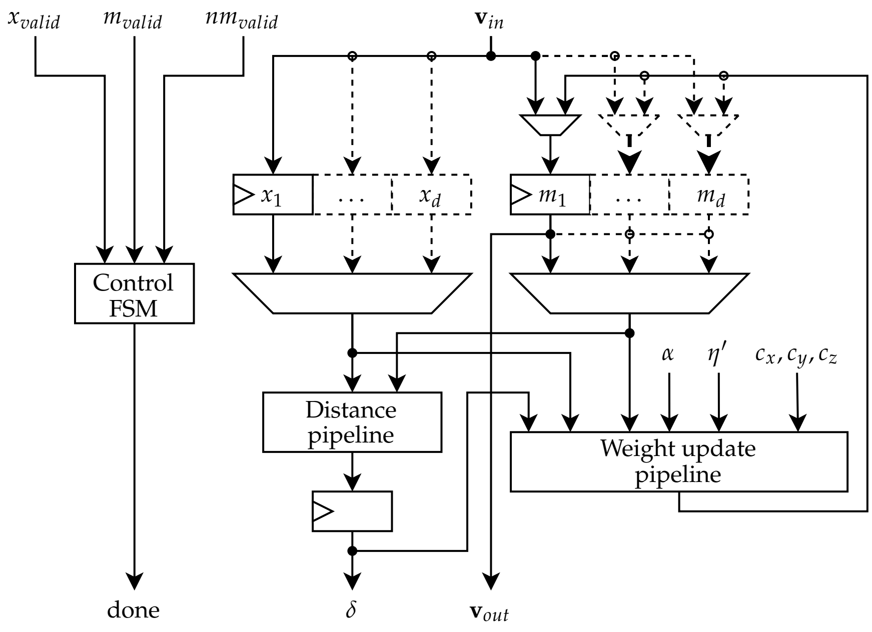


Figure 9. ACSOM unit architecture.

With this architecture choice, the increase of vector dimension only impacts the registers required to hold the input and weight vector, the rest of the architecture being unaffected in terms of logic resources. The general structure of the CSOM unit architecture is illustrated in Figure 9.

Two optimisations have been set up to further simplify implementation and reduce resources usage:

1. The term $-\frac{\sqrt{d}}{\eta(t)}$ used in the update Equation (4) is pre-computed in design-time and provided to the unit as $\eta'(t)$.
2. The Euclidean distance computation is implemented without the square root operation as this does not affect the BMU election and avoids implementing an expensive square root operator. The removal of the square root has a side effect by introducing a non-linearity in the computation of Equation (4) but it is compensated by the adaptation of the η parameter.

3.3. Distance Unit

The components $\mathbf{x}_i$ of the input and $\mathbf{m}_i$ of the unit weight are sequentially fed in this unit to compute the squared Euclidean distance δ between the two vectors as described in Equation (7).

$$\delta = \sum_{n=1}^{d} (\mathbf{m}_n - \mathbf{x}_n)^2 \tag{7}$$

The architecture of the distance unit is illustrated in Figure 10. It uses one subtractor, one multiplier and one adder. As the intermediate results need to be accumulated in a register until the vectors are processed in their entirety, the pipeline can not be fully used, and the pipeline interval of the unit is dependent on the pipeline length of the adder (The adder pipeline's length is of two clock cycles for the all the Flopoco formats used in this work).

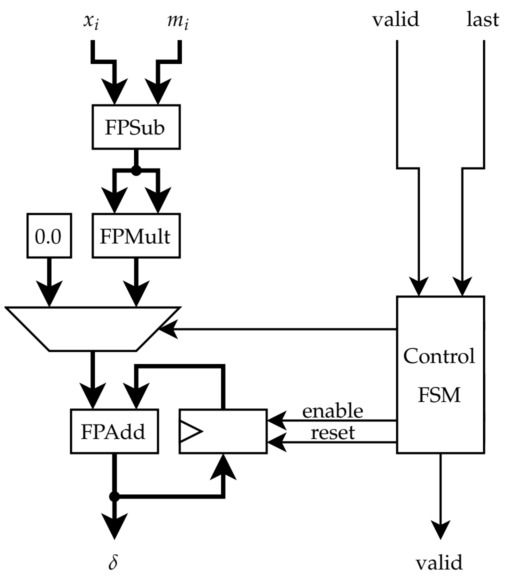


Figure 10. Euclidean distance pipeline.

