Digital Implementation of On-Chip Hebbian Learning for Oscillatory Neural Network

Edgar Luhulima[1], Madeleine Abernot[2], Federico Corradi[1], and Aida Todri-Sanial[1,2] (a.todri.sanial@tue.nl)
[1]*Eindhoven Univ. of Technology*, Eindhoven, Netherlands, [2]*LIRMM, Univ. of Montpellier, CNRS*, Montpellier, France

***Abstract*—This work proposes a digital implementation of an Oscillatory Neural Network (ONN) in a Field-Programmable Gate Array (FPGA), demonstrating excellent associative memory capabilities. This work goes beyond previous implementations by enabling on-chip learning directly in the FPGA. More specifically, we implement on-chip Hebbian learning, and we compare three different design strategies. The first strategy takes advantage of a System-on-Chip (SoC) composed of a Processing System (PS) and Programmable Logic resources (PL) to integrate Hebbian learning in PS. The two other strategies implement the Hebbian learning directly in PL. We compare the three different design strategies on a digit recognition task in terms of accuracy, utilization, execution time, and maximum frequency. We show that implementing Hebbian learning in PL gives more advantages in terms of resource utilization and latency than implementing Hebbian in PS with several orders of magnitude because the weight matrix computation is performed in hardware. Moreover, we develop an application interface to demonstrate the pattern learning and recognition capabilities of our digital ONN implementation.**

Keywords: *Artificial intelligence, auto-associative memory, pattern recognition, oscillatory neural network, FPGA implementation, Hebbian learning*

I. Introduction

In the past few years, there has been a rising trend of employing machine learning instead of traditional computing to perform tasks that previously seemed impossible to be performed by conventional machines. For example, machine learning technique as Artificial Neural Networks (ANN) has been used to perform object recognition [1], character recognition [2], and even speech recognition [3]. Different techniques have been explored to keep up with the growing constraints, especially with embedded applications. One of these constraints is power. One approach to employ a neural network with low power that suits embedded devices is Oscillatory Neural Network (ONN) [4]. ONN is a network of coupled oscillators with unique phase and frequency dynamics that can be used to perform low-power parallel computation [5, 6]. ONN has good associative memory capability [7] such that it can memorize patterns and retrieve them from corrupted input information. Also, different solutions has been explored to implement ONN in hardware, in analog [8], mixed analog and digital [9], or even fully-digital [10]. This paper focuses only on the digital implementation of the ONN from [10]. The current digital implementation is implemented in an FPGA. This digital implementation showcases a good associative memory capability of the ONN. However, the learning part of this ONN is still performed off-chip, meaning the weights are hard-coded every time the ONN is trained.

This work was supported by the European Union's Horizon 2020 research and innovation program, EU H2020 NEURONN (www.neuronn.eu) project under Grant 871501 and Horizon EU research and innovation program, Horizon EU PHASTRAC (https://phastrac.eu) project under Grant no. 101092096.

979-8-3503-1175-4/23/$31.00 ©2023 IEEE

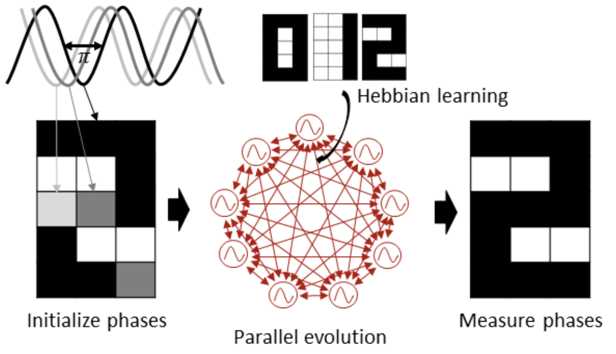


Fig. 1: Oscillatory Neural Network computing paradigm.

Recently, authors in [11], proposed a solution to enable on-chip learning, with the digital ONN taking advantage of the processing system (PS) of a Zynq processor. They performed on-chip learning with Hebbian and Storkey learning rules on a 15-neuron ONN configured for digit recognition. In this paper, we propose two other solutions to implement on-chip Hebbian learning [12] using the Programmable Logic (PL) resources of the Pynq FPGA board, and we compare the three solutions in terms of resource utilization. Finally, we also propose an application interface to demonstrate the pattern learning and recognition performances of our digital ONN solutions on a digit recognition application.

This paper is organized as follows. Section II describes the ONN computing paradigm, its auto-associative memory properties, and the digital implementation on FPGA. Then, Section III explains the different design approaches to implement on-chip Hebbian learning digitally. After, Section IV highlights the measurement results which validate each design, and compares the three design solutions. Section IV also presents the application demonstration on digit recognition. Finally, Section V discusses the advantages and limitations of the different solutions.

