.

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Accelerating Spike-by-Spike Neural Networks with Approximate Dot-Product on FPGA

YARIB NEVAREZ[1], DAVID ROTERMUND[2], KLAUS R. PAWELZIK[3], ALBERTO GARCIA-ORTIZ[4] (Member, IEEE),

[1]Institute of Electrodynamics and Microelectronics, University of Bremen, Bremen 28359, Germany (e-mail: nevarez@item.uni-bremen.de)
[2]Institute for Theoretical Physics, University of Bremen, Bremen 28359, Germany (e-mail: davrot@@neuro.uni-bremen.de)
[3]Institute for Theoretical Physics, University of Bremen, Bremen 28359, Germany (e-mail: pawelzik@@neuro.uni-bremen.de)
[4]Institute of Electrodynamics and Microelectronics, University of Bremen, Bremen 28359, Germany (e-mail: agaracia@item.uni-bremen.de)

Corresponding author: Yarib Nevarez (e-mail: nevarez@item.uni-bremen.de).

This work is funded by the Consejo Nacional de Ciencia y Tecnologia - CONACYT (the Mexican National Council for Science and Technology)

ABSTRACT The Spike-by-Spike (SbS) neural network algorithm is a powerful machine learning (ML) technique for image classification with an exceptional noise robustness. However, deep SbS networks represent an elevated memory footprint and computational cost for profitable implementation in embedded applications. As an alternative, based on the intrinsic error-resilience of neural networks, approximate computing is a promising approach for resource-efficient deployment of neural networks in resource-constrained devices. In this paper, we accelerate SbS neural networks with a dot-product hardware unit based on approximate computing with a quality configurable design. This approach reduces computational latency, memory footprint, and power dissipation while preserving accuracy. For output quality monitoring, we introduce a noise tolerance plot to visualize the impact of approximation on neural network accuracy. To demonstrate our approach, we address a design exploration flow using high-level synthesis and a Xilinx FPGA. As a result, the proposed design achieves up to $20.49\times$ latency enhancement, $8\times$ synaptic memory footprint reduction, and less than 0.5% of accuracy degradation on handwritten digit recognition task.

INDEX TERMS Artificial intelligence, spiking neural networks, approximate computing, logarithmic, parameterisable floating-point, optimization, hardware accelerator, embedded systems, FPGA

I. INTRODUCTION

THE exponential improvement in computing performance and the availability of large amounts of data are boosting the use of Artificial Intelligent (AI) applications in our daily lives. Among the various algorithms developed over the years, neural networks (NNs) have demonstrated remarkable performance in a variety of image, video, audio, and text analytics tasks [1], [2]. Historically, ANNs can be classified into three different generations [3]: the first one is represented by the classical McCulloch and Pitts neuron model using discrete binary values as outputs; the second one is represented by more complex architectures as Multi-Layer Perceptrons and Convolutional Neural Networks (CNN) using continuous activation functions; while the third generation is represented by Spiking Neural Networks (SNNs) using spikes as means for information exchange between groups of neurons. Although the AI field is currently dominated by Deep Neural Networks (DNN) from the second generation, nowadays the SNNs belonging to the third generation are receiving considerable attention [3]–[6] 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 III-A for more details).

Among the family of SNNs, the SbS neural network [5] 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 ANNs 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 [7].

Although SbS networks provide numerous advantages over traditional ANNs and CNNs, deep SbS networks are highly compute and data intensive, representing a challenge for efficient deployment in resource-limited devices. As an emerging SNN algorithm, most SbS models use floating-point numerical representation, which imposes high complexity of the required circuits for the floating-point operations. Model quantization has the potential to improve computational performance; however, this solution is often accompanied by quantization-aware training methods that, in some cases, are problematic or even inaccessible, particularly in deep SNN algorithms [8].

As an alternative, based on the relaxed need for fully precise or deterministic computation of neural networks, approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation. Some research papers have shown the feasibility of applying approximate computing to the inference stage of neural networks [9]–[12]. Such techniques usually demonstrated small inference accuracy degradation, but significant enhancement in computational performance, resource utilization, and energy consumption. Hence, by taking advantage of the intrinsic error-tolerance of neural networks, approximate computing is positioned as a promising approach for inference on resource-limited devices.

In this paper, we accelerate SbS neural networks with a dot-product hardware design based on approximate computing, this approach leverages the intrinsic error-tolerance of neural networks. For quality configurability, we parameterized the mantissa bit-width of the floating-point synaptic-weight vector. As a design parameter, the mantissa bit-width provides a tunable knob to trade-off for resource utilization versus quality of result (QoR) [13], [14], influencing the overall accuracy of the neural network. Since the lower-order bits have smaller significance than the higher-order bits, approximating them may have only a minor impact on QoR [15], [16]. Further on, we can completely remove the mantissa bits in order to use only the exponent of a floating-point representation. Therefore, the worst-case quality configuration becomes a logarithmic representation, which consequently leads to meaningful architectural-level optimizations using only adders and shifters for dot-product approximation in hardware. Moreover, since approximations and noise have qualitatively the same effect [17], we introduce a noise tolerance plot to visualize the impact of the approximation technique on the neural network accuracy.

Our main contributions are as follows:

- We develop a hardware component for dot-product approximation. To perform the sum of pairwise products of two vectors, this hardware module has the following three design features: (1) the pairwise product is approximated by adding integer exponents, and the sum of products is done by accumulating denormalized integer products, which increases computational throughput; (2) the synaptic weight vector uses either reduced custom floating-point or logarithmic representation, which reduces memory footprint; and (3) the neuron vector uses either standard or custom floating-point representation, which preserves QoR and overall inference accuracy.
- We address a design exploration with the proposed dot-product approximation using synaptic weight vector with 1-bit mantissa as well as completely neglected. We evaluate computational latency, accuracy degradation, resource utilization and power dissipation. Experimental results demonstrate $20.49\times$ latency enhancement versus CPU, and less than 0.5% of accuracy degradation on MNIST classification task.
- We present a noise tolerance plot as a quality monitor to visualize the impact of approximation on neural network accuracy.
- Our proposed design for dot-product approximation is adaptable as a building block for other error-resilient applications.

The rest of the paper is organized as follows. Section II covers the related work; Section III introduces the background to SbS networks; Section IV describes the system design and the approximate dot-product hardware module; Section V presents the experimental results thorough a design exploration flow; Section VI concludes the paper.

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

II. RELATED WORK

A. APPROXIMATE COMPUTING IN NEURAL NETWORKS

Approximate computing has been used in a wide range of applications to increase the computational efficiency in hardware [14]. For neural network applications, two main approximation strategies are used, namely network compression and classical approximate computing [18].

1) Network compression

Researchers focusing on embedded applications started lowering the precision of weight and activation map to shrink the memory footprint of the large number of parameters representing ANNs, a method known as network compression or quantization. Taking advantage of the intrinsic error-tolerance of neural networks, as well as their ability to compensate for approximation while training, reduced bit precision causes a small accuracy loss [19]–[22].

In hardware deployment, weight quantization (WQ) has shown up to $2\times$ improvement in energy consumption with an accuracy degradation of less than 1% [23], [24]. Some advanced quantization methods yield to binary neural networks (BNNs) allowing the use of XNORs instead of the conventional costly MACs [22]. In [25], Sun et al. report an accuracy of 98.43% on MNIST with a simple BNN. Hence, quantization is a powerful tool for improving the energy efficiency and memory requirements of ANN accelerators, with limited accuracy degradation.

These methods can be used for SNNs as well. In [26], Rathi et al. report up to $3.1\times$ improvement in energy consumption with an accuracy loss of around 3%. WQ allows the designer to realize a tradeoff between the accuracy of the SNN application against energy and area utilization of the neural networks. Approximate computing can also be applied at the neuron level, where irrelevant units are deactivated to reduce the computation cost of the SNNs [27]. This computation skipping can be applied randomly on synapses, training ANNs with stochastic synapses improves generalization, resulting in a better accuracy [28], [29]. Such method is compatible with SNNs and has been tested both during training [30], [31] and operation [32], and even to define the connectivity between layers [33], [34]. Implementations of spiking neuromorphic systems in FPGA [35] and hardware [36] demonstrated that synaptic stochasticity allows to increase the final accuracy of the networks while reducing memory footprint.

Quantization is therefore a powerful technique to improve energy efficiency and memory requirements of ANN and SNN accelerators, with small accuracy degradation; however, this approach requires quantization-aware training methods that, in some cases, are problematic or even inaccessible, particularly in emerging deep SNN algorithms [8].

2) Classical approximate computing

This approach consists of designing processing elements that approximate their computation by employing modified algorithmic logic units [14]. In [37], Kim et al. have shown SNNs using carry skip adders achieving $2.4\times$ latency enhancement and 43% more energy efficiency, with an accuracy degradation of 0.97% on MNIST classification task. Therefore, approximate computing provides important enhancement in energy efficiency and processing speed.

However, as the complexity of the dataset increases, as well as the depth of the network topology, such as ResNet [38] on ImageNet [39], the accuracy degradation becomes more important and may not be negligible anymore [22], especially for critical applications such as autonomous driving. Therefore, it is not certain that network compression techniques and approximate computing are suitable for any application.