3.3.1. Weight Update Unit

The weight update unit, illustrated in Figure 11, is used to update unit weight, both in the case of a BMU election and after an update request from a neighbor. It use one subtractor, one multiplier and one adder to implement Equation (8) in a pipelined manner.

$$\mathbf{m}_c = \mathbf{m}_c + \alpha(t)(\mathbf{x}(t) - \mathbf{m}_c) \tag{8}$$

The unit is able to update one weight component per clock cycle, with a pipeline latency of 9 clock cycles.

The datapath used to update the neuron's weight when selected as BMU (Equation (2)) can be re-used for the general grid update (Equation (4)) by computing a weight propagation coefficient WP combining the learning rate and the neighboring function. This coefficient is then used in place of the $\alpha(t)$ at the input of the multiplier.

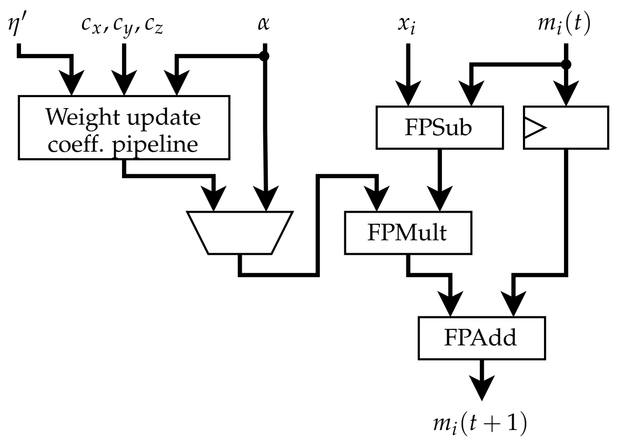


Figure 11. Weight update pipeline.

3.3.2. Weight Update Coefficient Unit

As explained above, this unit is used to compute the WP coefficient defined by Equation (9), used by the weight update unit in the case of a non-BMU weight update. The unit data path is illustrated in Figure 12. It is composed of an integer part (in the dotted-line square in the illustration), used to compute the number of hops to the BMU neuron c. The rest of the unit is implemented using a divider, one multiplier and one exponentiator.

$$WP = \alpha(t)e^{\left(-\frac{\sqrt{d}}{\eta(t)}\frac{hops(c,i)}{||\mathbf{m}_i - \mathbf{m}_j||}\right)} \tag{9}$$