II. Oscillatory Neural Networks

A. Computing paradigm

Oscillatory Neural Networks (ONNs) [13, 14, 15] are brain-inspired computing models emulating neural oscillations from the brain. In ONNs, each neuron is an oscillator coupled with synaptic elements [16]. In this work, we consider phase-based ONNs which encode information in the phase relationship between oscillators. For example, if we consider bipolar information $\{-1, 1\}$, a $\{-1\}$ represents an oscillator with 0^o phase, while a $\{+1\}$ represents an oscillator with 180^o phase. Phase-computing ONNs use the natural phase synchronization behavior of coupled oscillators to compute in parallel. Thus, using phase computing allows for fast and parallel computation while possibly reducing the voltage amplitude and so limiting the power consumption [5], making ONN attractive for edge AI. During training, couplings between oscillators are configured depending on the task to solve. Then, inference starts with the initialization of each oscillator with the input phase information. Then, thanks to coupling, oscillators interact among them and phases evolve in time until stabilization. Reading the stable oscillator's phases gives the ONN output solution, see Figure 1.

B. *Auto-associative memory*

ONN configured with fully-connected architecture using unsupervised learning rules is known to perform auto-associative memory or pattern recognition tasks [7], like in Hopfield Neural Networks (HNNs) [17]. In this case, the network memorizes patterns in its coupling using some unsupervised learning rules such that when the network is initialized with a corrupted input pattern, it will evolve and stabilize to one of the memorized patterns. The main learning algorithm used to configure ONN or HNN for pattern recognition is the unsupervised Hebbian learning rule [12]. Hebbian configures the synaptic weights W_{ij} between neuron i and neuron j following:

$$W_{ij}^p = \sum_p \sigma_i^p \sigma_j^p \tag{1}$$

with $W_{ii} = 0$ and p the number of memorized or training patterns σ. There are other unsupervised learning rules which can be used to train ONN for pattern recognition and give better capacity, meaning being able to learn and retrieve more patterns. However, Hebbian is the simplest algorithm, and so the easiest to implement. Thus, in this paper, we only consider the Hebbian learning rule.

C. *Digital ONN implementation*

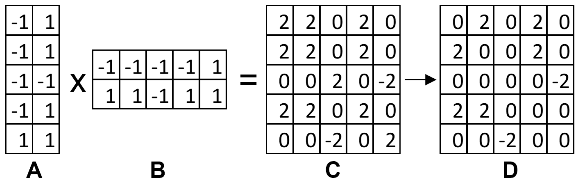
In this work, we focus on enabling on-chip learning for an ONN implemented on FPGA. A first fully-digital ONN design was introduced in [10] without the on-chip learning capability. In [10] each neuron is a phase-controlled digital oscillator allowing 16 phase stages, and each synapse is a 5-bits signed register. Then, a first solution to perform on-chip learning with the previous digital ONN design was introduced in [11] proposing to take advantage of the PS of a Zynq processor to implement Hebbian and Storkey learning rules in a 15-neuron ONN configured for digits recognition. In this work, we study two other solutions to enable on-chip Hebbian learning with the digital ONN on FPGA and compare the scalability of each solution.

III. DESIGNS AND METHODS

A. *Design exploration*

The weights for the ONN determine the coupling between two distinct neurons. In the current implementation, these weights are computed off-chip using the Hebbian learning rule, see Figure 2. This learning rule requires arranging the learning patterns into a column vector. Each element of the vector is bipolar (-1/1.) Multiple learning patterns translate to multiple column vectors, which are then appended to form a matrix. The matrix rows are equal to the number of neurons in the ONN. The matrix is multiplied with its transpose, and the diagonal elements of the product are set to 0, see Equation (1). The resulting square matrix defines the weights for the ONN of which the size is determined by the number of neurons. Figure 2 and Equation (1) illustrate the computation of the weights. The term Hebbian learning and weights computation are used interchangeably in this paper.