B. SPIKE-BY-SPIKE NEURAL NETWORKS ACCELERATORS

In an earlier research, Rotermund et al. demonstrated the feasibility of a neuromorphic SbS IP on a Xilinx Virtex 6 FPGA [40]. It provides a massively parallel architecture, optimized for memory access and suitable for ASIC implementations. Nonetheless, this design is considerably resource-demanding to be implemented as a complete SbS network in today's embedded technology. In [41], we presented a cross-platform accelerator framework for design exploration and testing of fully functional SbS network models in embedded systems. As a hardware/software (HW/SW) co-design solution, this framework offers a comprehensive high level software API that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. We use this design exploration framework to investigate approximate computing for efficient deployment of deep SbS networks on resource-limited devices.

III. BACKGROUND

A. SPIKE-BY-SPIKE NEURAL NETWORKS

As a generative model [5], 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

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

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 $< \eta_i(s_t) > p(s_t) = 0$

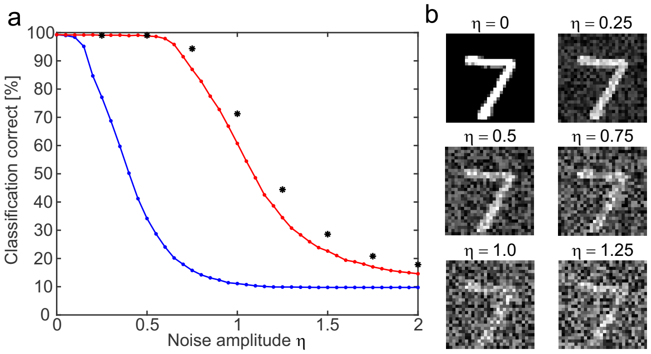

FIGURE 1. (a) Performance classification of SbS NN versus equivalent CNN, and (b) Example of the first pattern in the MNIST test data set with different amounts of noise.

(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 noise tolerance 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 dataset [42] without noise (see [7] for details). It shows the correctness for the MNIST test 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.

The SbS network algorithm is highly suitable for parallelization. SbS network models are constructed in sequential layered structures, each layer consists of many IPs which can be simulated independently while the communication between the IPs is organized by a low bandwidth signal – the spikes [43]. Technically, each IP is an independent computational entity (see **Fig.** 2), this allows to design specialized hardware architectures that can be massively parallelized.

IV. SYSTEM DESIGN

In this section, we introduce the system design of [41], as an accelerator framework of SbS networks for inference and incremental learning in embedded systems. In principle, this architecture is a hardware/software cross-platform for design exploration and deployment of scalable SbS networks on FPGA.

Regarding the software architecture, this is structured as a layered object oriented application framework written in C language. This offers a comprehensive high level software API that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. Conceptually this design is modular, reusable, and extensible. The overall structure is depicted in **Fig.** 3.

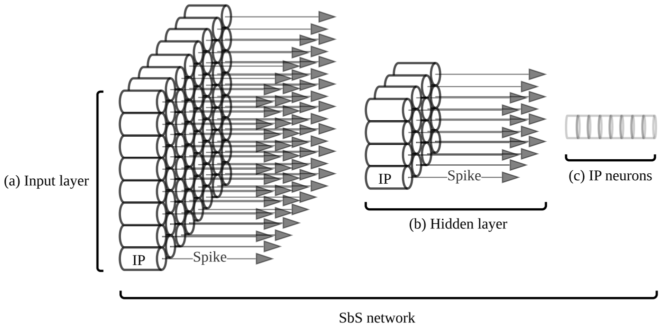

FIGURE 2. SbS IPs as independent computational entities. (a) Illustrates an input layer with a massive amount of IPs operating as independent computational entities. (b) Illustrate a hidden layer with an arbitrary amount of IPs as independent computational entities. (c) Illustrates a set of neurons grouped in an IP.

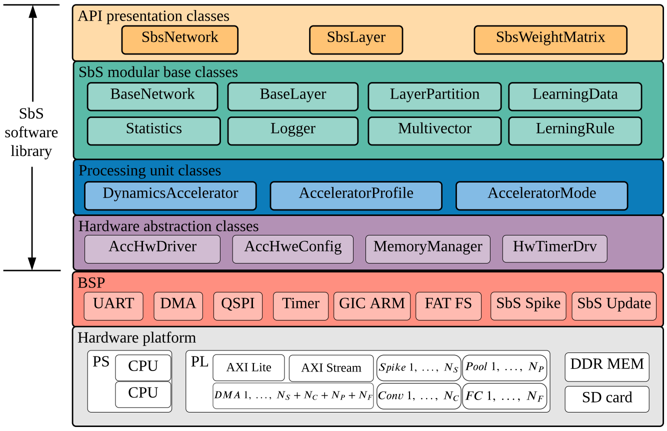

FIGURE 3. System-level overview of the software architecture.

A. HARDWARE ARCHITECTURE

As a hardware/software co-design, the system architecture is a CPU+FPGA-based platform, where the acceleration of SbS network computation is based on asynchronous execution of parallel heterogeneous PUs: *Spike* (input layer), *Conv* (convolution), *Pool* (pooling), and *FC* (fully connected). Each PU is connected through an AXI-Lite interface for the operational mode configuration, and AXI-Stream interfaces for data transfer via Direct Memory Access (DMA) allowing data movement with high transfer rate. Each PU asserts an interrupt flag once the task or transaction is complete, this interrupt event is handled by the CPU to collect results and start a new transaction.

The hardware architecture can resize its resource utilization by changing the number of PUs instances, this provides a good tradeoff between area and throughput (see **Fig.** 4).

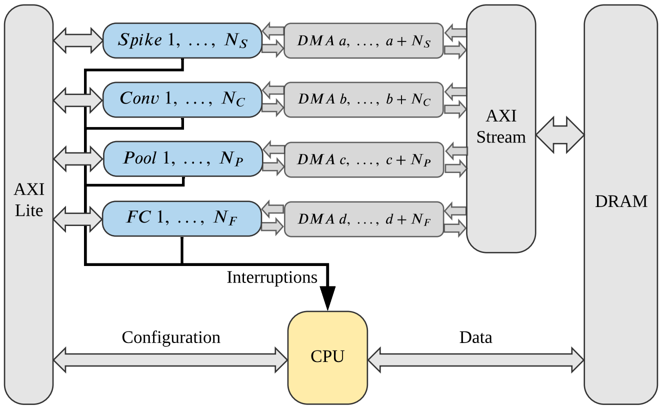

FIGURE 4. System overview of the proposed architecture with scalable number of heterogeneous PUs: *Spike*, *Conv*, *Pool*, and *FC*

The dedicated PUs for *Conv*, and *FC* implement the proposed approximate dot-product as a system component. The PUs are written in C using Vivado HLS (High-Level Synthesis) tool. In this publication, we illustrate the integration of the approximate dot-product component on the *Conv* processing unit.

B. CONV PROCESSING UNIT

This hardware module computes the IP dynamics defined by **Eq.** (1) and offers two modes of operation: *configuration* and *computation*.

1) Configuration mode

In this mode of operation, the PU receives and stores in on-chip memory the parameters to compute the IP dynamics: ϵ as the epsilon, N as the length of $h_\mu \in \mathbb{R}^N$, $K \in \mathbb{N}$ as the size of the convolution kernel, and $H \in \mathbb{N}$ as the number of IPs to process per transaction, this is the number of IPs in a layer or a partition.

Additionally, the processing unit also stores in on-chip memory the synaptic weight matrix using a number representation with a reduced memory footprint. Fundamentally, the synaptic weight matrix is defined by $W \in \mathbb{R}^{K\times K\times M\times N}$ with $0 \leq W(s_t|j) \leq 1$ and $\sum_{j=0}^{N-1} W(s_t|j) = 1$ [7]; hence, W employs only positive normalized real numbers. Therefore, for quality configurability, W is deployed using a reduced floating-point or logarithmic representation as flows:

- Custom floating-point. In this case, W is deployed with a reduced floating-point representation using the necessary bit width for the exponent and for the mantissa according to the given application. For example, 4-bit exponent, 1-bit mantissa; as a result: 5-bit custom floating-point.
- Logarithmic. In this case, the synaptic weight matrix is $W \in \mathbb{N}^{K\times K\times M\times N}$ with positive natural numbers. Since $0 \leq W(s_t|j) \leq 1$ and $\sum_{j=0}^{N-1} W(s_t|j) = 1$, W has only negative values in the logarithmic domain; hence, the sign bit is avoided, and the values are represented in its positive version. Therefore, W is deployed with a representation using the necessary bit width for the exponent according to the given application. For example, 4-bit exponent.

The PU can be reconfigured with different synaptic weight matrix and parameters as needed.

2) Computation mode

In this mode of operation, the PU executes a transaction to process a group of IPs using the previously given parameters and synaptic weight matrix. This process operates in six stages as shown in **Fig.** 5. In the first two stages, the PU receives $h_\mu \in \mathbb{R}^N$, then the PU calculates the firing spike, and stores it in $S^{new} \in \mathbb{N}^H$ (output spike vector). From the third to the fifth stage, the PU receives $S_t \in \mathbb{N}^{K\times K}$ (input spike matrix), then it computes the update dynamics, and then it dispatches $h_\mu^{new} \in \mathbb{R}^N$ (updated IP). The process repeats from the first to the fifth stage for H number of loops (for each IP of the layer or partition). Finally, the S^{new} is dispatched.