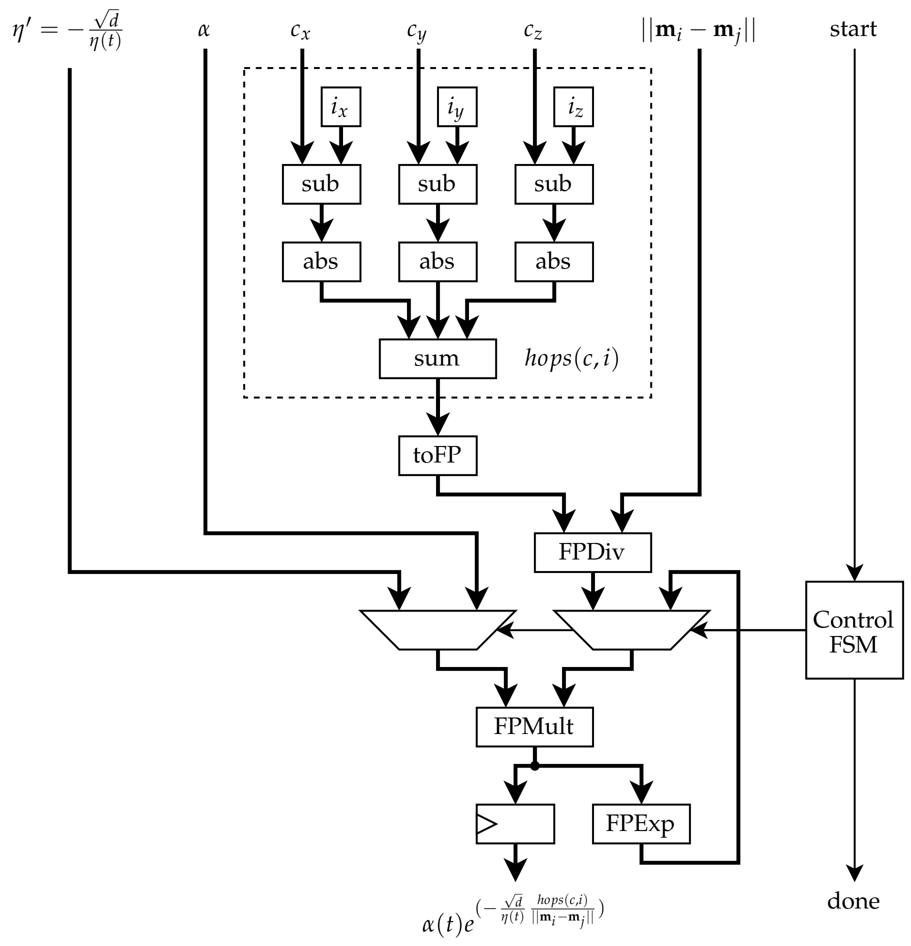

$\eta' = -\frac{\sqrt{d}}{\eta(t)}$
α
c_x
c_y
c_z
$||\mathbf{m}_i - \mathbf{m}_j||$
start
i_x
i_y
i_z
sub
sub
sub
abs
abs
abs
sum
$hops(c,i)$
toFP
FPDiv
Control FSM
FPMult
FPExp
$\alpha(t)e^{\left(-\frac{\sqrt{d}}{\eta(t)}\frac{hops(c,i)}{||\mathbf{m}_i - \mathbf{m}_j||}\right)}$
done

Figure 12. Weight update coefficient pipeline.

4. Experimental Setup and Results

This section presents the test methodologies used to evaluate both ACSOM and its VHDL implementation. First, ACSOM is compared with CSOM in terms of accuracy, then ACSOM hardware architecture is tested with different network sizes and number representations and finally a performance estimation of the architecture is presented and compared to existing results from the literature.

4.1. CSOM and ACSOM Comparison

As first step, a Python implementation of ACSOM have been used to characterize the variation of accuracy compared with a Python version of CSOM. Two networks of sizes 4x4 and 6x6 have been trained on a synthetic population in order to compare their behaviours. Given that the purpose of this test is to evaluate the effects of the modifications introduced by ACSOM (compared to CSOM), and not to qualify the absolute performance of the algorithms, the same arbitrary learning rate $\alpha = 0.3$ and network elasticity $\eta = 5.0$ have been used for both implementations. These parameters have been kept constant during the tests.

Both algorithms have been trained on a synthetic data-set of 1000 vectors with $d = 2$, distributed on two clusters. The first one with each component value randomly drawn from the $[0, 0.5]$ interval. The second cluster components where randomly chosen in the $[0.5, 1]$. Tests have been repeated 50 times for each algorithm and grid size with different weight initialization values to avoid favoring a test with a particularly advantageous initial position.

To compare the quantization performance of each implementation, a new set of 200 vectors have been randomly generated with the same distribution as used during training and used in a recall phase on the trained maps. For each vector, the Euclidean distance to the BMU has been calculated and stored in an error vector Δ. The Average Squared Error (ASE) defined in Equation (10) has been calculated for each test.

$$ASE(\boldsymbol{\Delta}) = \frac{\sum \Delta^2}{|\boldsymbol{\Delta}|} \tag{10}$$

The resulting ASE by grid size computed from the tests as well as the standard deviation (σ) of the error between each test presented in Table 3 show that the loss of accuracy is negligible.

Table 3. CSOM and ACSOM mean ASE and standard deviation comparison over 50 tests.

	4 × 4 Neurons		**6 × 6 Neurons**	
	Mean ASE	σ	**Mean ASE**	σ
CSOM	0.010760	0.000421	0.004567	0.000592
ACSOM	0.010842	0.000434	0.004576	0.000605

Figure 13 presents a qualitative comparison between CSOM and ACSOM in the form of a temporal evolution of a CSOM and ACSOM networks during a test. We can observe that in spite the fact that networks don't behave exactly in the same manner, the overall behavior and the resulting network topology can be considered qualitatively equivalent.