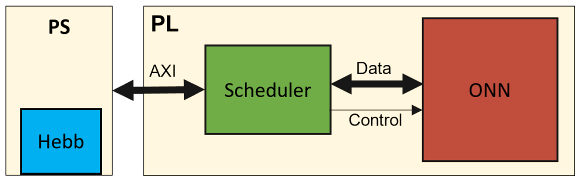
The weights are embedded into the hardware description code of the ONN which is then programmed into the FPGA. This limits the flexibility of modifying the learning patterns, therefore the weights, of the ONN after the FPGA is programmed. It prevents the possibility of on-chip learning. This chapter describes the different design variations that were explored in order to realize the implementation



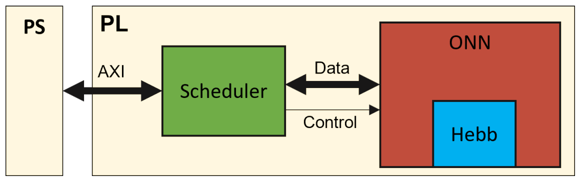
$$\underset{\textbf{A}}{\begin{array}{|c|c|}\hline -1 & 1\\ \hline -1 & 1\\ \hline -1 & -1\\ \hline -1 & 1\\ \hline 1 & 1\\ \hline\end{array}} \times \underset{\textbf{B}}{\begin{array}{|c|c|c|c|c|}\hline -1 & -1 & -1 & -1 & 1\\ \hline 1 & 1 & -1 & 1 & 1\\ \hline\end{array}} = \underset{\textbf{C}}{\begin{array}{|c|c|c|c|c|}\hline 2 & 2 & 0 & 2 & 0\\ \hline 2 & 2 & 0 & 2 & 0\\ \hline 0 & 0 & 2 & 0 & -2\\ \hline 2 & 2 & 0 & 2 & 0\\ \hline 0 & 0 & -2 & 0 & 2\\ \hline\end{array}} \rightarrow \underset{\textbf{D}}{\begin{array}{|c|c|c|c|c|}\hline 0 & 2 & 0 & 2 & 0\\ \hline 2 & 0 & 0 & 2 & 0\\ \hline 0 & 0 & 0 & 0 & -2\\ \hline 2 & 2 & 0 & 0 & 0\\ \hline 0 & 0 & -2 & 0 & 0\\ \hline\end{array}}$$

Fig. 2: Hebbian learning rule in the form of matrix multiplication. **(A)** Matrix of 2 learning patterns for 5 neurons. **(B)** Transposed matrix. **(C)** Square matrix product of matrix multiplication. **(D)** Weights with diagonal elements equal to 0.

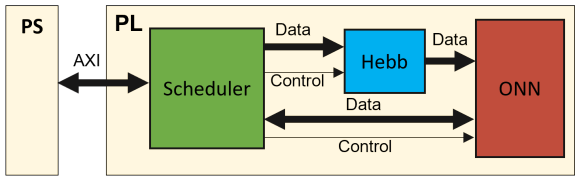
of on-chip Hebbian learning. This allows for the computation of the weights for the ONN to be executed on chip. In this paper, we discuss three designs. These designs make use of the two units in modern FPGAs, the PS and the PL. Figure 3 illustrates the block diagram of each design variation. The first design, introduced in [11], incorporates Hebbian learning, indicated as the Hebb block in the diagram, as a part of the application software running on the PS. The second and third design integrate Hebbian learning as a hardware block in the PL.



(a) Hebbian learning in PS



(b) Hebbian learning in PL



(c) Hebbian learning in PL with a dedicated hardware

Fig. 3: Design variations of the on-chip Hebbian learning

1) *ONN with Hebbian learning in PS:* The Hebbian learning in PS method, from [11], is designed as a function that computes the weights for the ONN in the application software. It is depicted in Figure 3a. The application software applies the Hebbian learning rule to each learning vector sequentially and accumulates the result in a weight matrix. This is a different technique than the one used in off-chip Hebbian learning. Instead of forming a matrix from different learning vectors, learning vectors are processed sequentially to produce a weight matrix. This process requires a multiplication between a vector with its transpose and an accumulation with previous results. After all learning vectors are processed, the final weight matrix is transmitted from PS to the Scheduler block in PL

through the AXI interface. The Scheduler is a control unit for the ONN block. It bridges the communication between the application software and the ONN. It writes the weights to the ONN during training. During inference, it writes the test pattern to the ONN and reads back the inference result, which is then transmitted to the application software. While this design is a solution for on-chip learning, the weights computation is still executed in software which is relatively slow compared to hardware. To maximize the benefit of implementation in an FPGA, we explored other designs that directly exploit PL hardware during the learning process.

2) ONN with Hebbian learning in PL: The Hebbian learning in PL integrates the learning or weights computation as hardware to the ONN block as shown in Figure 3b. Since the learning is executed in hardware, the performance in terms of execution time should improve. The weights computation is added to the hardware description language of the ONN. In contrast to the Hebbian learning in PS, the application software sequentially transmits the learning vectors instead of the weight matrix to the Scheduler. The Scheduler, which controls the ONN, writes the vectors to the ONN during training. The processing of the vectors is consequently executed sequentially in the ONN block. It multiplies a vector with its transpose, then the result is accumulated. During inference, it writes the test pattern to the ONN block and reads back the inference result. The drawback of this design is that it lacks parallelism since the multiplication and accumulation for each element of the vectors are performed serially. It also utilizes more resources in the FPGA in comparison to the previous design because of the addition of the learning block as hardware.