The computation of the update dynamics [see **Fig.** 5(d)] operates in two modular stages: *dot-product* and *neuron update*. First, the *dot-product* module calculates the sum of pairwise products of h_μ and $W(s_t)$, while storing each pairwise product as intermediate results. And then, the *neuron update* module calculates **Eq.** (1) reusing previous results and parameters.

The computation of the dot-product is the main piece of **Eq.** (1) and represents a considerable computational cost using standard floating-point in non-quantized network models. Fortunately, the pairwise product of h_μ and $W(s_t)$ is identified as an approximable factor in the dot-product piece of **Eq.** (1). In the following section, we focus on an optimized dot-product hardware design based on approximate computing.

C. DOT-PRODUCT HARDWARE MODULE

This module is part of an application-specific architecture optimized to approximate the dot-product of arbitrary length, see **Eq.** (2). For quality configurability, we parameterized the mantissa bit-width of $W(s_t)$, which provides a tunable knob to trade-off the resource utilization versus QoR. Since the lower-order bits have smaller significance than the higher-order bits, truncating them may have only a minor impact on QoR, we designate this as custom floating-point approximation. Further on, we completely remove the mantissa bits in order to use only the exponent of a floating-point representation. Hence, the worst-case quality configuration becomes a logarithmic representation, consequently leading to advantageous architectural optimizations using only adders and shifters for dot-product approximation in hardware, we designate this as logarithmic approximation.

In this section, we present three pipelined hardware modules using standard floating-point, custom floating-point, and logarithmic approximation.

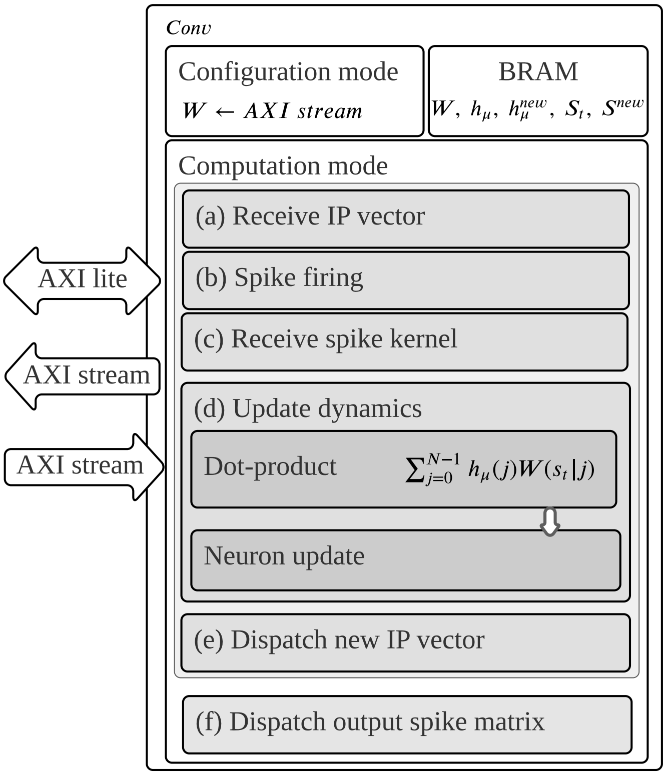

FIGURE 5. The *Conv* processing unit and its six stages: (a) receive IP vector, (b) spike firing, (c) receive spike kernel, (d) update dynamics, (e) dispatch new IP vector, (f) dispatch output spike matrix.

$$r_\mu(s_t) = \sum_{j=0}^{N-1} h_\mu(j)W(s_t|j) \tag{2}$$

1) Dot-product using standard floating-point computation

The hardware module to calculate the dot-product using standard floating-point computation is shown in **Fig.** 6, this diagram exhibits the hardware blocks and their clock cycle schedule. This module loads both $h_\mu(i)$ and $W(s|i)$ from BRAM, then the PU executes the pairwise product [**Fig.** 6(c)] and accumulation [**Fig.** 6(d)]. The intermediate results of $h_\mu(j)W(s_t|j)$ are stored in BRAM for reuse in the neuron update. The latency in clock cycles of this hardware module is defined by **Eq.** (3), where N is the dot-product length. This latency equation is obtained from the general pipelined hardware latency formula: $L = (N-1)\,II + IL$, where II is the initiation interval [**Fig.** 6(a)], and IL is the iteration latency [**Fig.** 6(b)]. Both II and IL are obtained from the high-level synthesis analysis.

$$L_{f32} = 10N + 9 \tag{3}$$

In this design, the high level synthesis tool infers computational blocks with considerable latency cost for standard floating-point. In the case of floating-point multiplication [**Fig.** 6(c)], the synthesis infers a hardware block with a latency cost of 5 clock cycles; theoretically, this block would handle exponents addition, mantissas multiplication, and mantissa correction if needed. Moreover, in the case of floating-point addition [**Fig.** 6(d)], the synthesis infers a hardware block with a latency cost of 9 clock cycles; theoretically, this block would handle mantissas alignment, addition, and correction if needed. Therefore, the use of standard floating-point in high-level synthesis results in a high computational cost that can be enhanced by a custom design.

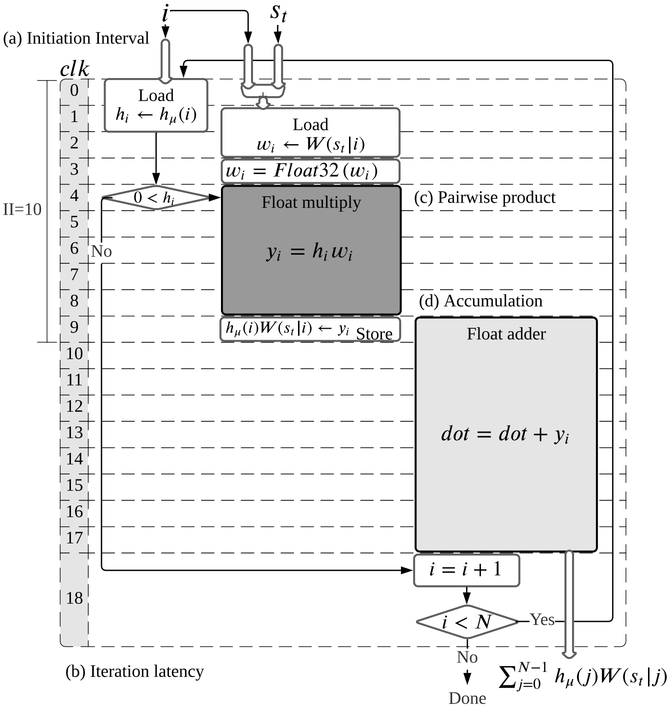

FIGURE 6. Dot-product hardware module using standard floating-point computation. (a) Illustrates the initiation interval of 10 clock cycles. (b) Illustrates the iteration latency of 19 clock cycles. (c) Illustrates the pairwise product block in dark-gray. (d) Illustrates the accumulation block in light-gray.

2) Dot-product using custom floating-point and logarithmic approximation

The hardware module to calculate dot-product using custom floating-point approximation is shown in **Fig.** 7. In this design, h_μ uses standard floating-point number representation, and $W(s)$ uses a positive reduced custom floating-point number representation, where the mantissa bit width is the quality configurability knob, this parameter is tuned by the designer to trade-off between QoR versus resource utilization, thus, energy consumption.

As the wort-case quality configuration, by completely removing the $W(s)$ mantissa leads to a slightly different hardware architecture using only adders and shifters, which computes the dot-product using logarithmic approximation, this is shown in **Fig.** 8.

Additionally, the exponent bit width of $W(s)$ is a design parameter for efficient resource utilization, and it is defined based on the application or deployment needs.

The custom floating-point and logarithmic approximation designs work in three phases: *Computation*, *Threshold-test*, and *Result normalization*.

- Phase I, *Computation*:
  In this phase, it is calculated the magnitude of the dot-product in a denormalized representation. This is done in two iterative steps: *pairwise product* and *accumulation*. Where *pairwise product* is executed either in custom floating-point or logarithmic approximation described below.
  - Pairwise product.
    - Custom floating-point. As shown in **Fig.** 7(c) in dark-gray, the pairwise product is approximated by adding exponents and multiplying mantissas of both $W(s|i)$ and $h_\mu(i)$. If the mantissa multiplication results in an overflow, then it is corrected by increasing the exponent and shifting the resulting mantissa by one position to the right. Then we have $h_\mu(j)W(s_t|j)$ as an intermediate result which is stored for future reuse on the neuron update calculation. In this design the pairwise product has a latency of 5 clock cycles.
    - Logarithmic. As shown in **Fig.** 8(c) in dark-gray, the pairwise product is approximated by adding $W(s|i)$ to the exponent of $h_\mu(i)$, since $W(s)$ values are represented in the logarithmic domain and h_μ in standard floating-point. In this design the pairwise product has a latency of one clock cycle.
  - Accumulation. As shown in both **Fig.** 7(d) and **Fig.** 8(d) in light-gray, first, it is obtained the denormalized representation of $h_\mu(j)W(s_t|j)$ by shifting its mantissa using its exponent as shifting parameter (barrel shifter). And then, this denormalized representation is accumulated to obtain the approximated magnitude of the dot-product.