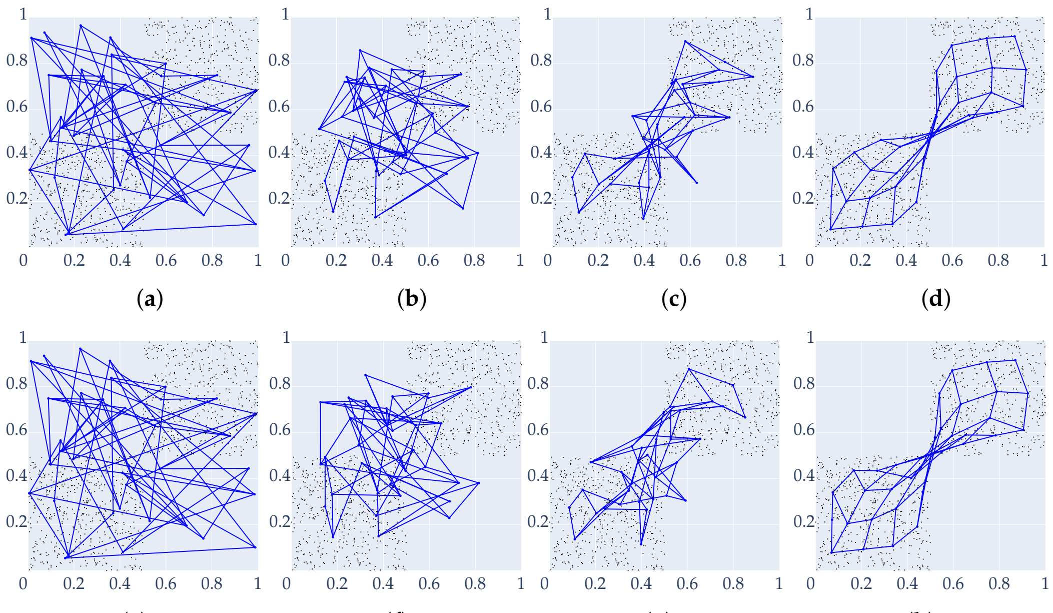


Figure 13. Comparative evolution of CSOM and ACSOM network during training over the same 1000 input vectors. (**a**) CSOM $t = 0$. (**b**) CSOM $t = 10$. (**c**) CSOM $t = 100$. (**d**) CSOM $t = 1000$. (**e**) ACSOM $t = 0$. (**f**) ACSOM $t = 10$. (**g**) ACSOM $t = 100$. (**h**) ACSOM $t = 1000$.

4.2. ACSOM Architecture

The ACSOM neuron architecture has been implemented in VHDL and simulated in Vivado 2020.2. The results have been compared to Python implementation to ensure correctness of individual computing units. The ACSOM neuron has also been physically verified by running on a SCALP board FPGA with a target frequency of 125 MHz. Higher frequency can easily be reached by modifying the Flopoco operators generation by increasing the pipeline depth (thus increasing the latency). This frequency has been fixed to remain synchronous with the underlying NoC. Four different number representation formats have been considered for implementation, Flopoco format FP5.4, FP6.8, FP6.16 and FP8.23; the first number indicates the number of bits for expressing the exponent and the second the number of bits for mantissa. The large FP8.23 format has been retained because it allows a direct comparison with the IEEE754 format (Sub-normal numbers case excepted) and has proven useful for the validation of the model. The FP5.4 format is the shortest supported by Flopoco. The number of Flip-flops (FF), Look-up-tables (LUT) and Digital Signal Processor blocks (DSP) required to implement a single ACSOM neuron have been obtained by synthesis and are presented in Table 4. Note that with a vector component with $d = 3$ or less, no Block RAM is needed, and that no DSP blocks are required for the two shortest formats FP5.4 and FP6.8.

Table 4. Logic resources requirement for implementation of one ACSOM neuron with $d = 3$.

	LUT	FF	DSP
FP5.4	888	663	0
FP6.8	1642	950	0
FP6.16	2938	1643	3
FP8.23	4172	2184	7

Once validated, a combinational routing layer has been used as a mock-up of the NoC layer in order to be able to simulate interactions of multiple neurons and provide a mechanism to elect the BMU. The simulation uses the NoC and unit network in a sequential manner; input distribution, distance computation, BMU election and weight update of all nodes are performed in a sequence before starting the process for a new input.

Three different network sizes have also been used to test the model: $4 \times 4 \times 2 = 32$, $4 \times 4 \times 3 = 48$ and $4 \times 4 \times 4 = 64$ neurons. This choice has been made arbitrarily with the goal of validating the 3D routing mechanism case and to observe the influence of the number of neurons on the quantization error.

