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 on FPGA with Hybrid Custom Floating-Point and Logarithmic Dot-Product Approximation

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: agarcia@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 Spiking neural networks (SNNs) represent a promising alternative to conventional neural networks. In particular, the so called Spike-by-Spike (SbS) neural networks provide an exceptional noise robustness and a reduced complexity. However, deep SbS networks require a memory footprint and a computational cost unsuitable for embedded applications. To address this problem, this work exploits the intrinsic error-resilience of neural networks to improve performance and to reduce the hardware complexity. More precisely, we design a dot-product hardware unit based on approximate computing with configurable quality using hybrid custom floating-point and logarithmic number representation. This approach reduces computational latency, memory footprint, and power dissipation while preserving inference accuracy. To demonstrate our approach, we address a design exploration flow using high-level synthesis and a Xilinx SoC-FPGA. The proposed design reduces $20.5\times$ computational latency and $8\times$ weight memory footprint, with less than 0.5% of accuracy degradation on a 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 intelligence (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, artificial neural networks (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 (MLPs) and convolutional neural networks (CNNs) using continuous activation functions; while the third generation is represented by spiking neural networks using spikes as means for information exchange between groups of neurons. Although the AI research is currently dominated by deep neural networks (DNNs) from the second generation, the SNNs belonging to the third generation are receiving considerable attention [3]–[6].

SNNs offer advantageous robustness and the potential to achieve a power efficiency closer to that of the human brain. SNNs operate reliably using stochastic elements that are inherently non-reliable mechanisms [7]. This provides superior resistance against adversary attacks [5], [8]. Beside robustness, SNNs have further advantages like the possibility of a more efficient asynchronous parallelization and higher energy efficiency than DNNs. For example, Loihi [9], a SNN developed by Intel, can solve LASSO optimization problems with an over three orders of magnitude better energy-delay product than conventional approaches. These advantages are motivating large research programs by major companies (e.g. Intel [9] and IBM [10]) as well as pan-european projects in the domain of spiking networks [4], and attempts to transfer features from SNNs to standard DNNs [11].

SNNs emulate the real behavior of neurons in different levels of detail. The more detailed the biological part is emulated, the greater the computational complexity [12], [13]. Most of today's SNNs use a very detailed model. In contrast, Spike-By-Spike (SbS) neural networks are on the less realistic side of the biological realism scale [5], [14]. In spite of that, SbS still uses stochastic spikes as a means of transmitting information between populations of neurons, and thus retains the robustness advantages of SNNs. Correspondingly, the hardware complexity of the approach is greatly reduced [15], [16].

The conceptual model in SbS (see **Sec.** III-A for a short review) differs fundamentally from conventional ANNs since (a) the building blocks 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 noise-robust stochastic version of Non-Negative Matrix Factorization (NNMF). As discussed in **Sec.** III-A, SbS incorporates the inherent robustness of SNNs and the regular flow of information from CNNs. These properties place the SbS network in between non-spiking NN and stochastically spiking NN, offering advantages from both worlds [14].

The computational demands of ANNs, and in particular SNNs, must be addressed by specialized hardware architectures. A significant research effort has been performed in SNN accelerators, see e.g. **Ref.** [4], [9], [10], [17]–[19]. However, hardware accelerators that focus on SbS have only been partially investigated so far [15], [16]. Enhanced SbS accelerators will have a double impact. From an engineering point of view, they will contribute to the deployment of robust neural networks in small embedded systems [15]; from a scientific point of view, they will facilitate experimentation with SbS methods for neuroscience [5], [20].

A fundamental point that can be optimized in current SbS accelerators is the use of approximation techniques. 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 [21]. 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 [22]–[25]. 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 with hybrid custom floating-point and logarithmic number representation. This hardware unit has a quality configurable scheme based on the bit truncation of the synaptic-weight vector. **Fig.** 1 illustrates the dot-product hardware module with standard floating-point (IEEE 754) arithmetic, and our approach with hybrid custom floating-point as well as logarithmic approximation. As a design parameter, the mantissa bit-width of the weight vector provides a tunable knob to trade-off between efficiency and quality of result (QoR) [26], [27]. Since the lower-order bits have smaller significance than the higher-order bits, truncating them may have only a minor impact on QoR [28], [29]. Further on, we can remove completely the mantissa bits in order to use only the exponent of a floating-point representation. Therefore, the most efficient setup and yet the worst-case quality configuration becomes a logarithmic representation, which consequently leads to significant architectural-level optimizations using only adders and shifters for dot-product approximation in hardware. Moreover, since approximations and noise have qualitatively the same effect [30], we apply noise tolerance plots as an intuitive visual measure to provide insights into the quality degradation of SbS networks under approximate processing effects.

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 multiplying truncated mantissas, and the sum of products is done by accumulating denormalized integer products with barrel shifters, 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 vectors with custom floating-point and logarithmic representation as shown in **Fig.** 1. We evaluate inference latency, accuracy degradation, resource utilization and power dissipation. Experimental results demonstrate $20.5\times$ latency enhancement versus embedded CPU (ARM Cortex-A9 at 666MHz), and less than 0.5% of accuracy degradation on a handwritten digit recognition task.
- We propose a noise tolerance plot as quality monitor,

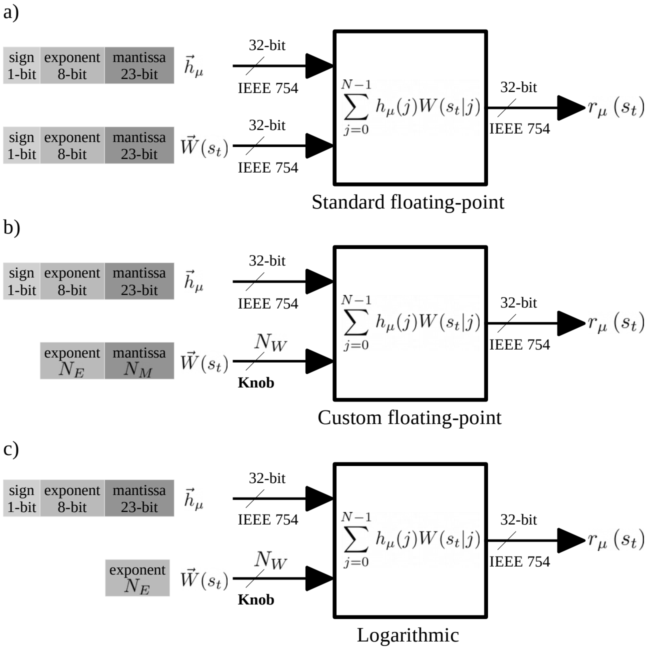

FIGURE 1. Dot-product hardware module with (a) standard floating-point (IEEE 754) arithmetic, (b) hybrid custom floating-point approximation, and (c) hybrid logarithmic approximation.

which serves as an intuitive visual model to provide insights into the accuracy degradation of SbS networks under approximate processing effects.

- Our proposed design for dot-product approximation is adaptable as a building block for other error-resilient applications (e.g., image/video processing).

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 [27]. 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 weights and activation maps to shrink the memory footprint of the large number of parameters representing ANNs, a method known as network compression or quantization. This practice takes advantage of the intrinsic error-tolerance of neural networks, as well as their ability to compensate for approximation while training. In this way, reduced bit precision causes a small accuracy loss [31]–[34].

In hardware development, weight quantization (WQ) has shown up to $2\times$ improvement in energy consumption with an accuracy degradation of less than 1% [35], [36]. Some advanced quantization methods yield to binary neural networks (BNNs) allowing the use of XNORs instead of the conventional costly multiply-accumulate circuits (MACs) [34]. In [37], Sun et al. report an accuracy of 98.43% on handwritten digit classification 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.

In addition to quantization, network pruning reduces the model size by removing structural portions of the parameters and its associated computations [38], [39]. This method has been identified as an effective technique to improve the efficiency of DNN for applications with limited computational budget [40]–[42].

These methods can be used for SNNs as well. In [43], Rathi et al. report up to $3.1\times$ improvement in energy consumption with an accuracy loss of around 3%. Weight quantization allows the designer to realize a trade-off between the accuracy of the SNN application and efficiency of resources. Approximate computing can also be applied at the neuron level, where irrelevant units are deactivated to reduce the computation cost of the SNNs [44]. This computation skipping can be applied randomly on synapses, training ANNs with stochastic synapses improves generalization, resulting in a better accuracy [45], [46]. Such methods are compatible with SNNs and have been tested both during training [47], [48] and operation [49], and even to define the connectivity between layers [50], [51]. Implementations of spiking neuromorphic systems in FPGA [52] and hardware [53] 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 [21].

2) Classical approximate computing

This approach consists of designing processing elements that approximate their computation by employing modified algorithmic logic units [27]. In [54], 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 a handwritten digit 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

[55] on ImageNet [56], the accuracy degradation becomes more important and may not be negligible anymore [34], especially for critical applications such as autonomous driving. Therefore, it is not certain that network compression techniques and approximate computing are suitable for all applications.

B. SPIKE-BY-SPIKE NEURAL NETWORKS ACCELERATORS

Recently, Rotermund et al. demonstrated the feasibility of a neuromorphic SbS IP on a Xilinx Virtex 6 FPGA [16]. It provides a massively parallel architecture, optimized to reduce memory access and suitable for ASIC implementations. Nonetheless, this design is considerably resource-demanding if implemented as a full SbS network in today's embedded technology.

