Accelerator Framework of Spike-By-Spike Neural Networks for Inference and Incremental Learning in Embedded Systems

Yarib Nevarez, Alberto Garcia-Ortiz
Institute of Electrodynamics and Microelectronics, U. Bremen
{nevarez,agaracia}@item.uni-bremen.de

David Rotermund, Klaus R. Pawelzik
Institute for Theoretical Physics, U. Bremen
{davrot,pawelzik}@neuro.uni-bremen.de

***Abstract*—Although artificial Spiking Neural Networks provide numerous advantages versus the traditional non spiking ones, their high complexity is limiting the use to server computers or dedicated ASIC implementations. As an alternative, the recently proposed Spike-by-Spike (SbS) Neural Networks provide reduced complexity while adding noise-robustness. In this work we propose an accelerator framework for inference and incremental learning targeting resource-constrained devices. The proposed architecture automatically distributes computational tasks to multiple accelerator units. This is the first SbS neural network implementation for embedded systems. Demonstration on a Xilinx Zynq-7020 achieves 99% of accuracy on MNIST dataset classification and a 5x latency enhancement compared to a Core-i7 computer running equivalent network topologies. To facilitate the research in this domain, the entire SbS accelerator framework is available as an open-source project.**

***Index Terms*—Artificial intelligence, spiking neural networks, hardware accelerator, embedded systems, FPGA**

I. Introduction

Over the past decade, the exponential improvement in computing performance and the availability of large amounts of data are boosting the use of Artificial Intelligent (AI) in our daily lives. AI is increasingly attracting the interest of industry and academia; in particular, Artificial Neural Networks (ANNs), an architecture inspired from the biological brain, is becoming the most frequently used form of AI.

Historically, ANNs can be classified into three different generations [1]: The first one is represented by the classical McCulloch and Pitts neuron model; the second one is represented by more complex continuos-output architectures as Multi-Layer Perceptrons and Convolutional Neural Networks (CNN); while the third generation is represented by Spiking Neural Networks (SNNs). They differ fundamentally in the neural computation and the neural coding strategy: while the first generation uses discrete binary values as outputs, the second generation uses continuous activation functions where learning can be easily implemented. In contrast, the third generation, uses spikes as means for information exchange between groups of neurons. This strategy mimics how real neurons interact through short pulses (the so called action potentials).

Although the AI landscape is currently dominated by Deep Neural Networks (DNN) from the second generation, nowadays the SNNs belonging to the third generation are receiving considerable attention [1]–[4] due to their advantages in terms of robustness and the potential to achieve a power efficiency close to that of the human brain (see section II for more details).

Among the family of SNNs, the Spike-by-Spike (SbS) neural network [3] is inspired by the natural computing of the mammalian brain, being a biologically plausible approach although with less complexity than other SNNs. The SbS model differs fundamentally from conventional artificial neural networks since (a) the building block of the network are inference populations (IP) which are an optimized generative representation with non-negative values, (b) time progresses from one spike to the next, preserving the property of stochastically firing neurons, and (c) a network has only a small number of parameters, which is an advantageous stochastic version of Non-Negative Matrix Factorization (NNMF), which is noise-robust and easy to handle. In regard to biological realism and computational effort to simulate neural networks, these properties place the SbS network in between non-spiking NN and stochastically spiking NN [5]. However, despite the favorable noise robustness and reduced complexity, the computational effort imposed by SbS is not suitable for applications in the current emerging technology of the Internet of Things (IoT) and Edge Computing. To enable SbS to perform in such an embedded context, dedicated architectures for SbS acceleration must be deployed, which is the main goal of this work.

In the literature we find plenty of hardware architectures dedicated to the second generation of NN implemented in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) designs [6], [7]. However the related work on SNN is much reduced. Recently, some state of the art survey on hardware architectures for SNN have been reported [1], [4]. In particular, Nassim Abderrahmane et al. briefly describe and compare some recent implementations of ASIC and FPGA where only two are suitable for embedded systems. As a typical example of the current state of the art, Furber et al., presents SpiNNaker [2], aiming to simulate very large SNNs in real-time. It is composed of 48 chips containing a shared memory and 18 ARM cores with small local memory each processor. The main feature of SpiNNaker are the support for several neuron models, synaptic plasticity rules, incremental learning capabilities and efficient communication system. This architecture is suitable for neuroscience research but not for embedded applications. Further on, in a previous

research Rotermund et al., demonstrated the feasibility of a neuromorphic SbS IP in a Xilinx Virtex 6 FPGA [8]. It provides a massively parallel architecture, optimized for memory access and suitable for ASIC implementations. However, this design is considerably resource-demanding to be deployed as a full and functional SbS network in the current embedded technology.