In order to set the learning rate α and the network elasticity η, a simple parameter exploration has been performed and two set of values have been selected as optimal for the small and large number representation. These parameters have been kept constant during the tests.

For each combination of these parameters, ACSOM have been trained on a synthetic data-set of 1000 vectors with $d = 3$, distributed on two clusters. The first one with each component value randomly drawn from the $[0, 0.5]$ interval. The second cluster components where randomly chosen in the $[0.5, 1]$ interval as illustrated in Figure 14.

Tests have been run 10 times for each parameter combination with different weight initialization values. All input and initialization values have first been quantified to the 5.4 format to avoid test bias towards the large representation format. This allows to really evaluate the effect of the representation quantization on the algorithm implementation instead of the effect on representing the input data.

To compare the quantization performance of each implementation, a new set of 200 vectors have been randomly generated with the same distribution as used during training and used in a recall phase on the trained maps to compute the resulting ASE.

Figure 15 summarizes the design space exploration of 12 different candidate architectures. It presents the ASE and hardware cost obtained by different SOM implementations on which two main parameters are modified: the topology and the floating point representation. It must be noted that the hardware cost is only considering the number of LUTs, this metric may be misleading because large representations are also using DSP blocks (not represented in the graph), while small representations are not. Despite this limitation, we considered LUT counting to be an acceptable measure of hardware complexity.

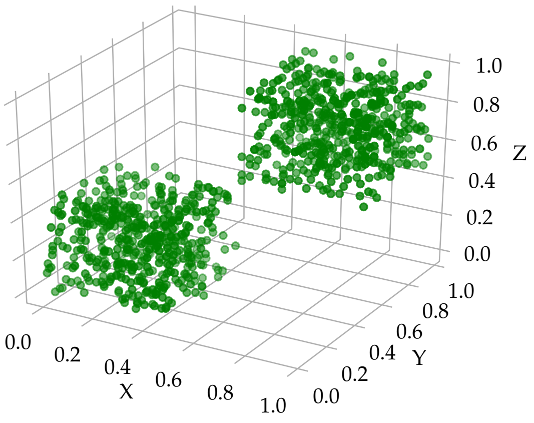


Figure 14. Input vector clusters.

From Figure 15 we can observe that, unsurprisingly, larger networks (4 × 4 × 4) obtain better ASE results than smaller networks (4 × 4 × 2) by incurring a higher hardware cost. The same applies for larger format representations, a format FP8.23 has a better accuracy and higher cost than a more compact one like FP6.8.

However, there are other less predictable conclusions we can draw. The results show that between the two larger FP formats (FP8.23 and FP6.16), the ASE difference is negligible. This confirms that there is no need to use large formats in this particular problem. Moreover, for large maps (4 × 4 × 4), the FP6.8 format still presents nearly the same ASE without using DSP blocks and using only 55% of the LUTs of the FP6.16 format implementation. Going further, by tolerating a 8% increase in ASE, the 4 × 4 × 4 neurons map that uses format FP5.4 offers a further 53% reduction of LUTs over FP6.8.