  The process of pairwise product and accumulation iterates for each element of the vectors, the computation latency is given by **Eq.** (4) for custom floating-point, and **Eq.** (5) for logarithmic, where N is the length of the vectors. Both pipelined hardware modules have the same throughput, since both have two clock cycles as initiation interval.

$$L_{custom} = 2N + 11 \tag{4}$$

$$L_{log} = 2N + 7 \tag{5}$$

- Phase II, *Threshold-test*:
  In this phase, the accumulated denormalized magnitude is tested to be above of a predefined threshold, this must be above zero, since the dot-product is a denominator in **Eq.** (1). If passing the threshold, then the next phase is executed, otherwise the rest of update dynamics is skipped. The threshold-test takes one clock cycle.
- Phase III, *Result-normalization*:
  In this phase, the dot-product is normalized to obtain the exponent and mantissa in order to build a standard floating-point for later use in the neuron update. The normalization is obtained by shifting the approximated dot-product magnitude in a loop until it is in the form of a normalized mantissa where the iteration count represents the exponent of the dot-product, each iteration takes one clock cycle.

The total latency of the hardware module using custom floating-point and logarithmic approximation is the accumulated latency of the three phases.

The proposed architecture using custom floating-point and logarithmic approximation overcomes the performance of the design using standard floating-point. The performance enhancement is achieved by decomposing the floating-point computation into an advantageous handling of exponent and mantissa using intermediate accumulation in a denormalized representation and only one final normalization.

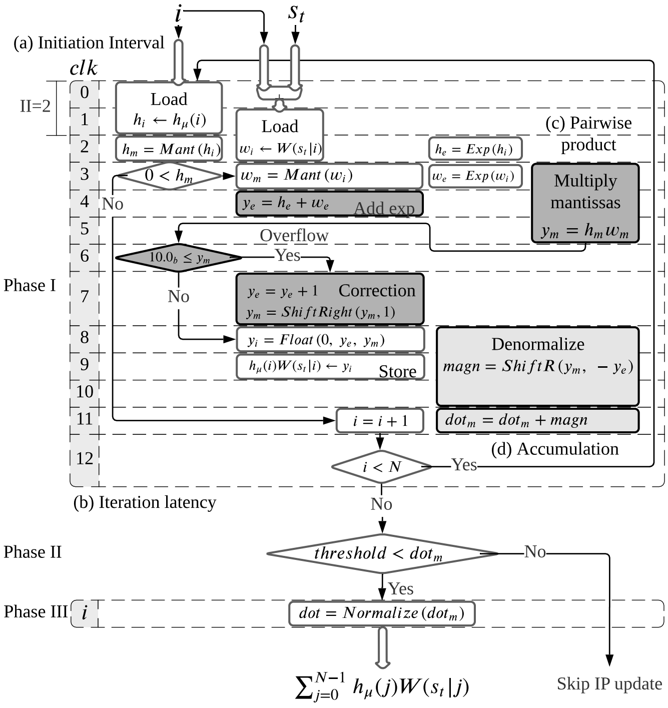

FIGURE 7. Dot-product hardware module using custom floating-point approximation. (a) Illustrates the initiation interval of 2 clock cycles. (b) Illustrates the iteration latency of 13 clock cycles. (c) Illustrates the pairwise product blocks in dark-gray. (d) Illustrates the accumulation blocks in light-gray.

V. EXPERIMENTAL RESULTS

The proposed architecture is demonstrated on a Xilinx Zynq-7020. This device integrates a dual ARM Cortex-A9 based processing system (PS) and programmable logic (PL) equivalent to Xilinx Artix-7 (FPGA) in a single chip [44]. The Zynq-7020 architecture conveniently maps the custom logic

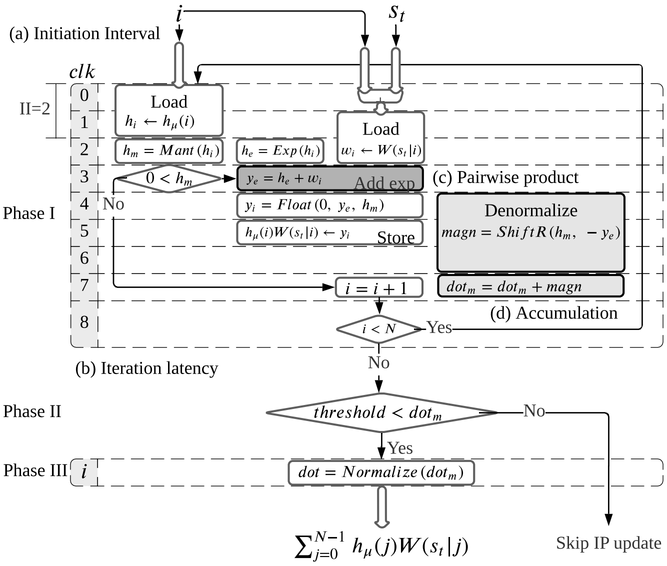

FIGURE 8. Dot-product hardware module using logarithmic approximation. (a) Illustrates the initiation interval of 2 clock cycles. (b) Illustrates the iteration latency of 9 clock cycles. (c) Illustrates the pairwise product block in dark-gray. (d) Illustrates the accumulation blocks in light-gray.

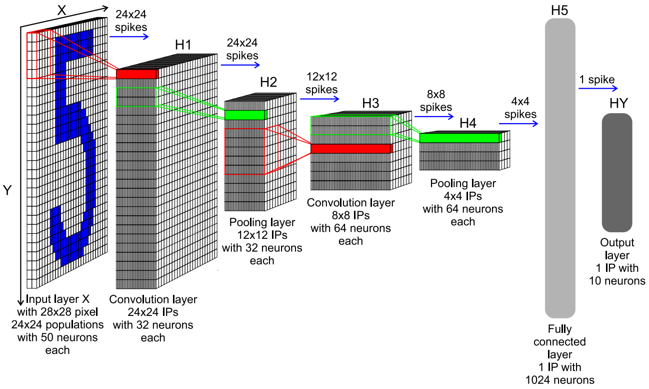
X
Y
24x24 spikes
H1
24x24 spikes
H2
12x12 spikes
H3
8x8 spikes
H4
4x4 spikes
H5
1 spike
HY
Input layer X with 28x28 pixel 24x24 populations with 50 neurons each
Convolution layer 24x24 IPs with 32 neurons each
Pooling layer 12x12 IPs with 32 neurons each
Convolution layer 8x8 IPs with 64 neurons each
Pooling layer 4x4 IPs with 64 neurons each
Fully connected layer 1 IP with 1024 neurons
Output layer 1 IP with 10 neurons

FIGURE 9. SbS network structure for MNIST classification task. Input X: Input layer with 28×28 normalization modules for 28×28 input pixel. From this layer spikes are send to layer *H1*. *H1*: Convolution layer *H1* with 24×24 IPs with 32 neurons each. Every IP processes the spikes from 5×5 spatial patches of the input pattern (x and y stride is 1). *H2*: 2×2 pooling layer *H2* (x and y stride is 2) with 12×12 IPs with 32 neurons each. The weights between *H1* and *H2* are not learned but set to a fixed weight matrix that creates a competition between the 32 features of *H1*. *H3*: 5×5 convolution layer *H3* (x and y stride is 1) with 8×8 IPs. Similar to *H1* but with 64 neuron for each IP. *H4*: 2×2 pooling layer *H4* (x and y stride is 2) with 4×4 IPs with 64 neurons each. This layer is similar to layer *H2*. *H5*: Fully connected layer *H5*. $1,024$ neurons in one big IP which are fully connected to layer *H4* and output layer *HY*. *HY*: Output layer *HY* with 10 neurons for the 10 types of digits. selected.

TABLE 1. Computation on CPU.

Layer	Latency (ms)
HX_IN	1.184
H1_CONV	4.865
H2_POOL	3.656
H3_CONV	20.643
H4_POOL	0.828
H5_FC	3.099
HY_OUT	0.004
TOTAL	34.279

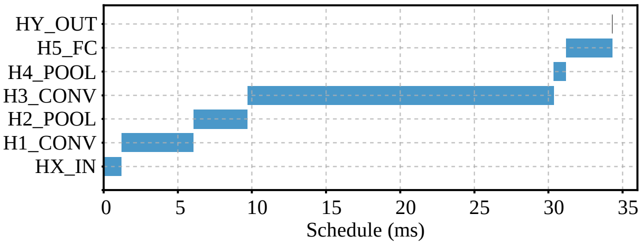

FIGURE 10. Computation on CPU.

and software in the PL and PS respectively as an embedded system.

In this platform, we implement the proposed hardware architecture to deploy the SbS network structure shown in **Fig.** 9 for MNIST classification task. The SbS model is trained in Matlab without any quantization method, using standard floating-point. The resulting synaptic weight matrices are deployed on the embedded system, where the SbS network is built as a sequential model using the API from the SbS embedded software framework [41]. This API enables to configure the computational workload of the neural network, which can be distributed among the hardware processing units and the CPU.

For the evaluation of our approach, we address a design exploration reviewing the computational latency, inference accuracy, resource utilization, and power dissipation. First, we benchmark the performance of SbS network simulation on CPU, and then on hardware processing units using standard floating-point computation. Afterwards, we evaluate our dot-product architecture addressing a design exploration using custom floating-point, and then logarithmic computation. Finally, we present a discussion of the given results.