In Ref. [15], 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 embedded software API that allows the construction of scalable sequential SbS networks with configurable hardware acceleration. However, this design works entirely with standard floating-point arithmetic (IEEE 754). This represents a large memory footprint and inadequate computational cost for error-resilient applications on resource-limited devices. In this article, we will 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

Technically, SbS is a spiking neural network approach based on a generative probabilistic model. It 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 $r(s) = \sum_i h(i)W(s|i)$. where $\vec{h}$ is an inference population composed of a group of neurons that compete with each other. 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 [5]

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

where ϵ is a parameter that also 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 $W(s|i)$. The interested reader is referred to [5] for a more detailed exposition.

From a practical point of view, SbS provides a mechanism to obtain a sparse representation of input patterns. Given a set of training samples $\{x_\eta\}$ it learns weights, W, that allow to express the input patterns as a linear sparse non-negative combination of features. During inference, it provides a mechanism for expressing each the test input x_μ as $x_\mu \approx W\,h_\mu$ where all entries are non-negative.

The inference procedure consists in generating indices s_t distributed according to a categorical distribution of the input pattern $s_t \sim \text{Categorical}(x_\mu(0), x_\mu(1), .., x_\mu(N-1))$. Starting with a random h and executing iteratively **Eq.** (1) the SbS algorithms finds h_μ. The fundamental concept of SbS can be extended from vector to matrix inputs. In this case, the linear operation $W\,h_\mu$ can be replaced by a convolution to obtain a convolutional SbS layer.

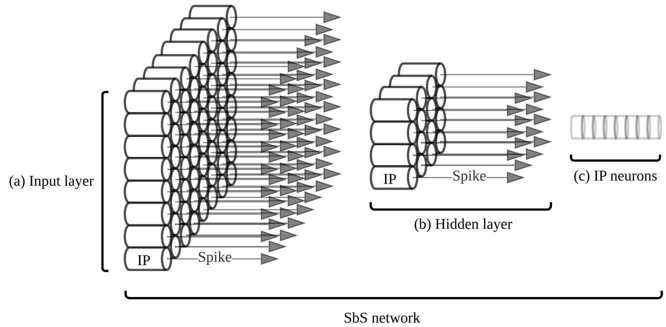

FIGURE 2. SbS IPs as independent computational entities, (a) illustrates an input layer with a massive amount of IPs operating as independent computational entities, (b) shows a hidden layer with an arbitrary amount of IPs as independent computational entities, (c) exhibits a set of neurons grouped in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

Further on, SbS network models can be constructed in sequential layered structures [14], where each layer consists of many inference populations (IPs), which can be simulated independently while the communication between the IPs is organized by a low bandwidth signal – the spikes. Technically, each IP is an independent computational entity (see **Fig.** 2). This allows to design specialized hardware architectures that can be massively parallelized.

To illustrate the advantages of SbS, an example of the noise tolerance of SbS is presented in **Fig.** 3. It compares the classification performance of a SbS network and a standard convolutional network, with the same amount of neurons per layer as well as the same layer structure. We trained on MNIST dataset [57] without noise (see [14] for details). The figure shows the correctness for the MNIST test set with its

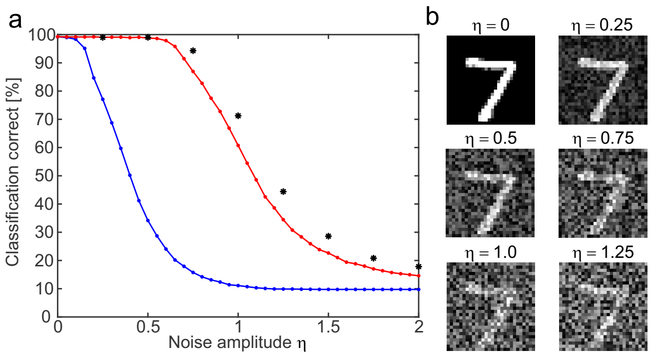

FIGURE 3. (a) Performance classification of SbS NN versus equivalent CNN, and (b) example of the first pattern in the MNIST test data set with different amounts of noise.

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.

IV. SYSTEM DESIGN

In this section, we revise the system design of [15]. In Ref. [15], we presented a scalable hardware architecture composed of generic homogeneous accelerator units (AUs). This design works entirely with standard floating-point arithmetic (IEEE 754), which represents an unnecessary overhead for error-resilient applications. Furthermore, this architecture does not implement stationary synaptic weight matrices in the hardware AUs, resulting in heavy data movement and longer computational latency.

In this publication, we present an enhanced hardware architecture composed of specialized heterogeneous processing units (PUs) with hybrid custom floating-point and logarithmic dot-product approximation. This approach represents an advantageous design for error-resilient applications in resource-constrained devices due to the reduced computational costs and memory footprint. Furthermore, the proposed approach allows the implementation of stationary synaptic weight matrices. These novelties result in an improved overall system design.

Regarding the software architecture, this is structured as a layered object-oriented application framework written in the C programming language. This offers a comprehensive high level embedded software application programming interface (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.** 4.

Algorithm 1: SbS layer update.

input: $L \in \mathbb{R}^{L_W \times L_H \times N}$, $S_t^{in} \in \mathbb{N}^{L_W \times L_H}$, N, ϵ
output: $L^{new} \in \mathbb{R}^{L_W \times L_H \times N}$, $S_t^{out} \in \mathbb{N}^{L_W \times L_H}$
Update layer :
for $L_X \leftarrow 0, L_Y \leftarrow 0$ **to** $L_W - K - 1, L_H - K - 1$ **do**
  Generate spike :
  $th \leftarrow MT19937PseudoRandom()$
  $acu \leftarrow 0$
  for $idx \leftarrow 0$ **to** $N - 1$ **do**
    if $th \leq acu$ **or** $idx = N - 1$ **then**
      $S_t^{out}(L_X, L_Y) \leftarrow idx$
      goto *Update IP*
    end if
    $acu \leftarrow acu + \vec{h}_\mu(idx)$
  end for
  Update IP :
  $\vec{h}_\mu \leftarrow L(L_X, L_Y)$
  for $K_X \leftarrow 0, K_Y \leftarrow 0$ **to** $K - 1, K - 1$ **do**
    $s_t \leftarrow S_t^{in}(L_X + K_X, L_Y + K_Y)$
    $\vec{w} \leftarrow W(K_X, K_Y, s_t)$
    $\vec{p} \leftarrow 0$
    Dot-product :
    $r_\mu \leftarrow 0$
    for $j \leftarrow 0$ **to** $N - 1$ **do**
      $\vec{p}(j) \leftarrow \vec{h}_\mu(j)\vec{w}(j)$
      $r_\mu \leftarrow r_\mu + \vec{p}(j)$
    end for
    if $r_\mu(s_t) \neq 0$ **then**
      Update IP vector :
      for $i \leftarrow 0$ **to** $N - 1$ **do**
        $h_\mu^{new}(i) \leftarrow \frac{1}{1+\epsilon}\left(h_\mu(i) + \epsilon\frac{\vec{p}(i)}{r_\mu}\right)$
      end for
      Set the new IP vector on the layer :
      $L^{new}(L_X, L_Y) \leftarrow \vec{h}_\mu^{new}$
    end if
  end for
end for

A. HARDWARE ARCHITECTURE

As a hardware/software co-design, the system architecture is an embedded CPU+FPGA-based platform, where the acceleration of SbS network computation is based on asynchronous[1] execution in parallel heterogeneous processing units: *Spike* (input layer), *Conv* (convolution), *Pool* (pooling), and *FC* (fully connected). **Fig.** 5 illustrates the system hardware architecture as a scalable structure. For hyperparameter configuration, each PU uses AXI-Lite interface. For data transfer, each PU uses AXI-Stream interfaces via Direct Memory Access (DMA) allowing data movement with high transfer rate. Each PU asserts an interrupt flag once the job or transaction is complete. This interrupt event is handled by the

[1]The system is synchronous at the circuit level, but the execution is asynchronous in terms of jobs.

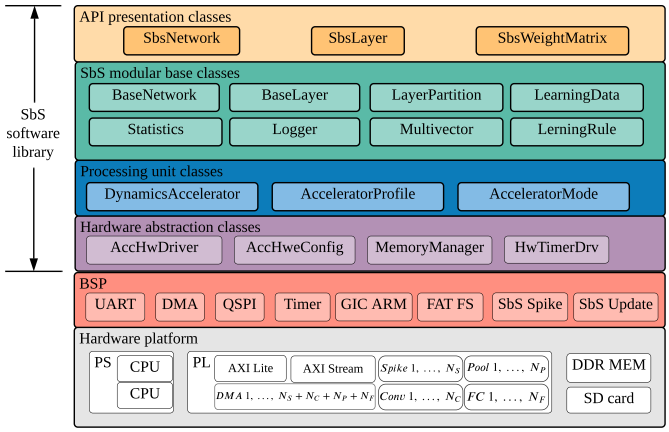

FIGURE 4. System-level overview of the embedded software architecture.