3) ONN with Hebbian learning in PL with a dedicated hardware: The last design uses a separate dedicated hardware for Hebbian learning, as shown in Figure 3c, whose purpose is to add parallelism to the weight matrix computation. This dedicated hardware, represented by the Hebb block in the diagram, is a multiply and accumulate (MAC) unit. During training, the Scheduler receives the learning vectors from the application software and writes them to the MAC unit. The MAC unit has two data inputs, A and B. It is illustrated in Figure 4 for a learning vector of size 5. Elements of the learning vector are pushed serially into the multiplier through input A. The elements of the transposed vector are pushed in parallel to input B. As an improvement to the previous design, the multiplication result for a single column of the weight matrix is computed in a single clock cycle. This result will be accumulated with the previous results in the accumulator. The accumulator also acts as a weights buffer for the ONN. The Scheduler, similar to previous designs, writes the test pattern to the ONN, reads back the inference result and transmits it to the application software during inference. Because some computations are parallelized, this design should yield a better execution time for learning than the previous design.

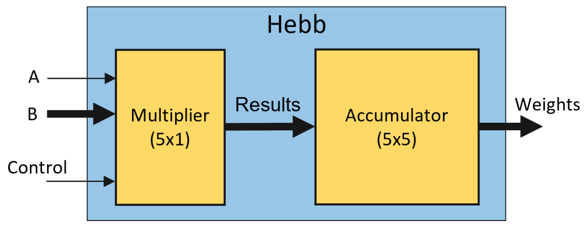


Fig. 4: MAC unit for a learning vector of size 5

B. FPGA Implementation

As a proof of concept, all three designs are implemented and tested on a PYNQ-Z2 board, which integrates a Dual ARM Cortex-A9 processor with 85K of programmable logic cells and 630 KB of block RAM. The board is supported by a Jupyter-based framework and Python APIs. These APIs provide access to the low-level control of the hardware on PYNQ. They also allow the overlay, which configures the architecture of the FPGA, to be loaded through a Python interface. Xilinx's Vivado is used in the development of the digital design and the generation of bitstream. A pattern, in the form of numerical digits, is chosen to showcase the associative memory capability of the ONN. A 5x3 ONN is set up on the FPGA and trained to learn digit 0, 1, and 2. Then a fuzzy digit is presented to the ONN during inference. A set of fuzzy digits is divided into 3 groups. Each group corresponds to each learning digit. In this setup, a learning pattern or digit consists of 15 black-and-white pixels represented in bipolar values. On the other hand, a fuzzy digit can contain a fractional value between -1 and 1 to indicate a grayscale pixel. This grayscale pixel represents a corrupted pixel in this application. Hamming distance (HD) is used as the metric to determine the fuzziness of a digit. The HD value determines how many pixels deviate from its corresponding learning digit. Figure 5 illustrates the fuzzy digits with different HDs.

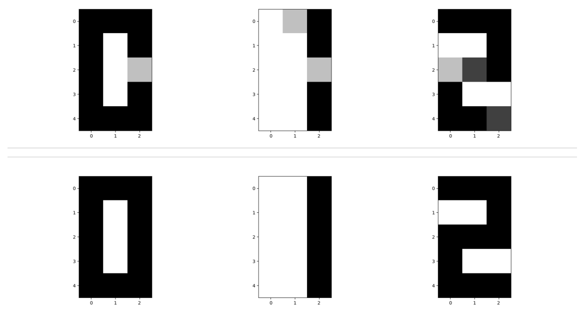


Fig. 5: Fuzzy digits 0,1,2 with HD = 1,2,3 respectively, and their corresponding learning digit.

The application software loads the overlay and controls the communication between PS and PL. The communication between PS and PL depends on the design. For Hebbian learning in PS, the software sends the computed weight matrix and the fuzzy digits to PL. For the other designs, the software sends the learning and fuzzy digits to PL. In all three designs, the software carries the retrieval of the inference result from PL. The user-defined learning and fuzzy digits are written in the software code. This software is mainly used for testing and for comparing the designs. In the demo, which is later described in section IV, a user interface feature is added to the software, allowing users to define the learning and fuzzy digits in real time. The Scheduler handles every data transfer to and from PL. State machines in the Scheduler handles the data transfer with the AXI, Hebb, and ONN blocks. The clock frequency for these blocks can be configured by modifying the clock divider in the Scheduler. To compare the learning and inference execution time between designs, a clock cycle counter is included in the Scheduler. This counter counts the number of clock cycles taken for performing the learning and inference.