A. PERFORMANCE BENCHMARK

1) Benchmark on CPU

We examine the performance of the CPU for SbS network simulation with no hardware coprocessing. In this case, the embedded software builds the SbS network as a sequential model mapping the entire computation to the CPU (ARM Cortex-A9) at 666 MHz and a power dissipation of $520mW$.

The SbS network computation on the CPU achieves a latency of $34.279ms$ per spike with an accuracy of 99.3% correct classification on the $10,000$ image test set at 1000 spikes. The latency and schedule of the SbS network computation are displayed in **Tab.** 1 and **Fig.** 10 respectively.

2) Benchmark on processing units using standard floating-point

To benchmark the computation on hardware PUs using standard floating-point, we implement the system architecture shown in **Fig.** 11. In this case, the embedded software builds the SbS network as a sequential model mapping the network computation to the hardware processing units at 200 MHz as clock frequency.

The heaviest layers of the neural network are partitioned

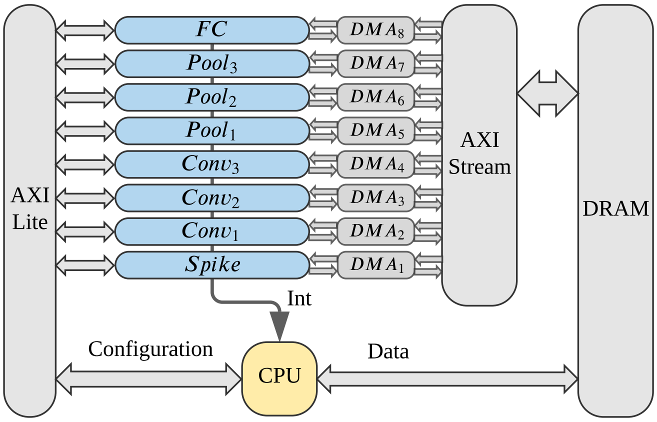

FIGURE 11. System overview of the top-level architecture with 8 processing units.

TABLE 2. Performance of processing units using standard floating-point computation.

Hardware mapping		Computation schedule (ms)			
Layer	PU	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.056	0.370	0.426
H1_CONV	Conv1	0.058	0.598	2.002	2.658
H2_POOL	Pool1	0.658	0.126	1.091	1.875
	Pool2	0.785	0.125	1.075	1.985
H3_CONV	Conv2	0.911	0.280	3.183	4.374
	Conv3	1.193	0.279	3.176	4.648
H4_POOL	Pool3	1.473	0.037	0.481	1.991
H5_FC	FC	1.512	0.101	1.118	2.731
HY_OUT	CPU	1.615	0.004	0	1.619

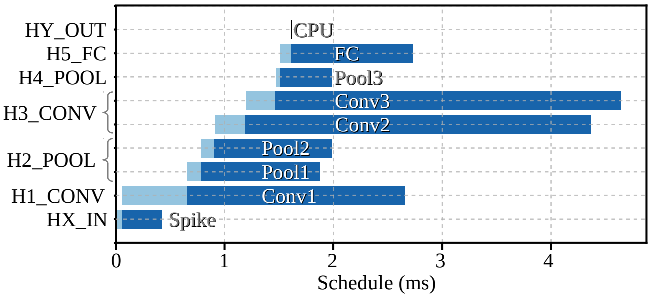

FIGURE 12. Performance of processing units using standard floating-point computation.

for asynchronous parallel processing. Hence, it is distributed the computational workload of *H2_POOL* and *H3_CONV* among two PUs each one, since these are the heaviest pooling and convolution layers respectively. The output layer *HY_OUT* is fully processed by the CPU, since it is the lightest one. The hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 2 and **Fig.** 12.

In the computation schedule, the following terms are defined: $t_s(n)$ as the start time for the processing of the neural network layer (as a computation node) $n \in L$ where L represents the set of layers; $t_{CPU}(n)$ as the CPU preprocessing time; $t_{PU}(n)$ as the PU latency; and $t_f(n)$ as the finish time. For data preparation, the $t_{CPU}(n)$ is the period of time in which the CPU writes a DRAM buffer with h_μ (neuron vector) of the current processing layer and S_t (spike vector) from its preceding layer, this buffer is streamed to the PU via DMA.

The total execution time of the CPU is defined by **Eq.** (6). In a cyclic inference, the execution time of the network computation is the longest path among the processing units including the CPU, this is denoted as the latency of an spike cycle, and it is defined by **Eq.** (8). The total execution time of the network computation is the latest finish time defined by **Eq.** (9).

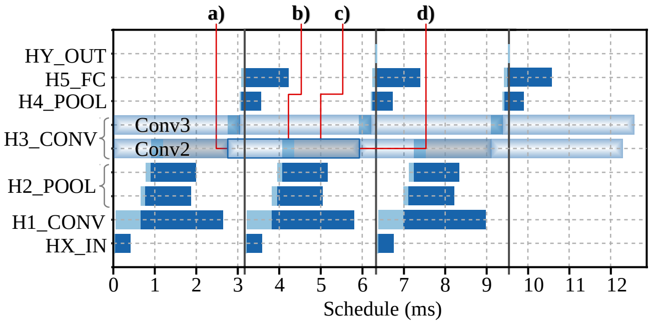

FIGURE 13. Performance bottleneck of cyclic computation on processing units using standard floating-point. a) Illustrates the starting of t_{PU} of *Conv2* on a previous computation cycle. b) Illustrates t_{CPU} of *Conv2* on the current computation cycle. c) Illustrates the CPU waiting time (in gray color) for *Conv2* as a busy resource (awaiting for *Conv2* interruption). d) Illustrates the t_f from the previous computation cycle, and the starting of t_{PU} on the current computation cycle (*Conv2* interruption on completion, and start current computation cycle).

$$T_{CPU} = \sum_{n \in L} t_{CPU}(n) \tag{6}$$

$$T_{PU} = \max_{n \in L}(t_{PU}(n)) \tag{7}$$

$$T_{SC} = \begin{cases} T_{PU}, & \text{if } T_{CPU} \leq T_{PU} \\ T_{CPU}, & \text{otherwise} \end{cases} \tag{8}$$

$$T_f = \max_{n \in L}(t_f(n)) \tag{9}$$

Using standard floating-point represents a high computational cost. As the heaviest layer, the computational workload of *H3_CONV* is evenly partitioned among two PUs: *Conv2* and *Conv3*. However, in the cyclic schedule, *Conv2* causes the performance bottleneck as shown in **Fig.** 13. In this case, the CPU has to await for *Conv2* to finish the computation of the previous cycle in order to start the current computation cycle. Therefore, applying **Eq.** (8), we obtain a latency of 3.183 ms per spike cycle.

This deployment achieves an accuracy of 98.98% correct classification on the $10,000$ image test set at 1000 spikes. Further on, we benchmark the noise tolerance report using input patterns with positive additive equidistributed random noise up to 55% of amplitude as illustrated in **Fig.** 14. The post-implementation resource utilization and power dissipation are shown in **Tab.** 3 and **Tab.** 4, respectively.

Each *Conv* PU instantiates an on-chip stationary weight matrix of $52,000$ entries to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $W \in \mathbb{R}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. In order to reduce BRAM utilization, we use a custom floating-point representation composed of 4-exponent and 4-bit mantissa. Each 8-bit entry is promoted to its standard floating-point representation for the dot-product computation. The methodology to find the appropriate bit width parameters for custom floating-point representation is presented in the next section.

TABLE 3. Resource utilization of processing units using standard floating-point.

PU	LUT	FF	DSP	BRAM 18K
Spike	2,640	4,903	2	2
Conv	2,765	4,366	19	37
Pool	2,273	3,762	5	3
FC	2,649	4,189	8	9

TABLE 4. Power dissipation of processing units using standard floating-point.

PU	Power (mW)
Spike	38
Conv	89
Pool	59
FC	66
CPU	520

B. DESIGN EXPLORATION FOR CUSTOM FLOATING-POINT AND LOGARITHMIC APPROXIMATION

In this section, we address a design exploration to evaluate our approach for SbS neural network simulation using custom floating-point and logarithmic approximation. First, we examine the synaptic weight matrix of each SbS network layer in order to determine the minimum requirements for numeric representation and memory storage. Second, we implement the proposed dot-product architecture using the minimal floating-point and logarithmic representation as design parameters. Finally, we evaluate overall performance, inference accuracy, resource utilization, and power dissipation.

1) Parameters for numeric representation of synaptic weight matrix

We obtain the parameters for numeric representation from the $\log_2$-histograms of the synaptic weight matrix of each layer as shown in **Fig.** 12. Since $0 \le W(s_t|j) \le 1$ and $\sum_{j=0}^{N-1} W(s_t|j) = 1$, the W elements have only negative values in the logarithmic domain; hence, the sign bit is disregarded and the values are stored in its positive version, as stated in Section IV-A. The smallest floating-point entry of W represents the minimum exponent value, as defined by **Eq.** (10), and the bit width needed for its absolute binary representation is defined by **Eq.** (11).