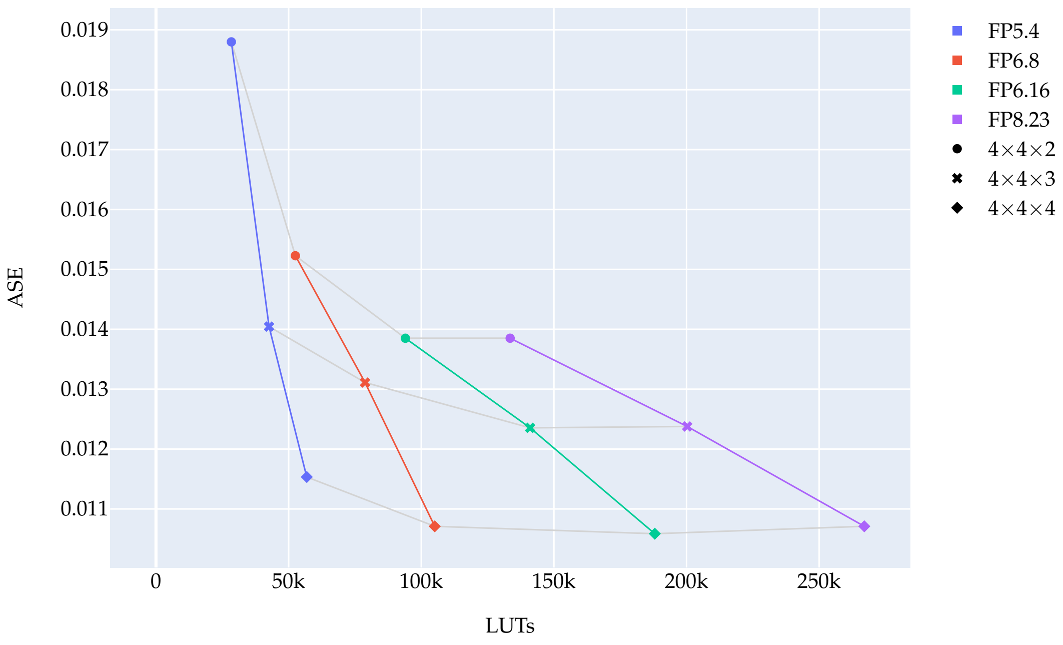


Figure 15. Average squared error versus LUTs.

A more interesting and less expected observation of the results, suggests that using a larger grid with smaller resolution yields a much better ASE with a lower hardware complexity in most of the cases. The comparison between the ASE obtained by the 4 × 4 × 4 SOM with FP5.4 and the 4 × 4 × 3 with FP8.23 shows that it is worthless to spend hardware resources to increase floating point representation, and that it is better to increase the number of SOM nodes by keeping a low representation format.

4.3. Performance Comparison

Speed performance of SOM implementations are usually presented in term of Connection Update Per Second (CUPS). In Table 5 we present the CUPS achievable by our ACSOM implementation based on the simulated computation latency and the estimated latency of the underlying NoC architecture.

Our results are compared to three FPGA SOM implementations which are of particular interest in our case because they are all based on a NoC-based FPGA implementations [10,11,23].

The network size, vector dimension and representation of ACSOM have been adapted in each test to match as close as possible the reference implementations. For each of the three estimations, ACSOM activity has been modeled as the exact sequence used during the simulation, with the worst case estimated NoC latency added to all the routing operation in order to obtain MCUPS figures.

Table 5. Performance comparison.

Work	Network Size	Vectors Dimension	Vectors Representation	Frequency	MCUPS
[10]	5 × 5	3	16-bit	1.51 MHz	113
Our work	5 × 5	3	FP6.16	125 MHz	14
[11]	5 × 5	32	8-bit	200 MHz	724
Our work	5 × 5	32	FP6.8	125 MHz	65
[23]	16 × 16	32	8-bit	294 MHz	28,282
Our work	16 × 16	32	FP6.8	125 MHz	209

Compared with [10,11], ACSOM is one order of magnitude slower, and two orders of magnitude slower than [23]. This is to be expected as we explicitly choose to not implement the traditional speed optimizations, but mainly because of the floating point representation used and its asynchronous nature. It is also to be noted that, as mentioned above, the 125MHz operating frequency used for these estimations is not the maximum that could be reached by this implementation, and that higher operating frequency coupled with large vector dimension would benefit of the pipelined operators and reduce the impact of the expected NoC latency.

5. Conclusions

In this paper we have presented ACSOM and its hardware implementation, an asynchronous version of the Cellular Self-Organising Maps algorithm. The main novelty introduced in the ACSOM algorithm, namely the asynchronous weight update mechanism, has been validated with different network configurations including 3D topologies. Indeed, it has been shown here that the behaviour during learning is similar to the one described in the original CSOM paper [14].

Concerning the presented hardware architecture of ACSOM, the FPGA synthesis results conducted during this work show that a high number of neurons is preferable to a precise representation. The advantages of low precision floating point representation over usual integer or fixed point still have to be proved in future work. The speed performances presented in the previous chapter are unprecedented given that, up to our knowledge, this work is the first using floating point representation for SOM hardware architectures and is also the first implementation of the Cellular SOM algorithm. The straightforward performance comparison shows a lower number of CUPS. However, it is worth to highlight the increased representation capability of the floating point representation mainly for fine tuning purposes.