IV. Results

A. Design comparison and test

Several design metrics are used to compare these designs. They consist of error rate, utilization, execution time and maximum clock frequency. The three designs are implemented on the PYNQ board for tests and measurements. How the design metrics are measured and the results are discussed in this chapter.

1) Error rate: The accuracy of all three implementations can be determined by their error rates. The error rate can help identify the correlation between the number of learning digits and the ability of the ONN to recognize fuzzy digits. The error rate is calculated as the number of errors divided by the number of fuzzy digits. An

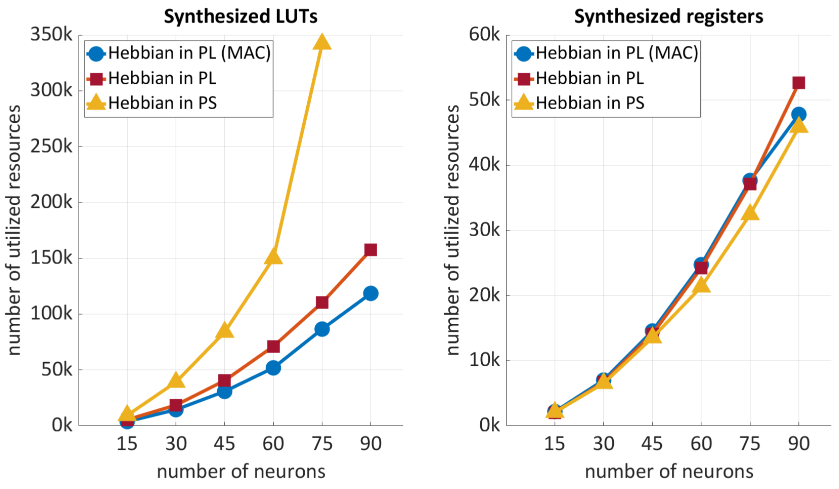


Fig. 6: Resource utilization of the 3 implementations

error is defined as when a fuzzy digit is incorrectly recognized by the ONN. The test results of the three implementations show that they produce the same error rate across different combinations of learning digits and HDs. The test was performed on a 5x3 (15 neurons) ONN. Table I shows the error rate of all three on-chip Hebbian learning implementations. The different combinations of 2 and 3 learning digits of digit 0,1,2 were tested. Each digit has 15 randomly generated and distinct fuzzy numbers for each HD. Thus, 45 fuzzy digits are used in total. HD = 1,2,3 are used in this test.

TABLE I: Error rate of all three on-chip Hebbian learning implementations for 5x3 ONN

Learning digits	HD	#Fuzzy digits	Errors	Error rate (%)
0,1	1	30	0	0
0,1	2	30	0	0
0,1	3	30	0	0
0,2	1	30	0	0
0,2	2	30	1	3.34
0,2	3	30	4	13.34
1,2	1	30	0	0
1,2	2	30	0	0
1,2	3	30	0	0
0,1,2	1	45	3	6.67
0,1,2	2	45	4	8.89
0,1,2	3	45	11	24.44

The result shows that the error rates of 2 learning digits, with respect to the HD, are lower than 3 learning digits. Thus, the error rate increases with the number of learning digits. It also increases with the HD.

2) Utilization: The size of the FPGA implementation is determined by the number of utilized resources. The resource utilization of the three implementations are compared by observing their number of synthesized LUTs and registers in Vivado. Figure 6 shows the number of utilized resources for LUTs and registers vs the number of neurons in the ONN. For Hebbian learning in PS, only the size of the ONN block is observed. For Hebbian learning in PL and PL (MAC), the total size of the ONN and Hebb block is observed. The Scheduler block is not included in this observation because the size is insignificant compared to other blocks and it grows linearly with the number of neurons. The number of neurons is increased to see how the number of utilized resources grows. The result shows that the number of synthesized LUTs for Hebbian learning in PL and PL (MAC) has a polynomial growth. This is because as the number of neurons increases, the weight matrix increases by a factor of n^2. The discrepancy in the number of synthesized LUTs between Hebbian in PL and Hebbian in PL (MAC) is because the weight matrix computation in Hebbian in PL (MAC) is pipelined. It removes the need to finish the multiplication before performing the accumulation. Thus, reducing the number of LUTs. The number of synthesized LUTs for Hebbian in PS grows almost exponentially. This is because the processing of the weight matrix increases by a factor of n^2 in the ONN block. The processing requires a large portion of the LUTs because it needs to sequentially parse the incoming weight matrix data. The numbers of synthesized registers for all 3 implementations show relatively the same growth and no significant discrepancies. The maximum number of neurons that can be implemented on the PYNQ board for Hebbian in PL (MAC) is 60. For Hebbian in PS and PL is 30. This is limited by the number of available LUTs on PYNQ. It can be concluded that Hebbian in PL (MAC) is a better choice among the other two for an application that requires a large number of neurons.