Beside the actual architectures, researches have also identified design methodologies as a critical problem for the efficient development of SNN [1]. For example, Nassim Abderrahmane et al., develop a behavioral level simulator for neuromorphic hardware architectural exploration named NAXT, capable to reduce the number of spikes while keeping the neuron's model resulting in lower power consumption. This work provides a great exploration of SNN for different network topologies and computation approaches on NAXT. However, incremental learning or refinement is missing as features in NAXT and in the embedded implementations summarized in the research.

To address the aforementioned problems, this paper presents a scalable hardware-software framework for SbS NN models targeting embedded systems applications. The proposed framework deploys a fully customizable SbS model allowing arbitrary dimensions, topologies, and acceleration configuration, ideal for rapid experimentation and research on devices with limited resources, particularly in the field of IoT and Edge computing.

On the software side, the architecture offers a comprehensive modern machine learning Application Programming Interface (API), being user-friendly, modular and extensible. On the hardware side, the architecture exploits the available resources of the target platform: from pure embedded software on a single Central Processing Unit (CPU), scaling to a variable number of hardware accelerators in a low-cost or a high-capacity FPGA. This design provides a configurable solution able to match different FPGA characteristics.

The approach is demonstrated using the MNIST dataset classification task deployed in a Xilinx Zynq-7020 achieving a 99% of accuracy, and a 5x latency enhancement compared to a Core-i7 desktop computer running an equivalent network topology in Matlab.

To promote the research on SbS, the entire framework is made available to the public as an open-source project at http://www.ids.uni-bremen.de/sbs-framework.html

II. Spike-by-Spike Neural Networks

As a generative model [3], the SbS model iteratively finds an estimate of its input probability distribution $p(s)$ (i.e. the probability of input node s to stochastically send a spike) by its latent variables via $r(s) = \sum_i h(i)W(s|i)$. An inference population sees only the spikes s_t (i.e. the index identifying the input neuron s which generated that spike at time t) produced by its input neurons, not the underlying input probability distribution $p(s)$ itself. By counting the spikes arriving at a group of SbS neurons, $p(s)$ is estimated by $\hat{p}(s) = 1/T \sum_t \delta_{s,s^t}$ after T spikes have been observed in total. The goal is to generate an internal representation $r(s)$ from the string of incoming spikes s_t such that the negative logarithm of the likelihood $L = C - \sum_\mu \sum_s \hat{p}_\mu(s) log\left(r_\mu(s)\right)$ is minimized. C is a constant which is independent of the internal representation $r_\mu(s)$ and μ denotes one input pattern from an ensemble of input patterns. Applying a multiplicative gradient descent method on L, an algorithm for iteratively updating $h_\mu(i)$ with every observed input spike s_t could be derived

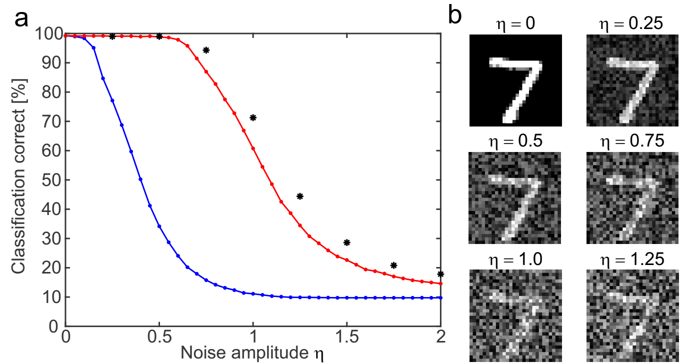

Figure 1. (a) Performance classification of SbS NN versus equivalent CNN, and (b) Example of the first pattern in the MNIST test data set with different amounts of noise.

$$h_\mu^{new}(i) = \frac{1}{1+\epsilon}\left(h_\mu(i) + \epsilon\frac{h_\mu(i)W(s_t|i)}{\sum_j h_\mu(j)W(s_t|j)}\right) \tag{1}$$

where ϵ is a parameter that controls the strength of sparseness of the distribution of latent variables $h_\mu(i)$. Furthermore, L can also be used to derive online and batch learning rules for optimizing the weights $W(s|i)$.