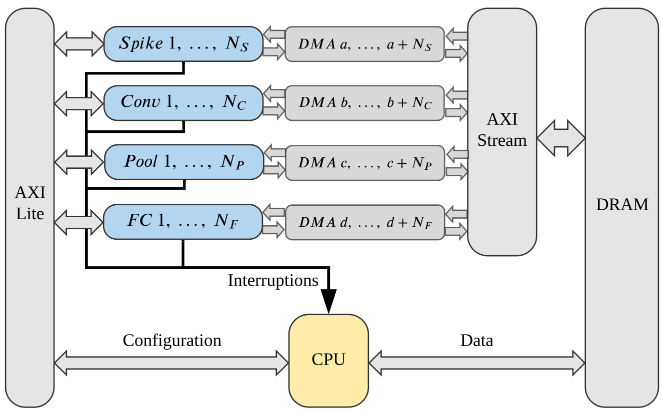

FIGURE 5. System-level hardware architecture with scalable number of heterogeneous PUs: *Spike*, *Conv*, *Pool*, and *FC*

embedded CPU to collect results and start a new transaction.

The hardware architecture can resize its resource utilization by changing the number of PUs instances prior to the hardware synthesis, this provides scalability with a good trade-off between area and throughput. The dedicated PUs for *Conv* and *FC* implement the proposed dot-product approximation 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 (BRAM) the hyperparameters to compute the IP dynamics: ϵ as the epsilon, N as the length of $\vec{h}_\mu \in \mathbb{R}^N$, $K \in \mathbb{N}$ as the size of the convolution kernel, and $H \in \mathbb{N}$ as the number of IPs to process per transaction. H is the number of IPs forming a layer or a partition.

Additionally, the processing unit also stores in on-chip memory (BRAM) 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 $\sum_{j=0}^{N-1} W(s_t|j) = 1$ [14]. Hence, W employs only positive normalized real numbers. Therefore, W is deployed using a reduced floating-point or logarithmic representation as follows:

- Custom floating-point representation. In this case, W is deployed with a reduced floating-point representation using the user defined bit-width for the exponent and for the mantissa. For example, 4-bit exponent, 1-bit mantissa; as a result: 5-bit custom floating-point. The methodology to determine the required bit-width is described in Section IV-C.
- Logarithmic representation. 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 omitted, and the values are represented in its positive form. 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 methodology to determine the required bit-width is described in Section IV-C.

In order to deploy different SbS network models, the *Conv* processing units can be configured with different synaptic weight matrices and hyperparameters as required through the embedded software.

2) Computation mode

In this mode of operation, the PU executes a transaction to process a group of IPs using the previously given hyperparameters and synaptic weight matrix. This process operates in six stages as shown in **Fig.** 6. In the first two stages, the PU receives $\vec{h}_\mu \in \mathbb{R}^N$, then the PU calculates the emitted 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 $\vec{h}_\mu^{new} \in \mathbb{R}^N$ (updated IP). This process repeats 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.** 6(d)) operates in two modular stages: *dot-product* and *neuron update*. First, the *dot-product* module calculates the sum of pairwise products of $\vec{h}_\mu$ and $\vec{W}(s_t)$, each pairwise product is stored as intermediate results. Subsequently, the *neuron update* module calculates **Eq.** (1) reusing previous results and parameters.

The calculation of the dot-product of **Eq.** (1) represents a considerable computational cost using standard floating-point in non-quantized network models. Fortunately, the pair product of $h_\mu(j)$ and $W(s_t|j)$ was defined by us as an

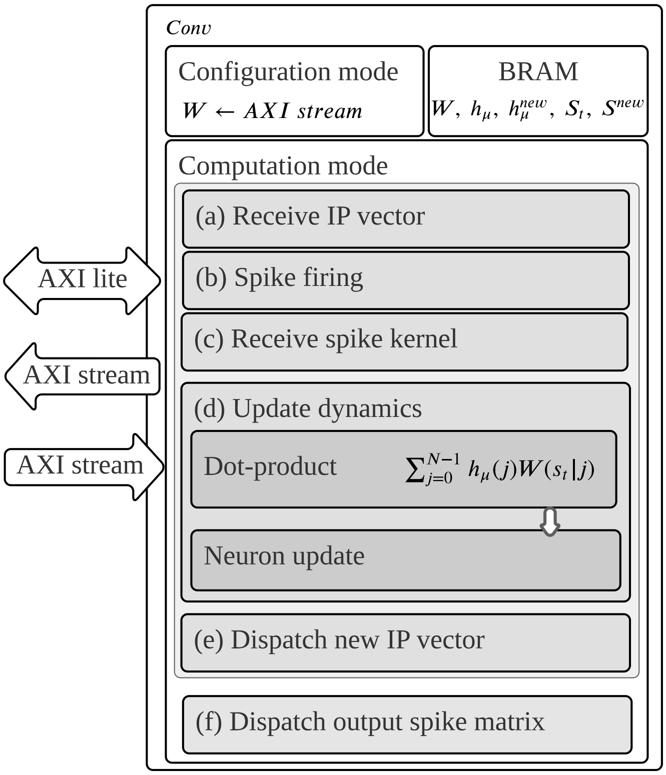

FIGURE 6. The *Conv* processing unit and its six stages: (a) receive IP vector, (b) spike firing, (c) receive spike kernel, (d) update dynamics, (e) dispatch new IP vector, (f) dispatch output spike matrix.

approximable factor in the dot-product 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 dot-product hardware 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 $\vec{W}(s_t)$, which provides a tunable trade-off between resource utilization and QoR. Since the lower-order bits have smaller significance than the higher-order bits, removing them may have only a minor impact on QoR. We designate this as hybrid custom floating-point approximation (see **Fig.** 7(a)).

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

Further on, we remove the mantissa bits completely in order to use only the exponent of a floating-point representation. Hence, the worst-case quality and yet the most efficient configuration becomes a logarithmic representation. Consequently, this structure leads to advantageous architectural optimizations using only adders and barrel shifters for dot-product approximation in hardware. We designate this as hybrid logarithmic approximation (see **Fig.** 7(b)).

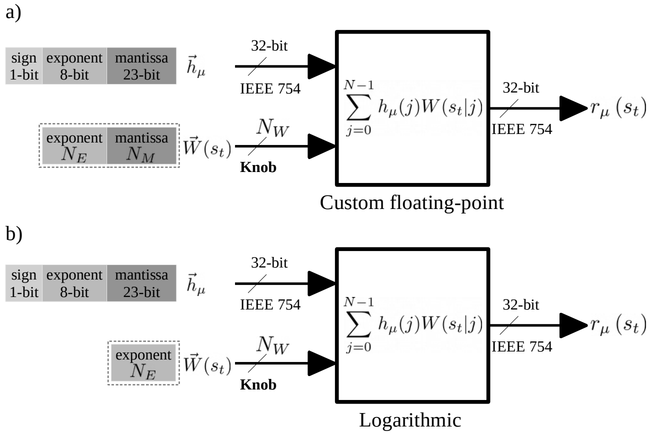

FIGURE 7. Dot-product hardware module with (a) hybrid custom floating-point approximation, and (b) hybrid logarithmic approximation.

In order to determine the required bit-width for the number representation, we use **Eq.** (3), **Eq.** (4), and **Eq.** (5).

$$E_{\min} = \log_2(\min_{\forall i}(W(i))) \quad (3)$$

$$N_E = \lceil \log_2(|E_{\min}|) \rceil \quad (4)$$

$$N_W = N_E + N_M \quad (5)$$

The **Eq.** (3) obtains the exponent of the minimum entry value in the synaptic weight matrix. 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, by searching for the smallest value, we obtain the biggest negative exponent ($E_{\min}$). Then, the **Eq.** (4) obtains the necessary bit-width to represent the exponent (N_E). Finally, we obtain the total bit-width by incorporating both exponent and mantissa bit-widths in **Eq.** (5). N_M denotes the mantissa bit-width, this is a knob parameter that is tuned by the designer to trade-off between resource utilization and QoR. The bit-width concept is illustrated in **Fig.** 7.

In this section, we will present three pipelined hardware modules with standard floating-point (IEEE 754) computation, hybrid custom floating-point approximation, and hybrid logarithmic approximation.

1) Dot-product with standard floating-point computation

The hardware module to calculate the dot-product with standard floating-point computation is shown in **Fig.** 8. This diagram presents the hardware blocks and their clock cycle schedule. This module loads both $h_\mu(j)$ and $W(s|j)$ from BRAM, then the PU executes the pairwise product (**Fig.** 8(c)) and accumulation (**Fig.** 8(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.** (6), where N is the dot-product length.