3) Execution time and maximum clock frequency: Execution time and maximum clock frequency are important metrics, especially for an application requiring a real-time constraint. Therefore, we must determine the execution time for the learning and inference and the maximum implementation frequency. This test was performed on a 5x3 ONN. The average learning period for Hebbian in PS is measured by measuring the average time span (μs) to learn a digit. In Hebbian in PL and PL (MAC), this is performed by using a counter in the Scheduler. This counter counts the number of clock cycles between the moment it is triggered and stopped. The Scheduler triggers the counter when the learning starts and stops it when the learning is done. The average inference period for all three implementations was also measured using a counter. The maximum clock frequency is determined by evaluating the timing analysis in Vivado. Table II shows the execution time and maximum clock frequency of the three designs. Hebbian in PL and PL (MAC) spend almost the same clock cycles for inference. Hebbian in PS is a little bit faster because the ONN block is more optimized. The average learning speed for Hebbian in PL (MAC), independent of their maximum clock frequencies, is 8x faster than Hebbian in PL. This is due to the parallelism in the weight matrix computation. At their maximum frequencies of 65 and 41 MHz, Hebbian in PL and PL (MAC) can perform learning 280x and 1500x faster than Hebbian in PS. In conclusion, Hebbian in PL and PL (MAC) are faster than Hebbian in

TABLE II: Execution time and maximum clock frequency for 5x3 ONN

Design	**Average learning period (clock cycles)**	**Average inference period (clock cycles)**	**Maximum clock frequency (MHz)**
Hebbian learning in PS	700 (*)	157	32.5
Hebbian learning in PL	162	218	65
Hebbian learning in PL (MAC)	19	217	41

(*) in μs instead of clock cycles because the learning period is measured in the application software.

PS because the weight matrix computation is performed in hardware and they have higher maximum clock frequencies. Thus, they are a better choice for an application that requires low latency.

B. Pattern recognition demo

The pattern recognition demo aims to showcase the associative memory capability with a user-defined pattern in real time. This demo enables users to create their own pattern and verify if the ONN can recognize it from a fuzzy representation. The demo is performed using the Jupyter platform, which runs a demo application from the ARM core of the PYNQ board. A 5x3 ONN is used for the demo with the Hebbian in PL (MAC) design. Figure 7 illustrates the complete demo setup with a PYNQ-Z2 board and the user interface. The user interface consists of 3 grid boxes namely the Learning, Input, and Output pattern grid boxes. In the Learning pattern grid box, a user can create the desired pattern. This pattern is comprised of black and white pixels. The Learn button sends the pattern to the ONN for learning. The Reset ONN button resets the weights in the ONN. The Input pattern grid box is used to generate the fuzzy pattern. The fuzzy pattern is comprised of black, white and grayscale pixels. The Send button sends the fuzzy pattern for inference. Finally, the inference result will be displayed in the Output pattern grid box.

V. Discussion

This chapter discusses the advantages and limitations of the different learning method designs of digital ONN on FPGA. All three designs show the same performance in terms of accuracy for 5x3 ONN. The test shows that the accuracy can be improved by limiting the number of learning patterns. Another way to improve the accuracy is to resort to a more advanced learning rule. The results show that Hebbian in PL and PL (MAC) yield a better resource utilization trend of synthesized LUTs than Hebbian in PS. This is due to the handling of weight matrix data in the ONN block of Hebbian in PS which consumes a lot of resources. The number of utilized resources can be minimized by optimizing the data parsing in the ONN block. Hebbian in PL (MAC) is slightly better than Hebbian in PL as it can offer more neurons i.e. larger ONN size. For the current implementation in PYNQ board, the maximum number of neurons Hebbian in PL (MAC) is 60 and Hebbian in PL is 30. For both designs, optimization on the Hebb and ONN block can help improve the number of resource utilization for example by adding more pipelines. Hebbian in PL and PL (MAC) are again better than Hebbian in PS in terms of overall execution time and maximum clock frequency. The bottleneck of the overall execution time of Hebbian PS lies on the Hebb block which is implemented in software. It can be concluded that the MAC unit significantly improves the average learning period in hardware. The average inference period, for Hebbian in PL and PL (MAC), can be improved by optimizing the ONN block to have more parallelism on the processing of the inference data. This test shows that implementing Hebbian learning in hardware gives more advantage in terms of execution time or speed. One of the limitations of the Hebb learning designs is the memory size which determines the maximum number of stored patterns. The memory size is determined by the number of encoding bits of the weight. Each weight is encoded in a 5-bit signed integer for Hebbian in PS and PL, and a 6-bit signed integer for Hebbian in PL (MAC). Since the weight is computed off the bipolar learning vector elements by the Hebbian learning rule, the memory size can be formulated as memory size = $\frac{2^n}{2}$ to calculate the memory size. Variable n represents the number of encoding bits. The term in the denominator is a result of using signed integer. The number of learning patterns is used as the unit for the memory size.