Fundamentally, SbS is a stochastic gradient descent dynamics consistent with Non-Negative Matrix Factorization (NNMF) having several advantages. The stochasticity of gradient descent could in principle overcome local minima. Furthermore, it favors sparse solutions with little fluctuations (which is the case for overcomplete representations). Finally this specific mechanism for inducing sparseness selects those sparse solutions that are robust against noise in the inputs.

In SbS, the expected change at a given h-state (i.e. $\Delta h_i^{s_t} \propto \left\langle \frac{p(s_t|i)h_i}{\sum_j p(s_t|j)h_j} - 1 \right\rangle_{p(s_t)}$ for all $i \in (1, ..., N)$ is exactly the same we would have in a low pass version of NNMF ($\Delta h_i = \sum_s \frac{p(s)p(s|i)h_i}{\sum_j p(s|j)h_j} - 1$). Then, for each given h-state h, the changes of h induced by SbS consist of the expected vector Δh plus fluctuations $\eta_i(s_t)$ with $< \eta_i(s_t) > p(s_t) = 0$ (i.e. $\Delta h_i^{s_t} = \sum_s \frac{p(s)p(s|i)h_i}{\sum_j p(s|j)h_j} + \eta_i(s_t)$). Thus, SbS performs a random walk with mean Δh and some variance and we have a stochastic process in h-space with the correct drift (Δh) and diffusion. Such processes drift towards states where the drift vanishes except for remaining fluctuations. Thus, it produces a Brownian motion finally leading to a probability density for h-states centered around the fixed point.

An example of the robustness of SbS is presented in **Fig.** 1. It compares the classification performance of a SbS network and a tensor flow network, with the same amount of neurons per layer as well as the same layer structure. We trained on MNIST

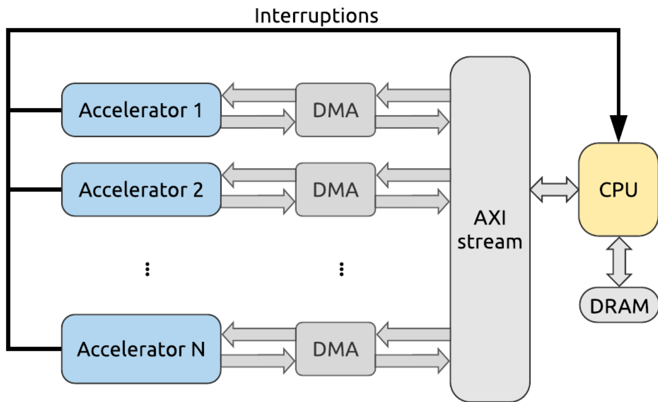

Figure 2. System-level overview of the proposed hardware architecture.

training data without noise (see [5] for details). It shows the correctness for the MNIST test data set with its 10000 patterns in dependency of the noise level for positive additive uniformly distributed noise. The blue curve shows the performance for the tensor flow network, while the red curve shows the performance for the SbS network with 1200 spikes per inference population. Beginning with a noise level of 0.1, the respective performances are different with a p - level of at least 10^{-6} (tested with the Fisher exact test). Increasing the number of spikes per SbS population to 6000 (performance values shown as black stars), shows that more spike can improve the performance under noise even more.

III. The proposed architecture

In this section, we present a hardware-software solution for rapid prototyping and deployment of SbS NN with customizable hardware acceleration in embedded systems. In principle, the proposed architecture is a cross-platform software library for SbS model simulation, capable of utilizing the available hardware resources for numerical calculations on either linear algebra, parallel neuromorphic, or mixed computation. The proposed framework allows modular software customization to employ dedicated Graphics Processing Unit (GPU), Intel's Math Kernel Library (MKL), or any available customized hardware. In this work, we focus on a customizable FPGA platform.