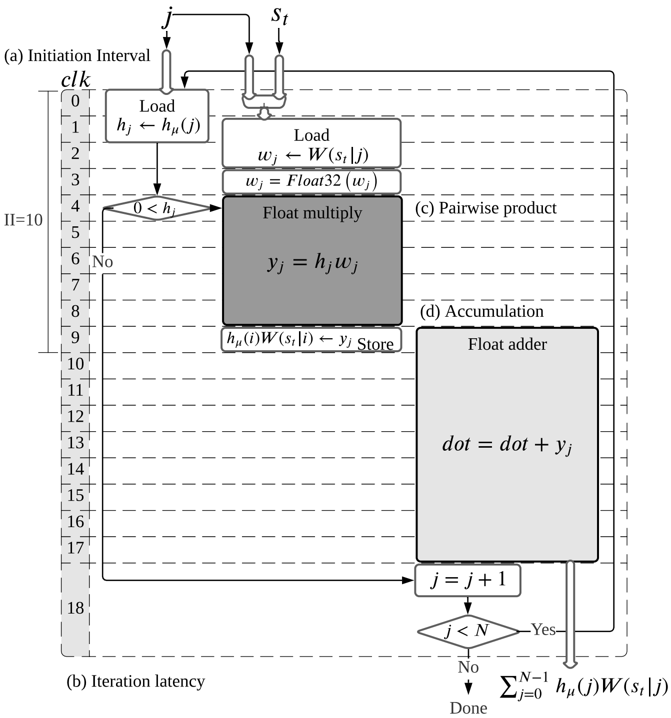

FIGURE 8. Dot-product hardware module with standard floating-point (IEEE 754) computation, (a) exhibits the initiation interval of 10 clock cycles, (b) presents the iteration latency of 19 clock cycles, (c) shows the pairwise product block in dark-gray, and (d) illustrates the accumulation block in light-gray.

This latency equation is obtained from the general pipelined hardware latency formula: $L = (N-1)\,II + IL$, where II is the initiation interval (**Fig.** 8(a)), and IL is the iteration latency (**Fig.** 8(b)). Both II and IL are obtained from the high-level synthesis analysis. The equation for the latency with standard 32-bit floating-point is:

$$L_{f32} = 10N + 9 \tag{6}$$

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.** 8(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.** 8(d)), the synthesis infers a hardware block with a latency cost of 9 clock cycles. Seemingly, this block would handle mantissas alignment, addition, and correction if needed. Therefore, the use of standard floating-point in high-level synthesis results in high computational cost, which represents unnecessary overhead in error-tolerant applications.

2) Dot-product with hybrid custom floating-point and logarithmic approximation

The hardware module to calculate dot-product with hybrid custom floating-point approximation is shown in **Fig.** 9. In this design, $h_\mu(j)$ uses standard 32-bit floating-point number representation, and $W(s|j)$ 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 and resource utilization, thus, energy consumption.

As the most efficient setup and yet the worst-case quality configuration, by completely truncating the mantissa of $W(s|j)$ leads to a slightly different hardware architecture using only adders and shifters, which computes the dot-product with hybrid logarithmic approximation. This is shown in **Fig.** 10.

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

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

- Phase I, *Computation*:
  This phase approximates the magnitude of the dot-product in a denormalized representation. This is calculated in two iterative steps over each vector element: *pairwise product* and *accumulation*, where *pairwise product* is executed either in hybrid custom floating-point or hybrid logarithmic approximation described below.
  - Pairwise product.
    - Hybrid custom floating-point approximation. As shown in **Fig.** 9(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 get $h_\mu(j)W(s_t|j)$ as an intermediate result which is stored for future reuse in the neuron update calculation. In this design the pairwise product has a latency of 5 clock cycles.
    - Hybrid logarithmic approximation. As shown in **Fig.** 10(c) in dark-gray, the pairwise product is approximated by adding $W(s|i)$ to the exponent of $h_\mu(i)$, since $W(s|j)$ values are represented in the logarithmic domain and $h_\mu(j)$ in standard floating-point. In this design the pairwise product has a latency of one clock cycle.
  - Accumulation. As shown in both **Fig.** 9(d) and **Fig.** 10(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). Then, this denormalized representation is accumulated to obtain the approximated magnitude of the dot-product.

  The process of pairwise product and accumulation iterates over each element of the vectors. The computation latency is given by **Eq.** (7) for hybrid custom floating-point, and **Eq.** (8) for hybrid 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.

$$L_{custom} = 2N + 11 \tag{7}$$

$$L_{log} = 2N + 7 \tag{8}$$

- Phase II, *Threshold-test*:
The accumulated denormalized magnitude is tested to be above of a predefined threshold, it must be above zero, since the dot-product is the 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 convert it to 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 with hybrid custom floating-point and hybrid logarithmic approximation is the accumulated latency of the three phases.

The proposed architectures with approximation approach exceeds the performance of the design with standard floating-point. This 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.

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 [58]. The Zynq-7020 architecture conveniently maps the custom logic 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.** 11 for handwritten digit classification task using MNIST data set. 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. There, the SbS network is built as a sequential model using the API from the SbS embedded software framework [15]. This API allows to configure the computational workload of the neural network, which can be distributed among the hardware processing units and the embedded CPU.

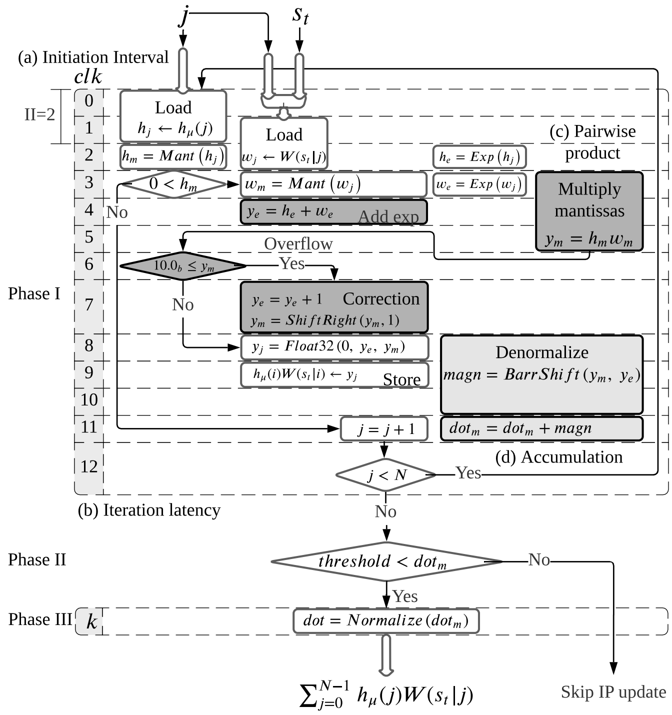

FIGURE 9. Dot-product hardware module with hybrid custom floating-point approximation, (a) exhibits the initiation interval of 2 clock cycles, (b) presents the iteration latency of 13 clock cycles, (c) shows the pairwise product blocks in dark-gray, and (d) illustrates the accumulation blocks in light-gray.

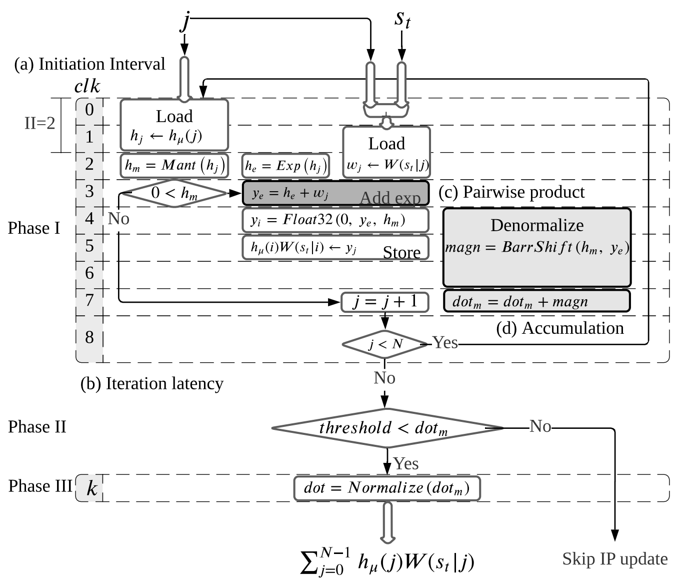

FIGURE 10. Dot-product hardware module with hybrid logarithmic approximation, (a) exhibits the initiation interval of 2 clock cycles, (b) presents the iteration latency of 9 clock cycles, (c) shows the pairwise product block in dark-gray, and (d) illustrates the accumulation blocks in light-gray.

For the evaluation of our approach, we address a design exploration by reviewing the computational latency, inference accuracy, resource utilization, and power dissipation. First, we benchmark the performance of SbS network simulation on the embedded CPU, and then repeat the measurements

TABLE 1. Computation on embedded 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

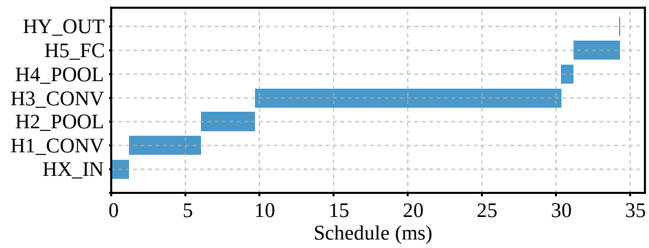

FIGURE 12. Computation on embedded CPU.

on hardware processing units with standard floating-point computation. Afterwards, we evaluate our dot-product architecture, addressing a design exploration with hybrid custom floating-point approximation, as well as the hybrid logarithmic approximation. Finally, we present a discussion of the presented results.