In this case, the Hebbian in PS and PL can only store a maximum of 16 patterns. While the Hebbian in PL (MAC) can store up to 32 patterns. The memory size grows exponentially with the number of encoding bits. Increasing the number of encoding bits will have an impact on the number of utilized resources. Increasing the number of stored patterns above the limit will result to a loss of information due to overflow on the signed integer.

VI. Conclusion

In summary, this paper provides three unique designs for on-chip Hebbian learning in a digital implementation on FPGA. They serve a purpose which is to make an on-chip learning possible. These designs make use of the PS and PL units in an FPGA. The first design incorporates the Hebbian learning to the PS. The second design integrates the Hebbian learning into a hardware block in PL. The last design also integrates the Hebbian learning into a hardware block but it provides more parallelism, as it uses a dedicated MAC unit for the weight matrix computation. Several design metrics are used to compare these designs. In terms of error rate, they show an identical performance. The error rate increases with the number of learning digits and Hamming distance. Hebbian in PL (MAC) utilizes the least LUTs amongst the three. The utilization of Hebbian in PL and PL (MAC) has a polynomial growth as the number of neurons increases. While the utilization of Hebbian in PL grows almost exponentially. It is due to the processing of the incoming weight matrix data in the ONN block. The maximum number of neurons that can be implemented on PYNQ-Z2 board for Hebbian in PL (MAC) is 60, while for Hebbian in PL and PS is 30. Thus, Hebbian in PL (MAC) is a better choice for an application that requires a large number of neurons. Hebbian in PL and PL (MAC) performs learning and inference faster than Hebbian in PS, resulting in a better choice for an application that requires low latency.

References

[1] J. Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[2] L.D. Jackel et al. “A neural network approach to hand-print character recognition”. In: *COMPCON Spring ’91 Digest of Papers*. 1991, pp. 472–475. DOI: 10.1109/CMPCON.1991.128851.

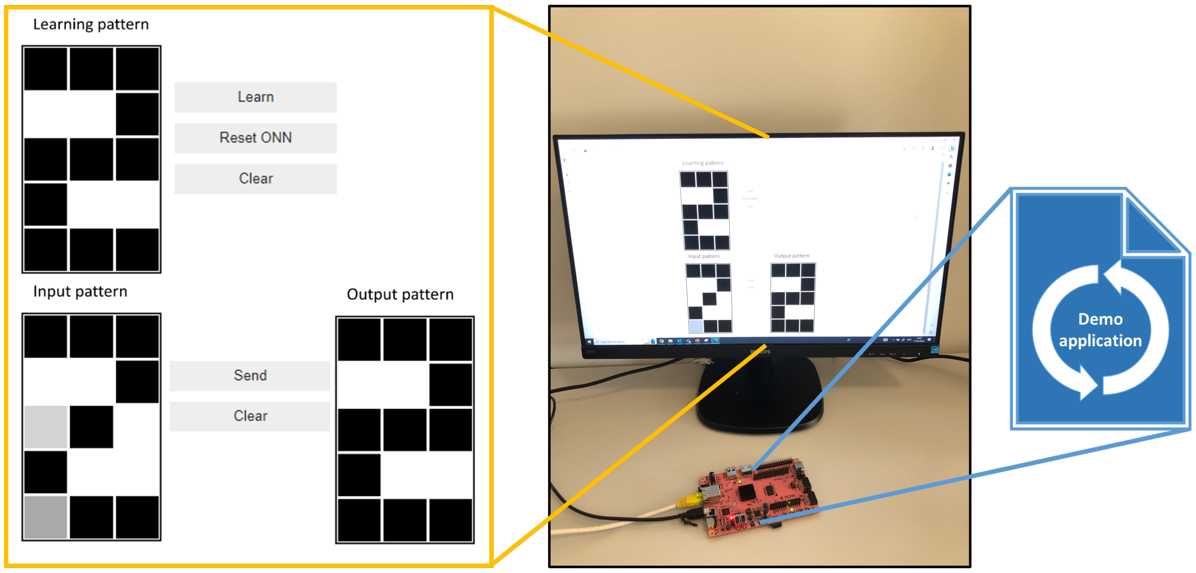


Fig. 7: Pattern recognition demo running on a PYNQ-Z2 board with a user interface.