A. Hardware

As a hardware and software design, this architecture is composed simply by one CPU and several hardware accelerators (see **Fig.** 2). Each accelerator is connected through an AXI lite interface for parameter configuration, and AXI stream interfaces for data transfer via Direct Memory Access (DMA) allowing data movement with high transfer rate. Upon conclusion of computation, each hardware accelerator activates an interrupt flag that is processed by the software handlers to collect results and start new transfer.

a) The accelerator unit: As a hardware peripheral, an accelerator performs the update dynamics of an IP and the spike calculation (see **Eq.** (1)). The dynamic equations are written in SystemC. Despite the cost of hardware utilization and computation latency, for this prototype, the SbS model is demonstrated using 32-bit floating point numbers. In the target device (Xilinx Zynq-7020), this implementation requires 12 BRAM-18K, 8 DSP48E, $4,171$ Flip-Flops, and $4,874$ LUTs. The configuration registers of each accelerator are mapped on memory to read and write the parameters associated to computation of the SbS dynamics: number of vectors (IPs and weights), vector length, filter size, and epsilon. Similarly, the memory mapping contains the hardware control register flags: initialize, start, done, idle, ready and interrupt enable. As a special case, it is also implemented a smaller accelerator to merely compute spikes, since the input layer is the one producing the highest amount of spikes and it does not require update dynamic on the IP neuron values.

b) Data transfer: The CPU and the accelerators share an external DRAM as the main memory. Based on the software execution, the CPU writes the data on the memory as a sequenced frame containing firstly a 32-bit random number (MT19937), secondly the IP vector, and finally a series of weight vectors. Then the CPU configures the parameters associated with the particular SbS model on the hardware accelerators and then triggers its DMA for data transfer. Any number of IP frames can be allocated, written, and then transferred at once. The DMA puts the accelerator results back into the main memory.

c) Scalability: The number of hardware accelerators is imposed by the FPGA capacity. An entire SbS-network may be processed either by one single accelerator in a time multiplexed manner, or one single SbS-layers can be partitioned to be processed by several hardware accelerators. Any arbitrary number of accelerators can perform in pipeline.

B. Software

The software architecture is structured as a layered object-oriented application framework. For cross-platform portability this framework is written in C language. Conceptually this design is modular, reusable, and extensible. The overall structure is depicted in **Fig.** 3.

a) Presentation layer: On the top software-layer, the application framework offers a simplified API. This provides the data types, methods, and constructors to build SbS models with customized network depth, layer types, dimensions, learning rules, statistics, and acceleration distribution or scheme.

b) Modular base classes: This layer encapsulates the SbS model classes and their algorithms. Here are defined the function virtual-tables for polymorphic SbS-layers: input, convolution, pooling, fully-connected, and output layer, as well as the multivector class which is used as a tensor. This software-layer also contains utility classes for statistics, data logging, and communication.

c) Accelerator layer: This layer is the interface for the hardware acceleration and peripherals. This prepares the input data for calculation and captures the output results, this involves hardware profiles and configurations.

d) Hardware abstraction layer: This is a platform-dependent layer that provides an abstraction for the hardware

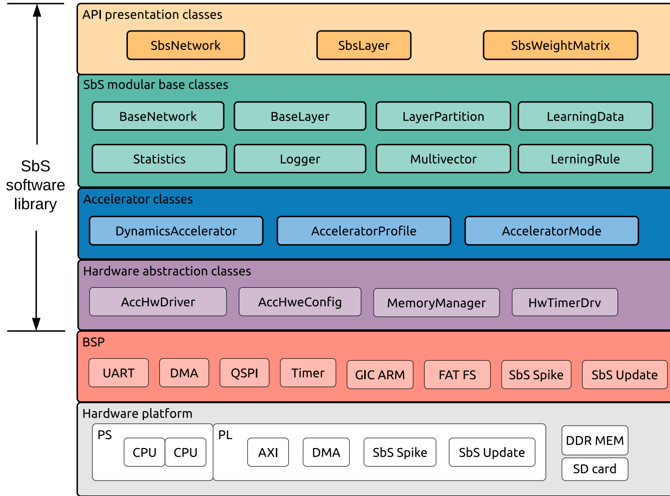

Figure 3. System-level overview of the proposed software architecture.

function calls, this encapsulates hardware initialization, interruption handling, and memory management. This software layer uses proprietary data types and functions from the board support package (BSP) in case of an embedded system, or would use the system calls in case of an embedded Linux environment. This layer uses platform timers for software characterization, statistics, and provides support for upper layers.