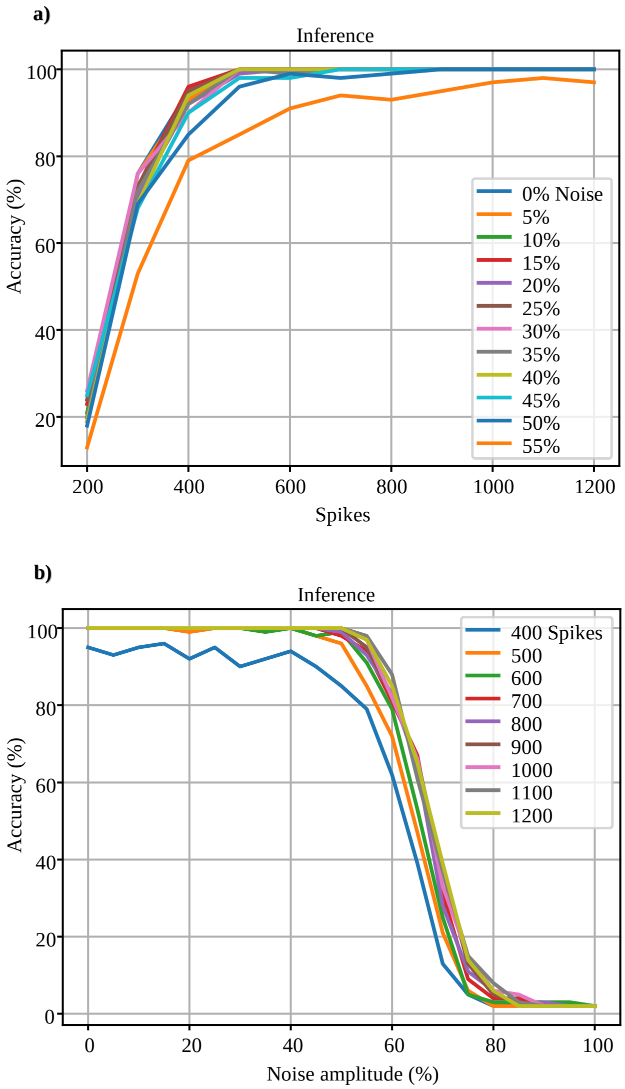

FIGURE 14. Noise tolerance on hardware PU using standard floating-point computation (benchmark). (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

$$E_{\min} = \log_2(\min_{\forall i}(W(i))) \tag{10}$$

$$N_E = \lceil \log_2(|E_{\min}|) \rceil \tag{11}$$

Applying **Eq.** (10) and **Eq.** (11) to the given SbS network, we obtain $E_{\min} = -13$ and $N_E = 4$. Thus, for absolute binary representation of the exponents, it is needed 4-bits.

For quality configurability, the mantissa bit width is a knob parameter that is tuned by the designer. This parameter leverages the builtin error-tolerance of neural networks and performs a trade-off between resource utilization and QoR. In the following subsection, we present a case study with 1-bit mantissa corresponding to the custom floating-point approximation.

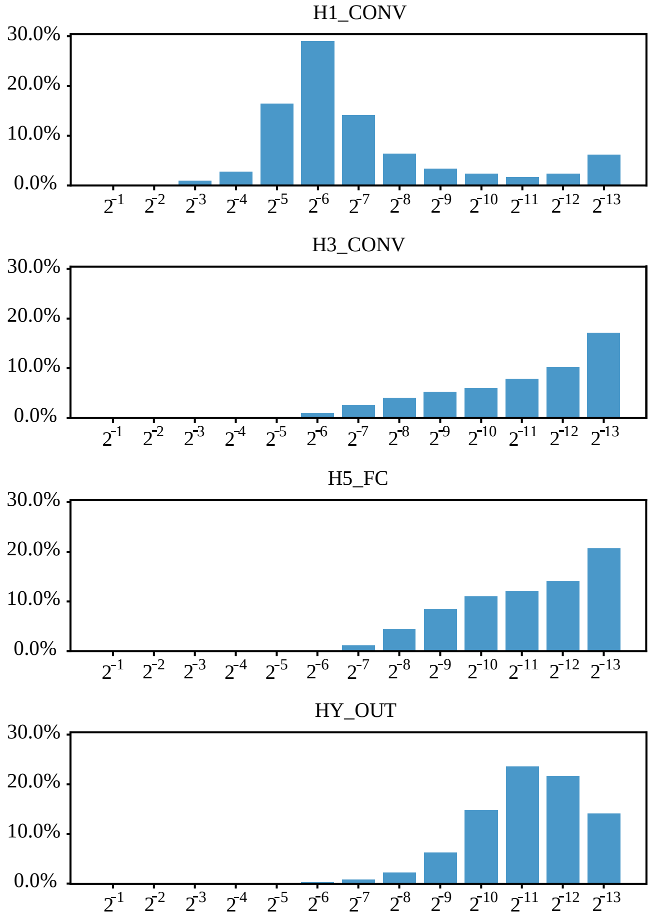

FIGURE 15. $\log_2$-histogram of each synaptic weight matrix showing the percentage of matrix elements with given integer exponent.

2) Design exploration for dot-product using custom floating-point approximation

For this design exploration, we use a custom floating-point representation composed of 4-bit exponent and 1-bit mantissa, this format is used for the synaptic weight vector on the proposed dot-product architecture. In this case, each *Conv* PU instantiates an on-chip stationary weight matrix for $52,000$ entries of 5-bit each one, which is enough to store $W \in \mathbb{R}^{5\times5\times2\times32}$ and $W \in \mathbb{R}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. The same dot-product architecture is implemented in *FC* processing unit, however, due to lack of BRAM resources, this PU does not instantiate on-chip stationary synaptic weight matrix. Instead, *FC* receives neuron and synaptic vectors during performance. The hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 5 and **Fig.** 16.

As shown in the computation schedule in **Tab.** 5 and **Fig.** 16, this implementation achieves a maximum hardware PU latency of $1.309ms$ according to **Eq.** (7), and a CPU latency of $1.673ms$. Therefore, applying **Eq.** (8), we obtain a latency of $1.673ms$ per spike cycle as shown in **Fig.** 16. In this case, the cyclic bottleneck is in the performance of the CPU.

TABLE 5. Performance of hardware processing units using custom floating-point approximation.

Hardware mapping		Computation schedule (ms)			
Layer	**PU**	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.307	0.362
H1_CONV	Conv1	0.057	0.654	1.309	2.020
H2_POOL	Pool1	0.713	0.131	1.098	1.942
	Pool2	0.845	0.125	1.098	2.068
H3_CONV	Conv2	0.972	0.285	1.199	2.456
	Conv3	1.258	0.279	1.184	2.721
H4_POOL	Pool3	1.538	0.037	0.484	2.059
H5_FC	FC	1.577	0.091	0.438	2.106
HY_OUT	CPU	1.669	0.004	0	1.673

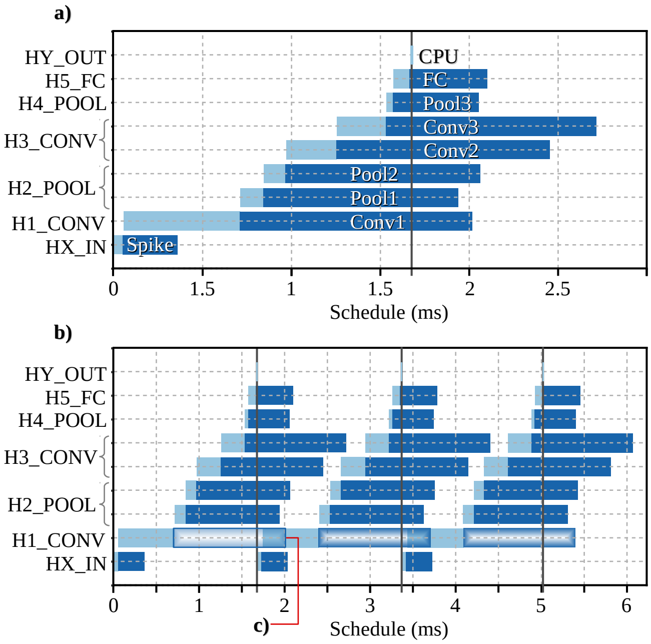

FIGURE 16. Performance on processing units using custom floating-point approximation. a) Illustrates computation schedule. b) Illustrates cyclic computation schedule. c) Illustrates the performance of *Conv2* from a previous computation cycle during the preprocessing of *H1_CONV* on the current computation cycle without bottleneck.

This configuration achieves an accuracy of 98.97% correct classification on the $10,000$ image test set at 1000 spikes, this indicates an accuracy degradation of 0.33%. For output quality monitoring, the noise tolerance is illustrated in **Fig.** 17. Only the band of 55% noise amplitude is decaying; hence, this report demonstrates a minimal impact on the neural network accuracy.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 6 and **Tab.** 7, respectively.

TABLE 6. Resource utilization of processing units using custom floating-point approximation.

PU	LUT	FF	DSP	BRAM 18K
Conv	3,139	4,850	19	25
FC	3,265	5,188	8	9

TABLE 7. Power dissipation of processing units using custom floating-point approximation.