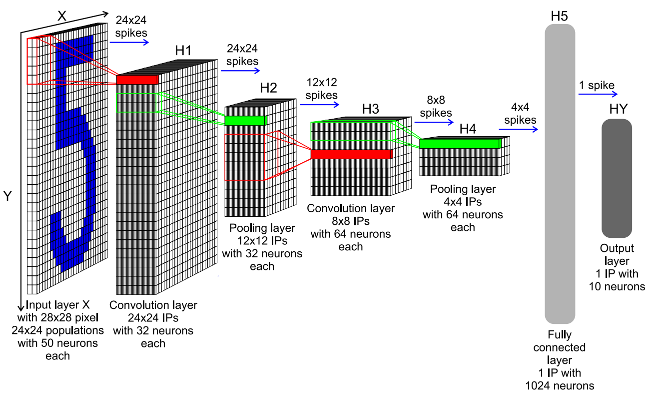

FIGURE 11. SbS network structure for MNIST classification task. Input *X*: Input layer with 28×28 normalization modules for 28×28 input pixel. From this layer spikes are send to layer *H1*. *H1*: Convolution layer *H1* with 24×24 IPs with 32 neurons each. Every IP processes the spikes from 5×5 spatial 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.

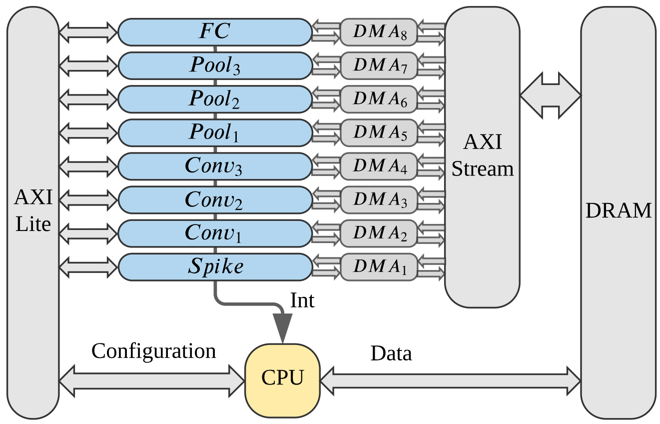

FIGURE 13. System overview of the top-level architecture with 8 processing units.

A. PERFORMANCE BENCHMARK

1) Benchmark on embedded 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 $1.658W$.

The SbS network computation on the CPU achieves a latency of $34.28ms$ per spike with an accuracy of 99.3% correct classification on the $10,000$ image test set with 1000 spikes. The latency and schedule of the SbS network computation are displayed in **Tab.** 1 and **Fig.** 12 respectively.

2) Benchmark on processing units with standard floating-point computation

To benchmark the computation on hardware PUs with standard floating-point, we implement the system architecture shown in **Fig.** 13. 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 layers of the neural network with the most neurons are partitioned for asynchronous parallel processing. Since *H2_POOL* and *H3_CONV* are the layers with the most neurons, the computational workload is distributed between two PUs for each one of these layers. The output layer *HY_OUT* is fully processed by the CPU, since it is the layer with fewest neurons. The hardware mapping and the computation schedule of this deployment are displayed in **Tab.** 2 and **Fig.** 14.

In the computation schedule, the following terms are defined as follows: $t_s(n)$ as the start time for the processing of the neural network layer (as a compute 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 duration in which the CPU writes a DRAM buffer with $\vec{h}_\mu$ (vector of neuron latent variables) of the current processing layer and S_t (input spike matrix) from its preceding layer. This buffer is streamed to the PU via DMA.

TABLE 2. Performance of processing units with standard floating-point (IEEE 754) 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

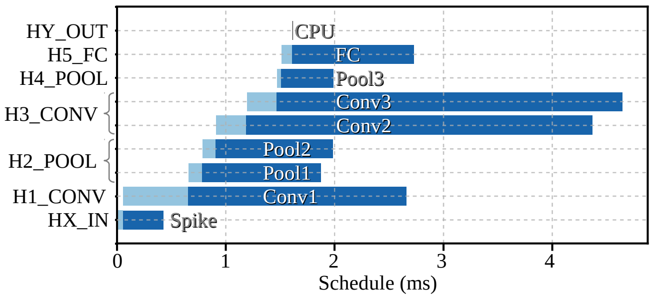

FIGURE 14. Performance of processing units with standard floating-point (IEEE 754) computation.

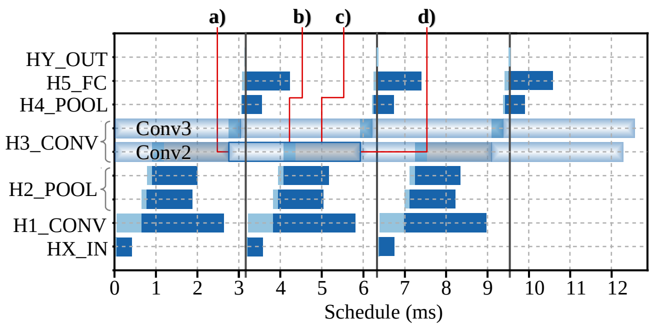

FIGURE 15. Performance bottleneck of cyclic computation on processing units with standard floating-point (IEEE 754) arithmetic, (a) exhibits the starting of t_{PU} of *Conv2* on a previous computation cycle, (b) presents t_{CPU} of *Conv2* on the current computation cycle, (c) shows the CPU waiting time (in gray color) for *Conv2* as a busy resource (awaiting for *Conv2* interruption), and (d) illustrates the t_f from the previous computation cycle, the starting of t_{PU} on the current computation cycle (*Conv2* interruption on completion, and start current computation cycle).

The total execution time of the CPU is defined by **Eq.** (9). In a cyclic spiking 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.** (11). The total execution time of the network computation is the last finish time (t_f) in the schedule defined by **Eq.** (12).

$$T_{CPU} = \sum_{n \in L} t_{CPU}(n) \tag{9}$$

$$T_{PU} = \max_{n \in L}(t_{PU}(n)) \tag{10}$$

$$T_{SC} = \begin{cases} T_{PU}, & \text{if } T_{CPU} \leq T_{PU} \\ T_{CPU}, & \text{otherwise} \end{cases} \tag{11}$$

$$T_f = \max_{n \in L}(t_f(n)) \tag{12}$$

Using standard floating-point requires a high computational cost. As the largest 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.** 15. 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. In contrast, as the smallest layer, the computational workload of *HY_OUT* is fully processed by the CPU. **Tab.** 2 and **Fig.** 14 show $4\mu s$ as the processing latency of *HY_OUT*. This latency is negligible compared to the overall performance assessment. Accelerating *HY_OUT* would yield a negligible gain. Moreover, assigning a dedicated hardware PU to *HY_OUT* would add data transfer and hardware interruption handling overheads, which makes this unprofitable.

Applying **Eq.** (11), we obtain a latency of $3.18ms$ per spike cycle. This deployment achieves an accuracy of 98.98% correct classification on the $10,000$ image test set with 1000 spikes.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 3. Each *Conv* PU instantiates an on-chip stationary weight matrix of $52,000$ entries, wish is sufficient 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-bit 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 Section V-B1.

TABLE 3. Resource utilization and power dissipation of processing units with standard floating-point (IEEE 754) computation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Spike	2,640	4,903	2	2	38
Conv	2,765	4,366	19	37	89
Pool	2,273	3,762	5	3	59
FC	2,649	4,189	8	9	66

The implementation of dot-product with standard floating-point arithmetic (IEEE 754) utilizes proprietary multiplier and adder floating-point operator cores. Vivado HLS accomplishes floating-point arithmetic operations by mapping them onto Xilinx LogiCORE IP cores, these floating-point operator cores are instantiated in the resultant RTL [59]. In this case, the implementation of the dot-product with the standard floating-point computation reuses the multiplier and adder cores already instantiated and used in other computation sections of *Conv* and *FC* processing units. The post-implementation resource utilization and power dissipation of the floating-point operator cores are shown in **Tab.** 4.

TABLE 4. Resource utilization and power dissipation of multiplier and adder floating-point (IEEE 754) operator cores.

Core operation	DSP	FF	LUT	Latency (clk)	Power (mW)
Multiplier	3	151	325	4	7
Adder	2	324	424	8	6

3) Benchmark on noise tolerance plot

The noise tolerance plot serves as an intuitive visual model used to provide insights into accuracy degradation under approximate processing effects. This plot reveals inherent error-resilience, and hence, approximation-resilience. As an application-specific quality metric, this plot offers an effective method to estimate the overall quality degradation of the SbS network under different approximate processing effects, since both approximations and noise have qualitatively the same effect [30].

In order to experimentally obtain the noise tolerance plot, we measure the inference accuracy of the neural network with increasing number of spikes. Then we repeat the measurements with uniformly distributed noise applied on the input images. We gradually ascend the levels of the noise amplitude, until accuracy degradation is detected. **Fig.** 16 demonstrates this method using 100 sample images.

As benchmark, the tolerance plot in **Fig.** 16 revels accuracy degradation having 50% noise and convergence with 400 spikes. In this case, the particular SbS network with precise processing demonstrates a remarkable inherent error resilience, hence, a great opportunity for approximate processing.