C. Software flexibility

The software API is implemented with factory and container design patterns to allow the construction of SbS-layers and group them into a sequential SbS-network in a hierarchical design. It is possible to build any quantity of SbS-network models, however only one can perform computation on the hardware at a time. Each SbS-layer encapsulates IPs, weights, and learning data. These data structures are dynamically created by the framework at runtime and remain static during performance.

D. Accelerator framework and latency model

The accelerator framework can predict an overall performance latency based on the network model, hardware resources and their distribution. We have constructed a latency model of the accelerators using the timing characterization obtained from the high level synthesis.

The SbS update dynamic equation implemented in the hardware accelerators is given by:

$$D_{update} = L(16V + K(11V + 70) + 7) + 42 \quad (2)$$

while the spike hardware accelerator latency is:

$$D_{spk} = L(15V + 9) + 2 \quad (3)$$

The parameters are V, the IP vector length; L, the layer dimension given by $columns \times rows$; and finally K, the kernel or filter size. In this equation, each product represents a hardware-loop. The constant coefficients of V represent the computation latency of each IP element, while the single coefficients combine the initialization and computation of the composed hardware-loops.

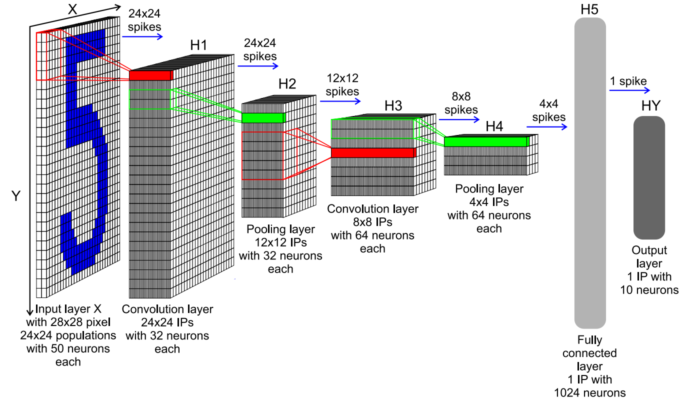

Figure 4. Overview of the implemented application.

As shown later in the experimental results, this formula provides an adequate latency model for the purpose of design exploration, however this model does not include the latency for DMA data movement.

IV. Experimental results

While this hardware architecture is designed to be portable across multiple embedded platforms, this architecture is demonstrated on a Xilinx Zynq-7020, which is equipped with a double ARM Cortex-A9 as a processing system (PS) and programmable logic (PL) equivalent to Xilinx Artix-7.

To demonstrate the advantages of the proposed architecture and methodology, we have implemented a six layered SbS model for MNIST classification task. The SbS NN model is shown in the **Fig.** 4.

For the sake of concreteness, the preliminary training has been performed in Matlab and the resulting weights are exported into binary files and stored in a micro SD card to be loaded into the embedded system. In a small bare metal software application, the SbS model is constructed using the API from the software framework and then deployed in a Xilinx Zynq-7020 at 250Mhz on the PL.

In a first experiment we test the proper functionality and latency of the application as a function of the number of accelerator units and hardware topology. The software API offers the possibility to assign the hardware accelerators over the layers of the SbS model with a simple configuration. We use the **tab.** I to assign different accelerator schemes defined on the table rows. The table shows the number of accelerators assigned to each layer; the fractional numbers indicate a shared accelerator, and the same color indicates the same accelerator.

The **tab.** I exhibits the hardware-software latency behavior. In the first row we observe that one accelerator is shared to compute six SbS-layers in a time multiplexed manner, in this case we have the longest latency caused by the span in the software handlers. The second last row shows the latency when having one accelerator assigned to each SbS-layer, this is not the shortest latency due the software span on the H3 layer,

Table I
OVERVIEW OF THE DIFFERENT IMPLEMENTED HARDWARE TOPOLOGIES.

Hardware topology							Latency (mS)	
Accelerator units	H1	H2	H3	H4	H5	HY	Model prediction	Experimental measurement
1	1/6	1/6	1/6	1/6	1/6	1/6	11.2079	12.082
2	1/3	1/3	1/3	1/3	1/3	1/3	8.849	9.324
3	1/3	1/2	1	1/3	1/2	1/3	5.2177	5.341
4	1/2	1/2	1	1/2	1/2	1	5.2177	5.247
5	1/2	1/2	2	1/2	1	1/2	4.4717	4.653
6	1	1/2	2	1/2	1	1	4.4717	4.694
6	1	1	1	1	1	1	5.2177	5.247
7	1	1	2	1	1	1	4.4717	4.686

which is a heavy-load layer. On the last row, two accelerator units are assigned to H3, this avoids the previous software span giving a fully pipelined case. The pipeline is achieved when the performance of the accelerator units is overlapped by the operation of the software handlers. This is also achieved when sharing accelerator units over smaller layers as we observe in the scheme with five and six accelerators.