Future Work

The next step of this work is the merging of the ACSOM unit with the SCALP NoC routing layer. Such integration will result in the architecture described in Figure 7b. This shall allow the validation of the scalability of the system over an asynchronous 3D array of SCALP boards. The NoC routing layer will allow the transmission of messages from one unit to any other unit in the system in a seamless manner for the different units. Local communications will obviously be favored in the case of an ACSOM implementation in order to reduce the risk of routing congestion, but global communication will still be required for the broadcasting of the input vectors. A special communication primitive will also be required for the BMU election under the form of a min-reduce operation. This step will permit to verify the CUPS performances obtained in FPGA.

ACSOM embodiment will enable real-time weight updates driven by physical input data. We have ongoing work on building extension boards permitting to endow SCALP with sensors (cameras, microphones, touch). Such multiplicity of input data from different sensors will permit us to apply ACSOM to multi-modal learning that could be of particular interest for tracking with incomplete or faulty input data.

Performance improvement will be addressed by parallelising and pipelining the processing of multiple input vectors. Several nodes will serve as data inputs (i.e. different sensors connected to different boards), different data input broadcast will be performed simultaneously in an asynchronous manner. The use in parallel of the ACSOM nodes processing pipelines will permit to handle a multiplicity of states for the ongoing learning iterations.

Author Contributions: Conceptualization, A.U., Q.B. and J.S.; methodology, Q.B. and A.U.; software, Q.B.; validation, Q.B. and A.U.; formal analysis, Q.B.; investigation, Q.B. and J.S.; resources, Q.B.; data curation, Q.B.; writing—original draft preparation, Q.B.; writing—review and editing, A.U. and Q.B; visualization, Q.B.; supervision, A.U.; project administration, A.U.; funding acquisition, A.U. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Swiss National Science Foundation (SNSF-175720) and French national Research Agency (ANR-17-CE24-0036).

Data Availability Statement: The VHDL sources and Pyhton scripts used to produce the data presented in this work are available at https://gitlab.com/scalp_hw/scalp_acsom (accessed on 16 November 2021).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ASE	Average Squared Error
ACSOM	Asynchronous Cellular Self-Organizing Maps
BMU	Best Matching Unit
CSOM	Cellular Self-Organizing Maps
CUPS	Connetion Updates Per Second
DSP	Digital Signal Processor
FF	Flip-flop
FPGA	Field-Programmable Gate Array
LUT	Look-up-table
LVDS	Low Voltage Differential Signaling

NoC	Network On-Chip
SOM	Self-Organizing Maps
SoC	System On-Chip
VHDL	VHSIC Hardware Description Language