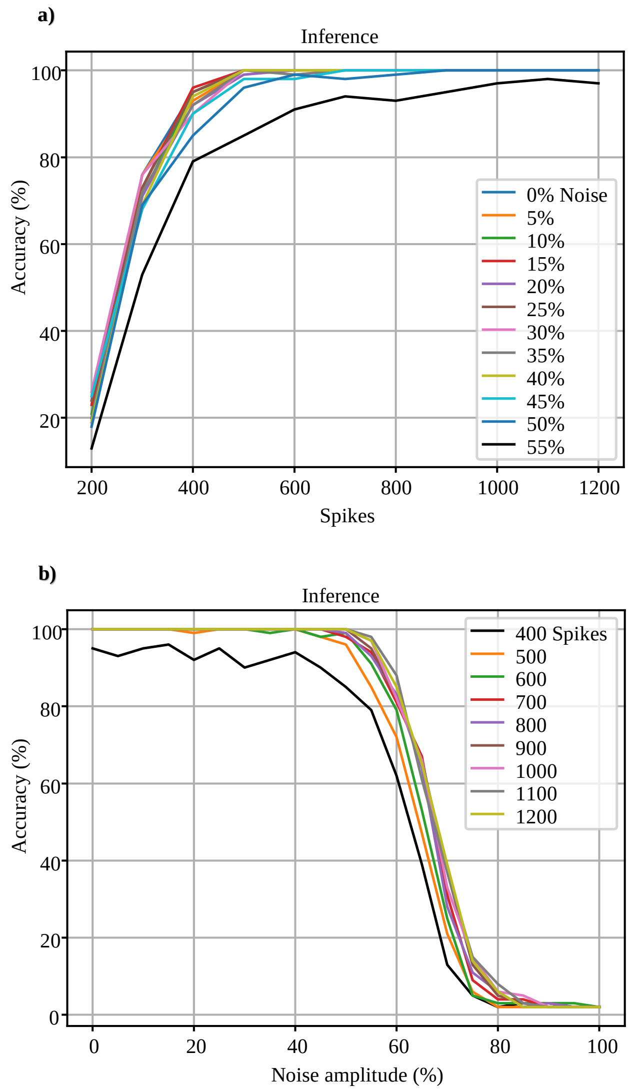

FIGURE 16. Noise tolerance on hardware PU with standard floating-point (IEEE 754) computation (benchmark/reference), (a) exhibits accuracy degradation applying 50% of noise amplitude, and (b) illustrates convergence of inference with 400 spikes.

B. DESIGN EXPLORATION WITH HYBRID CUSTOM FLOATING-POINT AND LOGARITHMIC APPROXIMATION

In this section, we address a design exploration to evaluate our approach for SbS neural network simulation using hybrid 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 the overall performance, the inference accuracy, the resource utilization, and the power dissipation.

1) Parameters for numeric representation of synaptic weight matrix

We obtain information for the numerical representation of the synaptic weight matrices from their $\log_2$-histograms presented in **Fig.** 17. These histograms show the distribution of synaptic weight values in each matrix. We observe that the minimum integer exponent value is -13. Hence, applying **Eq.** (3) and **Eq.** (4) to the given SbS network, we obtain $E_{\min} = -13$ and $N_E = 4$, respectively. Therefore, 4-bits are required for the absolute binary representation of the exponents.

For quality configurability, the mantissa bit-width is a knob parameter that is tuned by the designer. This procedure 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.

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

For this design exploration, we use a custom floating-point representation composed of 4-bit exponent and 1-bit man-

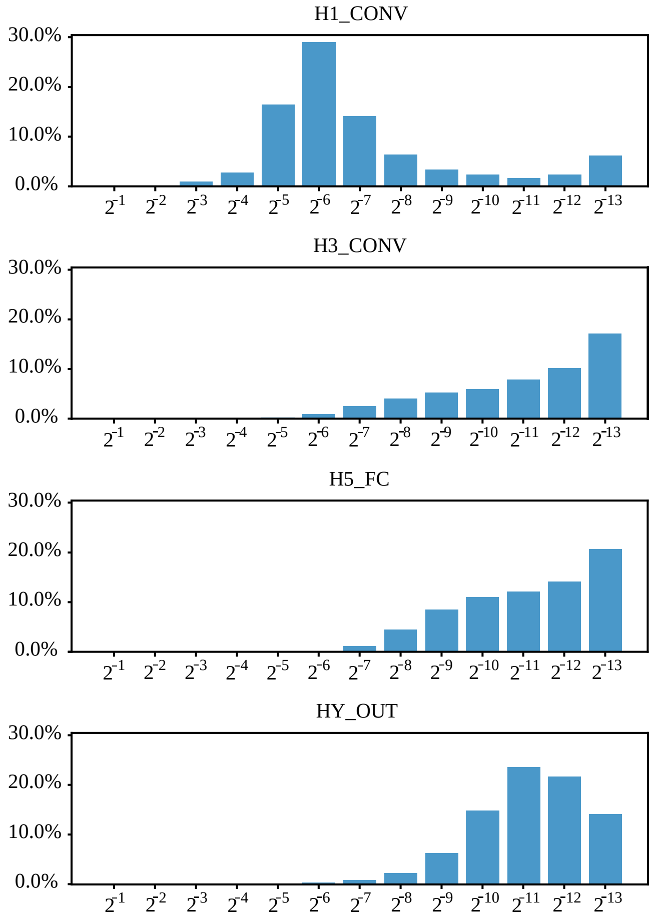

FIGURE 17. $\log_2$-histogram of each synaptic weight matrix showing the percentage of matrix elements with given integer exponent.

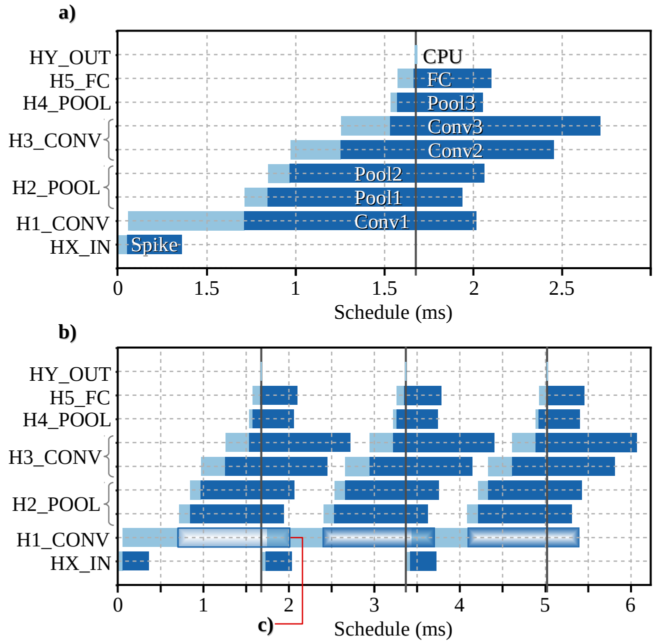

FIGURE 18. Performance on processing units with hybrid custom floating-point approximation, (a) exhibits computation schedule, (b) presents cyclic computation schedule, and (c) shows the performance of *Conv2* from a previous computation cycle during the preprocessing of *H1_CONV* on the current computation cycle without bottleneck.

tissa. This format is used for the synaptic weight vector on the proposed dot-product architecture. Each *Conv* PU instantiates an on-chip stationary weight matrix for $52,000$ entries of 5-bit. The available memory size is large 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 the processing unit of the fully connected layer (*FC*). However, due to lack of BRAM resources, this PU can not instantiate on-chip stationary synaptic weight matrix. Instead, *FC* receives the $\vec{W}(s_t)$ (weight vectors) during operation as well as $\vec{h}_\mu$ and S_t. The hardware mapping and the computation schedule of this implementation are displayed in **Tab.** 6 and **Fig.** 18.

As shown in the computation schedule in **Tab.** 6 and **Fig.** 18, this implementation achieves a maximum hardware PU latency of $1.30ms$ according to **Eq.** (10), and a CPU latency of $1.67ms$. Therefore, applying **Eq.** (11), we obtain a latency of $1.67ms$ per spike cycle as shown in **Fig.** 18. In this case, the cyclic bottleneck is in the performance of the CPU.

This configuration achieves an accuracy of 98.97% correct classification on the $10,000$ image test set with 1000 spikes. This indicates an accuracy degradation of 0.33%. For output quality monitoring, the noise tolerance plot in **Fig.** 19 revels accuracy degradation for noise higher than 50% on the input images, and convergence of inference with 400 spikes. Thus, the particular SbS network implementation under approximate processing effects demonstrates a minimal impact on the overall accuracy. This proves an inherent error resilience, and hence, remaining approximation budget.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 5 .

TABLE 5. Resource utilization and power dissipation of processing units with hybrid custom floating-point approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,139	4,850	19	25	82
FC	3,265	5,188	8	9	66

3) Design exploration for dot-product whit hybrid logarithmic approximation

As the most efficient setup and yet the worst-case quality configuration, we use a 4-bit integer exponent for logarithmic representation of the synaptic weight matrix. Each *Conv* processing unit implements the proposed dot-product architecture including 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 the *FC* processing unit without stationary synaptic weight matrix. The hardware mapping and the computation schedule of this implementation are displayed in **Tab.** 7 and **Fig.** 20.