PU	Power (mW)
Conv	82
FC	66

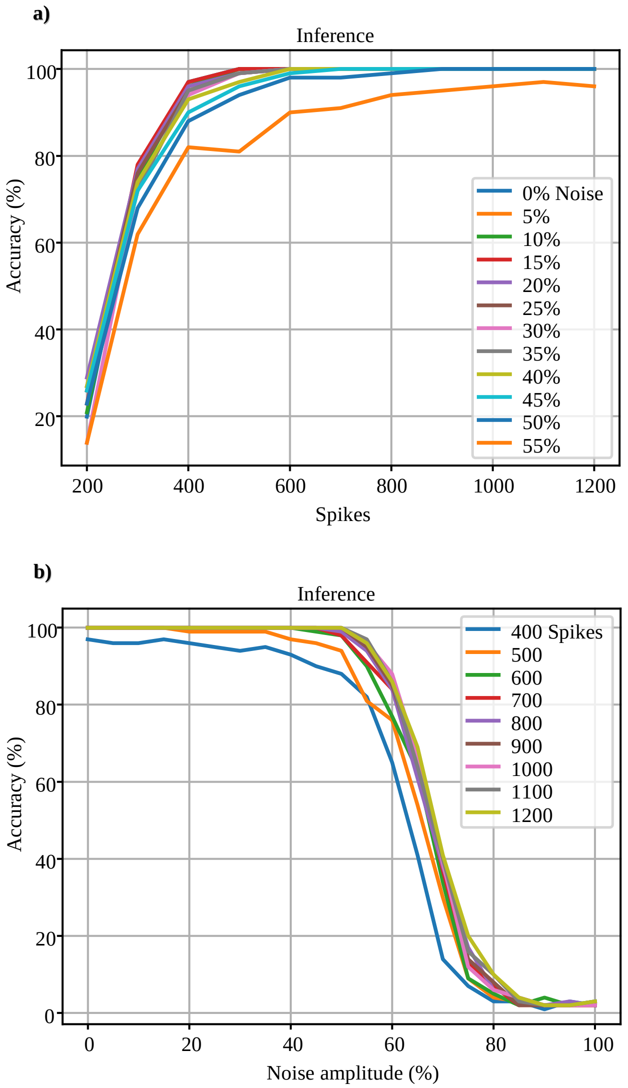

FIGURE 17. Noise tolerance on hardware PU using custom floating-point approximation. (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

TABLE 8. Performance of hardware processing units using logarithmic approximation.

Hardware mapping		Computation schedule (ms)			
Layer	PU	t_s	t_{CPU}	t_{PU}	t_f
HX_IN	Spike	0	0.055	0.264	0.319
H1_CONV	Conv1	0.057	0.655	1.271	1.983
H2_POOL	Pool1	0.714	0.130	1.074	1.918
	Pool2	0.845	0.126	1.106	2.077
H3_CONV	Conv2	0.973	0.285	1.179	2.437
	Conv3	1.258	0.278	1.176	2.712
H4_POOL	Pool3	1.538	0.037	0.488	2.063
H5_FC	FC	1.577	0.091	0.388	2.056
HY_OUT	CPU	1.669	0.004	0	1.673

3) Design exploration for dot-product using logarithmic approximation

As the worst-case quality configuration, we use a 4-bit integer exponent for logarithmic representation of the synaptic weight matrix. In this case, each *Conv* processing unit implements the proposed dot-product architecture and an on-chip stationary weight matrix for $52,000$ entries of 4-bit integer each one to store $W \in \mathbb{N}^{5\times5\times2\times32}$ and $W \in \mathbb{N}^{5\times5\times32\times64}$ for *H1_CONV* and *H3_CONV*, respectively. The same dot-product architecture is implemented in *FC* processing unit without stationary synaptic weight matrix. The hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 8 and **Fig.** 18.

As shown in the computation schedule in **Tab.** 8 and **Fig.** 18, this implementation achieves a maximum hardware PU latency of $1.271ms$ according to **Eq.** (7), and a CPU latency of $1.673ms$. Therefore, applying **Eq.** (8), we obtain a latency of $1.673ms$ per spike cycle as shown in **Fig.** 18. In this case, the cyclic bottleneck is in the CPU performance.

This quality configuration achieves an accuracy of 98.84% correct classification on the $10,000$ image test set at 1000 spikes, this indicates an accuracy degradation of 0.46%. For output quality monitoring, the noise tolerance is illustrated in **Fig.** 19. The band of 45% noise amplitude is decaying; hence, this report demonstrates a minor impact on the neural network accuracy.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 6 and **Tab.** 7, respectively.

TABLE 9. Resource utilization of processing units using logarithmic calculation.

PU	LUT	FF	DSP	BRAM 18K
Conv	3,086	4,804	19	21
FC	3,046	4,873	8	8

TABLE 10. Power dissipation of processing units using logarithmic calculation.

PU	Power (W)
Conv	78
FC	66

C. RESULTS AND DISCUSSION

As a benchmark, the SbS network simulation on CPU using 32-bit standard floating-point achieves an accuracy of 99.3% with a latency of $T_{SC} = 34.279ms$. As a second benchmark, the network simulation on hardware processing units using floating-point (for storage, 4-bit exponent and 4-bit mantissa, and then promoted to standard floating-point for the computation) achieves an accuracy of 98.98% with

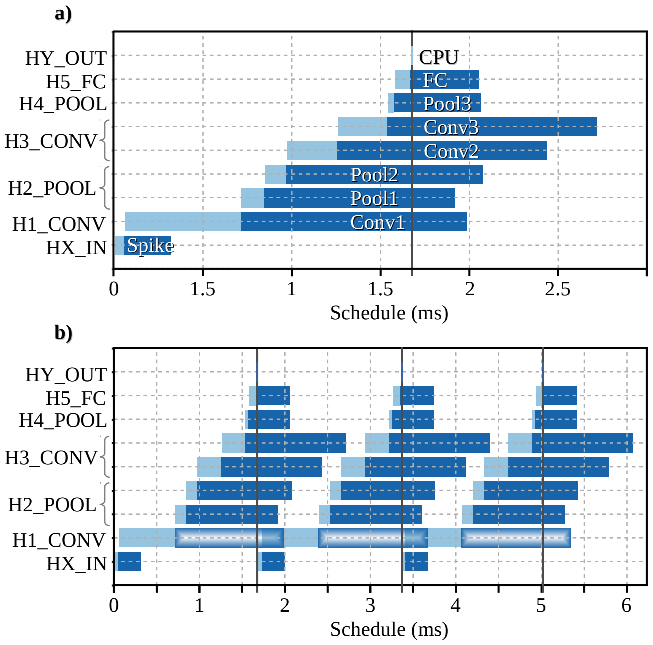

FIGURE 18. Performance of processing units using logarithmic approximation. a) Illustrates computation schedule. b) Illustrates cyclic computation schedule.

a latency $T_{SC} = 3.183ms$. As the benchmark result, we get a $10.77\times$ latency enhancement and an accuracy degradation of 0.32%.

As a demonstration of the proposed dot-product architecture, the SbS network simulation on hardware PUs with synaptic representation using 5-bit custom floating-point (4-bit exponent, 1-bit mantissa) and 4-bit logarithmic (4-bit exponent) achieve $20.49\times$ latency enhancement and accuracy of 98.97% and 98.84%, respectively. As a result, we obtain an accuracy degradation of 0.33% and 0.46%, respectively. Moreover, adding 50% of noise amplitude to the input images, the SbS network simulation presents an accuracy degradation of 0.67% and 4.08%, respectively. As output quality monitor, the resulting noise tolerance report demonstrates a sufficient QoR for minor impact on the output quality of the neural network using the proposed approximate computing technique.

Regarding resource utilization and power dissipation, the *Conv* processing units reach up to 43.24% of BRAM reduction, and 12.35% of improvement in energy efficiency over the standard floating-point implementation. The experimental results of the design exploration are summarized in **Tab.** 11.

VI. CONCLUSIONS

In this publication, we accelerate SbS neural networks with a dot-product functional unit based on approximate computing. This approach reduces computational latency, memory footprint, and power dissipation while preserving inference accuracy.

For output quality monitoring, we introduced a noise tolerance plot to visualize the impact of the approximate computing technique on the accuracy of the neural network.