References

1. Khacef, L.; Girau, B.; Rougier, N.P.; Upegui, A.; Miramond, B. Neuromorphic hardware as a self-organizing computing system. In Proceedings of the WCCI 2018—IEEE World Congress on Computational Intelligence, Workshop NHPU: Neuromorphic Hardware In Practice and Use, Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–4.
2. Kohonen, T. The self-organizing map. *Proc. IEEE* **1990**, *78*, 1464–1480. [CrossRef]
3. Araújo, C.F.; Araújo, A.F.R. Self-organizing Maps for Speech Recognition. In *Anais do 11 Congresso Brasileiro de Inteligência Computacional*; Braga, A.d.P., Bastos Filho, C.J.A., Eds.; SBIC: Porto de Galinhas, Brazil, 2013; pp. 1–6.
4. Amerijckx, C.; Verleysen, M.; Thissen, P.; Legat, J.D. Image compression by self-organized Kohonen map. *IEEE Trans. Neural Netw.* **1998**, *9*, 503–507. [CrossRef] [PubMed]
5. Hikawa, H.; Maeda, Y. Improved Learning Performance of Hardware Self-Organizing Map Using a Novel Neighborhood Function. *IEEE Trans. Neural Netw. Learn. Syst.* **2015**, *26*, 2861–2873. [CrossRef] [PubMed]
6. Pena, J.; Vanegas, M.; Valencia, A. Digital Hardware Architectures of Kohonen's Self Organizing Feature Maps with Exponential Neighboring Function. In Proceedings of the 2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006), San Luis Potosi, Mexico, 20–22 September 2006, pp. 1–8. [CrossRef]
7. Younis, B.M.; Mahmood, B.; Ali, F.H. Reconfigurable Self-Organizing Neural Network Design and it's FPGA Implementation. *AL Rafdain Eng. J.* **2009**, *17*, 99–115.
8. Brassai, S.T. FPGA based hardware implementation of a self-organizing map. In Proceedings of the IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, Tihany, Hungary, 3–5 July 2014; pp. 101–104. [CrossRef]
9. de Abreu de Sousa, M.A.; Del-Moral-Hernandez, E. Comparison of three FPGA architectures for embedded multidimensional categorization through Kohonen's self-organizing maps. In Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, USA, 28–31 May 2017; pp. 1–4. [CrossRef]
10. de Abreu de Sousa, M.A.; Del-Moral-Hernandez, E. An FPGA distributed implementation model for embedded SOM with on-line learning. In Proceedings of the 2017 International Joint Conference on Neural Networks (IJCNN), Anchorage, AK, USA, 14–19 May 2017; pp. 3930–3937. [CrossRef]
11. Abadi, M.; Jovanovic, S.; Ben Khalifa, K.; Weber, S.; Bedoui, M.H. A Scalable Flexible SOM NoC-Based Hardware Architecture. In *Advances in Self-Organizing Maps and Learning Vector Quantization*; Merényi, E., Mendenhall, M.J., O'Driscoll, P., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 165–175.
12. Hikawa, H. FPGA implementation of self organizing map with digital phase locked loops. *Neural Netw.* **2005**, *18*, 514–522.
13. Appiah, K.; Hunter, A.; Meng, H.; Yue, S.; Hobden, M.; Priestley, N.; Hobden, P.; Pettit, C. A binary Self-Organizing Map and its FPGA implementation. In Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, Atlanta, GA, USA, 14–19 June 2009; pp. 164–171. [CrossRef]
14. Girau, B.; Upegui, A. Cellular Self-Organising Maps-CSOM. In Proceedings of the WSOM'19—13th International Workshop on Self-Organizing Maps and Learning Vector Quantization, Clustering and Data Visualization, Barcelona, Spain, 26–28 June 2019. [CrossRef]
15. Novac, P.E.; Upegui, A.; Barrientos, D.; Sousa, C.; Rodriguez, L.; Miramond, B. Dynamic Structural and Computational Resource Allocation for Self-Organizing Architectures. In Proceedings of the 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Glasgow, UK, 23–25 November 2020; pp. 1–4.
16. Upegui, A.; Girau, B.; Rougier, N.; Vannel, F.; Miramond, B. Pruning self-organizing maps for cellular hardware architectures. In Proceedings of the 2018 IEEE NASA/ESA Conference on Adaptive Hardware and Systems (AHS), Edinburgh, UK, 6–9 August 2018; pp. 272–279.
17. Vannel, F.; Barrientos, D.; Schmidt, J.; Abegg, C.; Buhlmann, D.; Upegui, A. SCALP: Self-configurable 3D Cellular Adaptive Platform. In Proceedings of the 2018 IEEE Symposium Series on Computational Intelligence (SSCI), Bangalore, India, 18–21 November 2018; pp. 1307–1312. [CrossRef]
18. Barrientos, D.; Sousa, C.; Upegui, A.; Girau, B. SCALPsim, a tool for modeling asynchronous Self-Organizing 3D NoC architectures. In Proceedings of the ICECS 2020, 27th IEEE International Conference on Electronics Circuits and Systems, Glasgow, UK, 23–25 November 2020.
19. Duato, J.; Yalamanchili, S.; Ni, L.M. *Interconnection Networks: An Engineering Approach*; Morgan Kaufmann: San Francisco, CA, USA, 2003.
20. Sullivan, H.; Bashkow, T.R. A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I. In Proceedings of the 4th Annual Symposium on Computer Architecture, College Park, MD, USA, 23–25 March 1977; Association for Computing Machinery: New York, NY, USA, 1977; pp. 105–117. [CrossRef]
21. de Dinechin, F.; Pasca, B. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Des. Test Comput.* **2011**, *28*, 18–27. [CrossRef]

22. Khacef, L.; Miramond, B.; Barrientos, D.; Upegui, A. Self-organizing neurons: Toward brain-inspired unsupervised learning. In Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–9. [CrossRef]
23. Ben Khalifa, K.; Blaiech, A.; Abadi, M.; Bedoui, M. A New Hardware Architecture for Self-Organizing Map Used for Colour Vector Quantization. *J. Circuits Syst. Comput.* **2019**, *29*, 2050002. [CrossRef]