[3] W. Xiong et al. “The microsoft 2016 conversational speech recognition system”. In: *2017 IEEE ICASSP*. 2017, pp. 5255–5259. DOI: 10.1109/ICASSP.2017.7953159.

[4] A. Raychowdhury et al. “Computing with networks of oscillatory dynamical systems”. In: *Proceedings of the IEEE* 107.1 (2019), pp. 73–89. DOI: 10.1109/jproc.2018.2878854.

[5] C. Delacour et al. “Energy-Performance Assessment of Oscillatory Neural Networks Based on VO $_2$ Devices for Future Edge AI Computing”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2023), pp. 1–14. DOI: 10.1109/TNNLS.2023.3238473.

[6] A. Todri-Sanial et al. “How Frequency Injection Locking Can Train Oscillatory Neural Networks to Compute in Phase”. In: *IEEE Transactions on Neural Networks and Learning Systems* 33.5 (2022), pp. 1996–2009. DOI: 10.1109/TNNLS.2021.3107771.

[7] F.C. Hoppensteadt and E.M. Izhikevich. “Associative memory of weakly connected oscillators”. In: *Proceedings of ICNN’97*. Vol. 2. Houston, TX, USA: IEEE, 1997, pp. 1135–1138. ISBN: 978-0-7803-4122-7. DOI: 10.1109/ICNN.1997.616190.

[8] R. Shi et al. “On the design of phase locked loop oscillatory neural networks: Mitigation of transmission delay effects”. In: *2016 IJCNN*. 2016, pp. 2039–2046. DOI: 10.1109/IJCNN.2016.7727450.

[9] Th. Jackson, S. Pagliarini, and L. Pileggi. “An Oscillatory Neural Network with Programmable Resistive Synapses in 28 Nm CMOS”. In: *2018 IEEE ICRC*. McLean, VA, USA, Nov. 2018, pp. 1–7. DOI: 10.1109/ICRC.2018.8638600.

[10] M. Abernot et al. “Digital implementation of oscillatory neural network for image recognition applications”. In: *Frontiers in Neuroscience* 15 (2021). DOI: 10.3389/fnins.2021.713054.

[11] M. Abernot, Th. Gil, and A. Todri-Sanial. “On-Chip Learning with a 15-neuron Digital Oscillatory Neural Network Implemented on ZYNQ Processor”. In: *Proceedings of the ICONS 2022*. ICONS ’22. New York, NY, USA: Association for Computing Machinery, Sept. 2022, pp. 1–4. DOI: 10.1145/3546790.3546822. URL: https://doi.org/10.1145/3546790.3546822.

[12] R.G.M Morris. “D.O. Hebb: The Organization of Behavior, Wiley: New York; 1949”. In: *Brain Research Bulletin* 50.5-6 (1999), p. 437. DOI: 10.1016/s0361-9230(99)00182-3.

[13] N. Shukla et al. “Ultra low power coupled oscillator arrays for computer vision applications”. In: *2016 IEEE Symposium on VLSI Technology*. June 2016, pp. 1–2. DOI: 10.1109/VLSIT.2016.7573439.

[14] G. Csaba and W. Porod. “Coupled oscillators for computing: A review and perspective”. en. In: *Applied Physics Reviews* 7.1 (Mar. 2020), p. 011302. DOI: 10.1063/1.5120412.

[15] C. Delacour et al. “Oscillatory Neural Networks for Edge AI Computing”. In: July 2021, pp. 326–331. DOI: 10.1109/ISVLSI51109.2021.00066.

[16] C. Delacour and A. Todri-Sanial. “Mapping Hebbian Learning Rules to Coupling Resistances for Oscillatory Neural Networks”. In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X.

[17] J. J. Hopfield. “Neurons with graded response have collective computational properties like those of two-state neurons.” en. In: *Proceedings of the National Academy of Sciences* 81.10 (May 1984), pp. 3088–3092. DOI: 10.1073/pnas.81.10.3088.