TABLE 6. Performance of hardware processing units with hybrid 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

TABLE 7. Performance of hardware processing units with hybrid 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

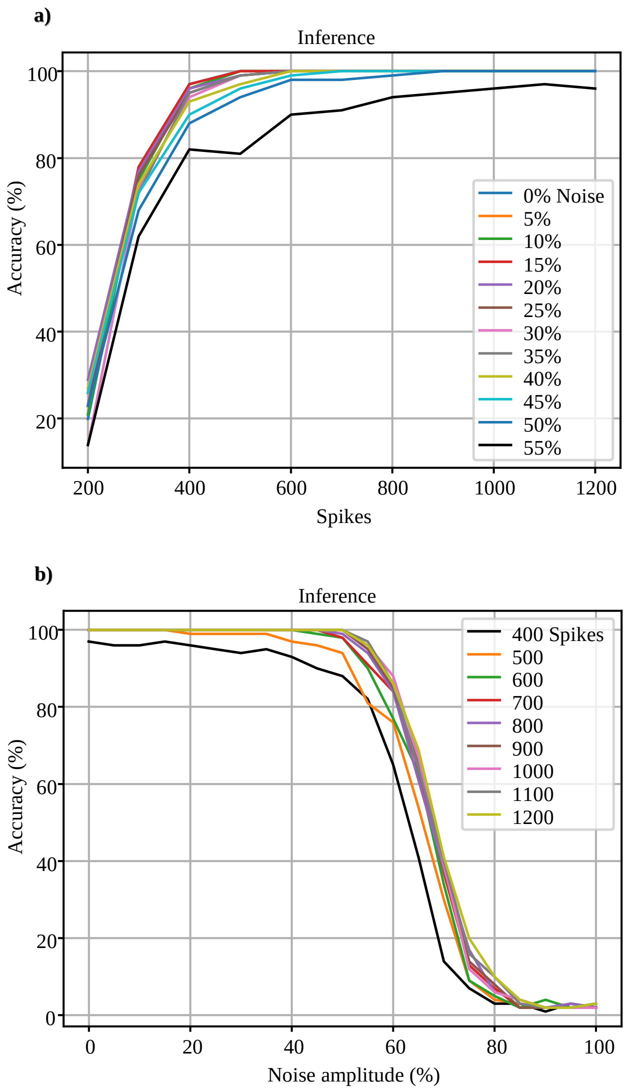

FIGURE 19. Noise tolerance on hardware PU with custom floating-point approximation, (a) exhibits accuracy degradation applying 50% of noise amplitude, and (b) illustrates convergence of inference with 400 spikes.

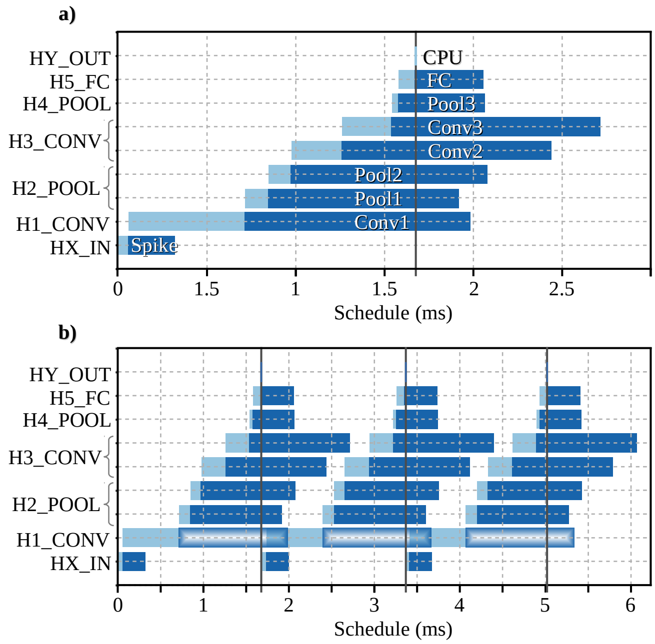

FIGURE 20. Performance of processing units with hybrid logarithmic approximation, (a) exhibits computation schedule, and (b) illustrates cyclic computation schedule.

As shown in the computation schedule in **Tab.** 7 and **Fig.** 20, this implementation achieves a maximum hardware PU latency of $1.27ms$ according to **Eq.** (10), and a CPU latency of $1.67ms$. Therefore, applying **Eq.** (11), we obtain a latency of $1.67ms$ per spike cycle as shown in **Fig.** 20. 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 with 1000 spikes. This indicates an accuracy degradation of 0.46%. For output quality monitoring, the noise tolerance plot in **Fig.** 21 revels accuracy degradation having 40% noise on the input images, and convergence of inference with 600 spikes. The particular SbS network implementation under approximate processing demonstrates a minor impact on the overall accuracy. This exhibits remaining budget for further approximate processing approaches.

The post-implementation resource utilization and power dissipation are shown in **Tab.** 8 .

C. RESULTS AND DISCUSSION

As a reference, the SbS network simulation on embedded CPU using standard 32-bit floating-point achieves an accu-

TABLE 8. Resource utilization and power dissipation of processing units with hybrid logarithmic approximation.

PU	LUT	FF	DSP	BRAM 18K	Power (mW)
Conv	3,086	4,804	19	21	78
FC	3,046	4,873	8	8	66

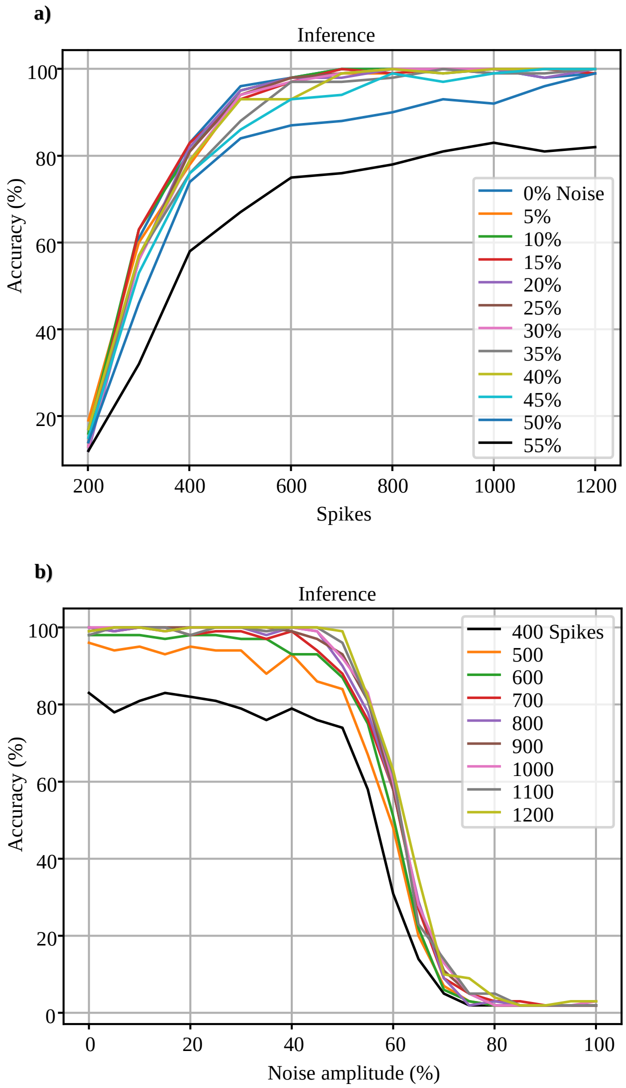

FIGURE 21. Noise tolerance on hardware PU with hybrid logarithmic approximation, (a) exhibits accuracy degradation applying 40% of noise amplitude, (b) illustrates convergence of inference with 600 spikes.

racy of 99.3% with a latency of $T_{SC} = 34.28ms$. As a second reference point, the network simulation on hardware processing units with standard floating-point achieves an accuracy of 98.98% with a latency $T_{SC} = 3.18ms$. As result we get a $10.7\times$ latency enhancement and an accuracy degradation of 0.32%. The tolerance plot in **Fig.** 16 reveals accuracy degradation having 50% noise on the input images, and convergence of inference with 400 spikes. In this case, the SbS network deployment with precise computing proves extraordinary inherent error resilience, and hence, this represents a great potential for approximate processing.

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 $20.5\times$ latency enhancement and accuracy of 98.97% and 98.84%, respectively. This results in an accuracy degradation of 0.33% and 0.46%, respectively. For output quality monitoring, the noise tolerance plot in **Fig.** 19 and **Fig.** 21 reveal accuracy degradation when having 50% and 40% noise on the input images, and convergence of inference with 400 and 600 spikes, respectively. Therefore, the design exploration under the proposed approximate computing approach indicates sufficient inherent error resilience for further and more aggressive approximate processing approaches.

Regarding resource utilization and power dissipation with the proposed approach, the *Conv* processing units have a 43.24% reduction of BRAM, and a 12.35% of improvement in energy efficiency over the standard floating-point implementation. However, the proposed approach does not reuse the available floating-point operator cores instantiated from other computational sections (see **Tab.** 4). Therefore, the logic required for the dot-product must be implemented, which is reflected as additional utilization of LUT and FF resources. The experimental results of the design exploration are summarized in **Tab.** 9. The platform implementations are summarized in **Tab.** 10, and their power dissipation breakdowns are presented in **Fig.** 22.

VI. CONCLUSIONS

In this work, we accelerate SbS neural networks with a dot-product functional unit based on approximate computing that combines the advantages of custom floating-point and logarithmic representations. This approach reduces computational latency, memory footprint, and power dissipation while preserving classification accuracy. For output quality monitoring, we applied noise tolerance plots as an intuitive visual measure to provide insights into the accuracy degradation of SbS networks under different approximate processing effects. This plot revels inherent error resilience, hence, the possibilities for approximate processing.