Next, we compared the latency time from the experimental implementation with the predictions from the theoretical model of the sec.III. The results show a good agreement with the latency model. The measurements show a fair prediction with a minimum error of 3.89% on the acceleration scheme with 5 accelerators which is the most resource-efficient configuration, in this case we have a fully pipelined operation. As a maximum error, we find 7.23%, which appears in the scheme that has a single shared accelerator to compute all network layers. In this case we have a time multiplexed operation. Since the latency model does not include the data movement, in the fully pipelined operation we observed a deviation caused mainly by the DMA data transmission to the accelerator, whereas in the time multiplexed operation, the deviation is caused mainly by the data transmission and reception. The results show that the models can be used for a fast design exploration of the optimal topology.

For the final demonstration, the design is instantiated with the maximum possible number of accelerator units according to the FPGA capacity of the Zybo Z7-20 (Xilinx Zynq-7020) 86% LUT utilization and power dissipation of 2.743 W. As a result we have a platform with 8 accelerators, one accelerator assigned for the input layer (to produce spikes) and seven accelerators available to compute the update dynamics. This configuration reveals the bottleneck in the software handlers which prevents further reduction of latency.

With these measurements for the particular SbS model, we find the most resource-efficient implementation with 5 accelerators, 65% LUT utilization and power dissipation of 2.424 W. This configuration achieves a 5x latency enhancement compared to a Core-i7 computer running equivalent network topology in Matlab. All the configurations produced accurate computations, the accuracy of pattern prediction is exactly the same as in the Matlab model, 99%.

V. CONCLUSIONS

This work presents a cross-platform accelerator framework for fast prototyping and testing of fully functional SbS network models in embedded systems. As a software-hardware solution, this framework offers a comprehensive high level software API that allows the construction of SbS network models with custom dimensions and configurable hardware acceleration. Further on, this accelerator framework allows to estimate a latency based on the given configuration. The accuracy is sufficient for high-level design exploration.

To show the advantages of the accelerator framework, it was implemented as a test with MNIST classification demonstrating a 5x latency enhancement compared to Matlab on a Core-i7 computer.

As future improvements, an optimization model is under development to automatically predict the optimal accelerator scheme. In addition, the software handler is identified as the throughput bottleneck, hardware handlers and custom data movers will be part of future work to overcome this limitation.

In this paper we have shown that with the proposed accelerator framework it is possible to effectively deploy NN of the third generation in small embedded systems.

VI. ACKNOWLEDGMENTS

This work is funded by the *Consejo Nacional de Ciencia y Tecnología – CONACYT* (the Mexican National Council for Science and Technology).

REFERENCES

[1] N. Abderrahmane, E. Lemaire, and B. Miramond, “Design space exploration of hardware spiking neurons for embedded artificial intelligence,” *Neural Networks*, vol. 121, pp. 366 – 386, 2020.

[2] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, “Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation,” *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, Aug 2013.

[3] U. Ernst, D. Rotermund, and K. Pawelzik, “Efficient computation based on stochastic spikes,” *Neural computation*, vol. 19, no. 5, pp. 1313–1343, 2007.

[4] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, “Spiking neural networks hardware implementations and challenges: A survey,” *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3304103

[5] D. Rotermund and K. R. Pawelzik, “Back-propagation learning in deep spike-by-spike networks,” *Frontiers in Computational Neuroscience*, vol. 13, p. 55, 2019.

[6] D. Wang, K. Xu, J. Guo, and S. Ghiasi, “DSP-efficient hardware acceleration of convolutional neural network inference on FPGAs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[7] M. Davies and et. al., “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, vol. 38, no. 1, pp. 82–99, January 2018.

[8] D. Rotermund and K. R. Pawelzik, “Massively parallel FPGA hardware for spike-by-spike networks,” *bioRxiv*, 2019.