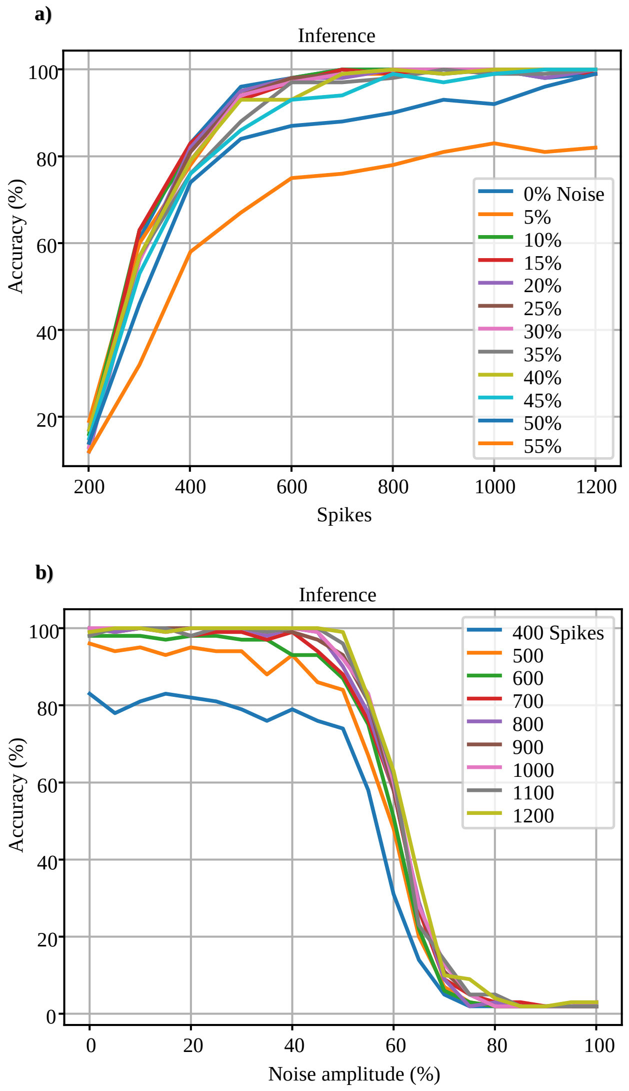

FIGURE 19. Noise tolerance on hardware PU using logarithmic approximation. (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

We demonstrate our approach addressing a design exploration flow on a Xilinx Zynq-7020 with a deployment of NMIST classification task, this implementation achieves up to $20.49\times$ latency enhancement, $8\times$ synaptic memory footprint reduction, less than 0.5% of accuracy degradation, with a 12.35% of energy efficiency improvement over the standard floating-point hardware implementation. Furthermore, with positive additive uniformly distributed noise at 50% of amplitude on the input image, the SbS network simulation presents an accuracy degradation of less than 5%. As output quality monitor, the resulting noise-tolerance demonstrates a sufficient QoR for minimal impact on the overall accuracy of the neural network using the proposed approximation technique. These results suggest available room for further

TABLE 11. Experimental results of design exploration.

Implementation	PU	Post-implementation resource utilization				Power (mW)	Latency		Accuracy (%)[e]		
		LUT	FF	DSP	BRAM 18K		T_{SC} (ms)	Gain[d]	Noise 0%	25%	50%
Standard floating-point[a]	Conv	2,765	4,366	19	37	89	3.183	10.77x	98.98	98.96	98.63
	FC	2,649	4,189	8	9	66					
Custom floating-point[b]	Conv	3,139	4,850	19	25	82	1.673	20.49x	98.97	98.94	98.47
	FC	3,265	5,188	8	9	66					
Logarithmic[c]	Conv	3,086	4,804	19	21	78	1.673	20.49x	98.84	98.83	95.22
	FC	3,046	4,873	8	8	66					

[a] Synaptic storage composed of 4-bit exponent and 4-bit mantissa. For computation, each value is promoted to its standard floating-point representation.
[b] Synaptic storage composed of 4-bit exponent and 1-bit mantissa.
[c] Synaptic storage composed of 4-bit exponent.
[d] Latency gain with respect to the CPU computation ($T_{SC} = 34.279ms$).
[e] Accuracy on 10,000 image test set at 1000 spikes.

and more aggressive approximation approaches.

In conclusion, based on the relaxed need for fully accurate or deterministic computation of SbS neural networks, approximate computing techniques allow substantial enhancement in processing efficiency with moderated accuracy degradation.

ACKNOWLEDGMENTS

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

REFERENCES

[1] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, vol. 61, pp. 85–117, 2015.

[2] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[3] 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.

[4] 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.

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

[6] 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

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

[8] M. ZHANG, G. Zonghua, and P. Gang, “A survey of neuromorphic computing based on spiking neural networks,” *Chinese Journal of Electronics*, vol. 27, no. 4, pp. 667–674, 2018.

[9] U. Lotrič and P. Bulić, “Applicability of approximate multipliers in hardware neural networks,” *Neurocomputing*, vol. 96, pp. 57–65, 2012.

[10] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, “Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing,” in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 145–150.

[11] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, “Design of power-efficient approximate multipliers for approximate artificial neural networks,” in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, pp. 1–7.

[12] Z. Du, K. Palem, A. Lingamneni, O. Temam, Y. Chen, and C. Wu, “Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators,” in *2014 19th Asia and South Pacific design automation conference (ASP-DAC)*. IEEE, 2014, pp. 201–206.

[13] J. Park, J. H. Choi, and K. Roy, “Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 5, pp. 787–793, 2009.

[14] J. Han and M. Orshansky, “Approximate computing: An emerging paradigm for energy-efficient design,” in *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 2013, pp. 1–6.

[15] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, “Impact: imprecise adders for low-power approximate computing,” in *IEEE/ACM International Symposium on Low Power Electronics and Design*. IEEE, 2011, pp. 409–414.

[16] S. Mittal, “A survey of techniques for approximate computing,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[17] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, “Approximate computing and the quest for computing efficiency,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[18] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, “Spiking neural networks hardware implementations and challenges: A survey,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 2, pp. 1–35, 2019.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[20] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Quantized neural networks: Training neural networks with low precision weights and activations,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[23] B. Moons and M. Verhelst, “A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets,” in *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*. IEEE, 2016, pp. 1–2.

[24] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G.-Y. Wei, “14.3 a 28nm soc with a 1.2 ghz 568nj/prediction sparse deep-neural-network engine with> 0.1 timing error rate tolerance for iot applications,” in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2017, pp. 242–243.

[25] X. Sun, S. Yin, X. Peng, R. Liu, J.-s. Seo, and S. Yu, “Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1423–1428.

[26] N. Rathi, P. Panda, and K. Roy, “Stdp-based pruning of connections and weight quantization in spiking neural networks for energy-efficient recognition,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 668–677, 2018.

[27] S. Sen, S. Venkataramani, and A. Raghunathan, “Approximate computing for spiking neural networks,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 193–198.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[29] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, “Regularization of neural networks using dropconnect,” in *International conference on machine learning*, 2013, pp. 1058–1066.

[30] E. O. Neftci, B. U. Pedroni, S. Joshi, M. Al-Shedivat, and G. Cauwenberghs, “Stochastic synapses enable efficient brain-inspired learning machines,” *Frontiers in neuroscience*, vol. 10, p. 241, 2016.

[31] G. Srinivasan, A. Sengupta, and K. Roy, “Magnetic tunnel junction based long-term short-term stochastic synapse for a spiking neural network with on-chip stdp learning,” *Scientific reports*, vol. 6, p. 29545, 2016.

[32] L. Buesing, J. Bill, B. Nessler, and W. Maass, “Neural dynamics as sampling: a model for stochastic computation in recurrent networks of spiking neurons,” *PLoS Comput Biol*, vol. 7, no. 11, p. e1002211, 2011.

[33] G. Bellec, D. Kappel, W. Maass, and R. Legenstein, “Deep rewiring: Training very sparse deep networks,” *arXiv preprint arXiv:1711.05136*, 2017.

[34] G. K. Chen, R. Kumar, H. E. Sumbul, P. C. Knag, and R. K. Krishnamurthy, “A 4096-neuron 1m-synapse 3.8-pj/sop spiking neural network with on-chip stdp learning and sparse weights in 10-nm finfet cmos,” *IEEE Journal of Solid-State Circuits*, vol. 54, no. 4, pp. 992–1002, 2018.

[35] S. Sheik, S. Paul, C. Augustine, C. Kothapalli, M. M. Khellah, G. Cauwenberghs, and E. Neftci, “Synaptic sampling in hardware spiking neural networks,” in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2016, pp. 2090–2093.

[36] M. Jerry, A. Parihar, B. Grisafe, A. Raychowdhury, and S. Datta, “Ultra-low power probabilistic imt neurons for stochastic sampling machines,” in *2017 Symposium on VLSI Circuits*. IEEE, 2017, pp. T186–T187.

[37] Y. Kim, Y. Zhang, and P. Li, “An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems,” in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 130–137.

[38] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[39] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, “Imagenet large scale visual recognition challenge,” *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

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

[41] Y. Nevarez, A. Garcia-Ortiz, D. Rotermund, and K. R. Pawelzik, “Accelerator framework of spike-by-spike neural networks for inference and incremental learning in embedded systems,” in *2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2020, pp. 1–5.

[42] Y. LeCun, “The mnist database of handwritten digits,” *http://yann. lecun. com/exdb/mnist/*, 1998.

[43] D. Rotermund and K. R. Pawelzik, “Massively parallel fpga hardware for spike-by-spike networks,” *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/06/14/500280

[44] U. Xilinx, “Zynq-7000 all programmable soc: Technical reference manual,” 2015.

YARIB NEVAREZ received the B.E. (Hons) degree in electronics from the Durango Institute of Technology, Durango, Mexico, in 2009, and the M.Sc. degree in Embedded Systems Design from the University of Applied Sciences Bremerhaven, Bremen, Germany, in 2017. He is currently pursuing a Ph.D. degree with the Institute of Electrodynamics and Microelectronics, University of Bremen, Germany. His research interest is focused mainly on System-on-Chip architectures and hardware implementation for deep learning accelerators in Embedded Systems. During his professional experience, he served as a Senior Embedded Software Engineer at Texas Instruments, IBM, Continental Automotive, TOSHIBA, and Carbon Robotics. He has designed and developed software architectures for graphic calculators, automotive systems, robotic drivers, and more.

DAVID ROTERMUND

KLAUS R. PAWELZIK

ALBERTO GARCIA-ORTIZ

• • •