We demonstrate our approach using a design exploration flow on a Xilinx Zynq-7020 with a deployment of SbS network for the MNIST classification task. This implementation achieves up to $20.5\times$ latency enhancement, $8\times$ weight memory footprint reduction, and 12.35% of energy efficiency improvement over the standard floating-point hardware implementation, and incurs in less than 0.5% of accuracy degradation. Furthermore, with a noise amplitude of 50% added on top of the input images, the SbS network presents an accuracy degradation of less than 5%. As output quality monitor, the resulting noise tolerance plots demonstrate a sufficient QoR for minimal impact on the overall accuracy of the neural network under the effects of the proposed approximation technique. These results suggest available room for further and more aggressive approximate processing approaches.

TABLE 9. Experimental results.

Dot-product implementation	PU	Post-implementation resource utilization				Power (mW)	Latency		Accuracy (%)[e]	
		LUT	FF	DSP	BRAM 18K		T_{SC} (ms)	Gain[d]	Noise 0%	50%
Standard floating-point computation[a]	Conv	2,765	4,366	19	37	89	3.18	10.7x	98.98	98.63
	FC	2,649	4,189	8	9	66				
Hybrid custom floating-point approx[b]	Conv	3,139	4,850	19	25	82	1.67	20.5x	98.97	98.47
	FC	3,265	5,188	8	9	66				
Hybrid logarithmic approximation[c]	Conv	3,086	4,804	19	21	78	1.67	20.5x	98.84	95.22
	FC	3,046	4,873	8	8	66				

[a] Reference with standard floating-point arithmetic (IEEE 754).
[b] Synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[c] Synaptic weight with number representation composed of 4-bit exponent.
[d] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[e] Accuracy on 10,000 image test set with 1000 spikes.

TABLE 10. Platform implementations.

Platform implementation	Post-implementation resource utilization				Power (W)	Clock (MHz)	Latency		Accuracy (%)[f]
	LUT	FF	DSP	BRAM 18K			T_{SC} (ms)	Gain[e]	
Ref. [15][a]	42,740	57,118	49	92	2.519	250	4.65	7.4x	99.02
This work (standard floating-point computation)[b]	39,514	56,036	82	180	2.420	200	3.18	10.7x	98.98
This work (hybrid custom floating-point approx)[c]	42,021	58,759	82	156	2.369	200	1.67	20.5x	98.97
This work (hybrid logarithmic approximation)[d]	41,060	57,862	82	148	2.324	200	1.67	20.5x	98.84

[a] Reference architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754).
[b] Reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754).
[c] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent and 1-bit mantissa.
[d] Proposed architecture with specialized heterogeneous PUs using synaptic weight with number representation composed of 4-bit exponent.
[e] Acceleration with respect to the computation on embedded CPU (ARM Cortex-A9 at 666 MHz) with latency $T_{SC} = 34.28ms$.
[f] Accuracy on 10,000 image test set with 1000 spikes.

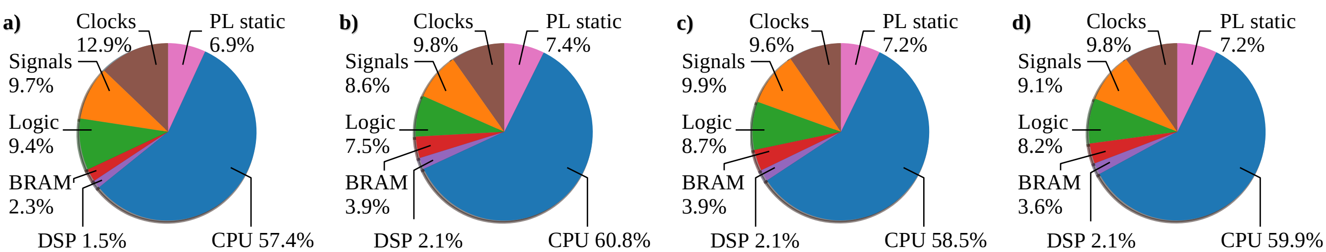

FIGURE 22. Power dissipation breakdown of platform implementations, (a) Ref. [15] architecture with homogeneous AUs using standard floating-point arithmetic (IEEE 754), (b) reference architecture with specialized heterogeneous PUs using standard floating-point arithmetic (IEEE 754), (c) proposed architecture with hybrid custom floating-point approximation, and (d) proposed architecture with hybrid logarithmic approximation.

In summary, 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] M. D. McDonnell and L. M. Ward, "The benefits of noise in neural systems: bridging theory and experiment," *Nature Reviews Neuroscience*, vol. 12, no. 7, pp. 415–425, 2011.
[8] J. Dapello, T. Marques, M. Schrimpf, F. Geiger, D. D. Cox, and J. J. DiCarlo, "Simulating a primary visual cortex at the front of cnns improves robustness to image perturbations," *bioRxiv*, 2020.
[9] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.
[10] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla,

N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, Oct 2015.

[11] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: Opportunities and challenges," *Frontiers in neuroscience*, vol. 12, 2018.

[12] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.

[13] K. Amunts, A. C. Knoll, T. Lippert, C. M. Pennartz, P. Ryvlin, A. Destexhe, V. K. Jirsa, E. DâĂŹAngelo, and J. G. Bjaalie, "The human brain project – synergy between neuroscience, computing, informatics, and brain-inspired technologies," *PLoS biology*, vol. 17, no. 7, p. e3000344, 2019.

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

[15] 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.

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

[17] K. Roy, A. Jaiswal, and P. Panda, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, no. 7784, pp. 607–617, 2019.

[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] A. R. Young, M. E. Dean, J. S. Plank, and G. S. Rose, "A review of spiking neuromorphic hardware communication systems," *IEEE Access*, vol. 7, pp. 135 606–135 620, 2019.

[20] D. Rotermund and K. R. Pawelzik, "Biologically plausible learning in a deep recurrent spiking network," *bioRxiv*, 2019.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," *Advances in neural information processing systems*, vol. 2, pp. 598–605, 1989.

[39] B. Hassibi and D. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," *Advances in neural information processing systems*, vol. 5, pp. 164–171, 1992.

[40] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[41] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[42] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.
[55] 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.
[56] 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.
[57] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.
[58] U. Xilinx, "Zynq-7000 all programmable soc: Technical reference manual," 2015.
[59] J. Hrica, "Floating-point design with vivado hls," *Xilinx Application Note*, 2012.

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 started his scientific career as a chemical technical assistant in 1992 and received a pre-diploma in electrical engineering at the Hochschule Bremen (City University for Applied Science) in 1996. In 2002, he finished his studies of physics at the University of Bremen with a diploma (specialization in neuroscience and solid state physics). In 2007 he received his PhD "Extraction of information from the dynamical activities of neural networks". Among other neuroscience projects, he participated in several project in the field of neuroprosthetics like the German-Israeli joint project "Models and Experiments towards Adaptive Control of Motor Prostheses" (METACOMP), the research focus Neurotechnology at the University of Bremen, and the Creative Unit "I-See: The artificial eye – chronic wireless interface to the visual cortex". In the BMBF project KALOMED, where the goal was to design a fully wireless recording system that can be implanted under the skull of an user, he worked as project organizer and hardware/ software/ firmware designer as well as data miner. He will be the co-organizer of the upcoming Era-Net Neuron (a joint Canadian / EU project) for the development of advanced techniques in the field of visual cortex prosthesis. Beside his research in the field of neuroprosthetics, he is keenly interested in information processing using spiking neuronal networks.

KLAUS R. PAWELZIK received his PhD in the field of Nonlinear Dynamics in 1990. From 1991 till March 1998 he was research assistant at several well-known institutes in Germany and the US. Since April 1998 he is a tenured professor for Theoretical Physics and Theoretical Biology at the University of Bremen. He works mainly on topics in Theoretical Neuroscience, but also on problems in Neuro-technology and studies models of other complex adaptive systems. His many publications underline his expertise in these fields. Currently he is the director of the Center of Cognitive Sciences at the University of Bremen and has raised a number of third-party funds, among them several in the field of Neurotechnology. There he recently filed a patent with the title "Artificial neural network data processing apparatus and data processing method".

ALBERTO GARCIA-ORTIZ obtained the diploma degree in Telecommunication Systems from the Polytechnic University of Valencia (Spain) in 1998. After working for two years at Newlogic in Austria, he started the Ph.D. at the Institute of Microelectronic Systems, Darmstadt University of Technology, Germany. In 2003, he received from the Department of Electrical Engineering and Information Technology of the university the Ph.D. degree with "summa cum laude." From 2003 to 2005, he worked as a Senior Hardware Design Engineer at IBM Deutschland Development and Research in Böblingen. After that he joint the start-up AnaFocus (Spain), where he was responsible for the design and integration of AnaFocus" next generation Vision Systems-on-Chip. He is currently full professor for the chair of integrated digital systems at the university of Bremen. Dr. Garcia-Ortiz received the "Outstanding dissertation award" in 2004 from the European Design and Automation Association. In 2005, he received from IBM an innovation award for contributions to leakage estimation. Two patents are issued with that work. He serves as editor of JOLPE and is reviewer of several conferences, journals, and European projects.
His interests include low-power design and estimation, communication-centric design, SoC integration, and variations-aware design.