.

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

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

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Spike-by-Spike Neural Networks using Custom Floating-Point and Logarithm Computation 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 Computational complexity and memory footprint are recognized as the fundamental constraints in artificial neural networks (ANNs) for deployment in embedded systems. Recent implementations using logarithmic quantization have demonstrated great potential to overcome the aforementioned limitations; however, this solution is often accompanied by quantization-aware training methods that, in some cases, are problematic or even inaccessible, particularly in emerging spiking neural network (SNN) algorithms. As an alternative, optimal hardware implementations may require the use of custom floating-point computation to improve performance while preserving accuracy of the neural network models.

In this publication, we present a hardware architecture for non-quantized SbS network simulations based on an optimized dot-product using custom floating-point and logarithmic number representation. As main characteristics, the proposed dot-product performs element-wise multiplication by adding integer exponents and accumulating denormalized products which decrease computation latency, while the synaptic weight vector uses a reduced number representation which reduces memory footprint and facilitates on-chip storage, the neuron vector uses standard floating-point number representation which preserves inference accuracy.

The proposed hardware architecture is demonstrated with a design exploration using high-level synthesis and a Xilinx FPGA. As a benchmark, we deploy a classification task of NMIST dataset with an accuracy of 99.3% and a CPU latency of 34.3 ms per spiking cycle. As a result, the proposed architecture with synaptic representation using 5-bit custom floating-point, and 4-bit logarithmic achieve 20.5x latency enhancement with less than 0.5% of accuracy degradation. Moreover, inserting positive additive uniformly distributed noise at 50% amplitude to the input images, the SbS network simulation presents an accuracy degradation of less than 1% using custom floating-point, and less than 5% using logarithmic computation.

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

I. INTRODUCTION

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

Historically, ANNs can be classified into three different generations [1]: The first one is represented by the classical McCulloch and Pitts neuron model; the second one is represented by more complex continuos-output architectures as Multi-Layer Perceptrons and Convolutional Neural Networks

(CNN); while the third generation is represented by Spiking Neural Networks (SNNs). They differ fundamentally in the neural computation and the neural coding strategy: while the first generation uses discrete binary values as outputs, the second generation uses continuous activation functions where learning can be easily implemented. In contrast, the third generation, uses spikes as means for information exchange between groups of neurons. This strategy mimics how real neurons interact through short pulses (the so called action potentials).

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

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

To address the aforementioned problems, in [6] we presented a scalable hardware-software framework for SbS NN models targeting embedded systems applications. This framework deploys fully customizable SbS models allowing arbitrary dimensions, topologies, and acceleration configuration, ideal for experimentation and research on devices with limited resources, particularly in the field of IoT and Edge computing. However, the hardware architecture operates with standard floating-point computation having elevated inference latency, memory footprint.

In this paper, we present architectural optimizations based on the use of custom floating-point and logarithmic computation to reduce inference latency and memory footprint while preserving accuracy and noise robustness targeting non-quantized network models. Furthermore, the proposed hardware architecture consists of a collection of heterogeneous processing units (PU), and exploiting the principle of parallelization of SbS networks.

The proposed hardware architecture exploits the available resources of the target platform: from pure embedded software on a single Central Processing Unit (CPU), scaling to a variable number of PUs for a low-cost or high-capacity FPGA. This design provides a configurable solution able to match different FPGA characteristics.

In this paper we present architectural optimizations using custom floating-point and logarithmic computation in a scalable design. We develop a hardware dot-product module for custom floating-point and logarithmic computation; this provides a good trade-off between resource utilization and throughput. These optimizations and methods are intended to be suitable for other ANNs, do not require model retraining, and can achieve comparable inference accuracy as the 32-bit floating-point counterpart.

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

II. RELATED WORK

In the literature we find plenty of hardware architectures dedicated to the second generation of NN implemented in Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) designs [7], [8]. However the related work on SNN is much reduced. Recently, some state of the art survey on hardware architectures for SNN have been reported [1], [4]. In particular, Nassim Abderrahmane et al. briefly describe and compare some recent implementations of ASIC and FPGA where only two are suitable for embedded systems. As a typical example of the current state of the art, Furber et al., presents SpiNNaker [2], aiming to simulate very large SNNs in real-time. It is composed of 48 chips containing a shared memory and 18 ARM cores with small local memory each processor. The main feature of SpiNNaker are the support for several neuron models, synaptic plasticity rules, incremental learning capabilities and efficient communication system. This architecture is suitable for neuroscience research but not for embedded applications. Further on, in a previous research Rotermund et al., demonstrated the feasibility of a neuromorphic SbS IP in a Xilinx Virtex 6 FPGA [9]. It provides a massively parallel architecture, optimized for memory access and suitable for ASIC implementations. However, this design is considerably resource-demanding to be deployed as a full and functional SbS network in the current embedded technology.

Beside the actual architectures, researches have also identified design methodologies as a critical problem for the efficient development of SNN [1]. For example, Nassim Abderrahmane et al., develop a behavioral level simulator for neuromorphic hardware architectural exploration named NAXT, capable to reduce the number of spikes while keeping the neuron's model resulting in lower power consumption. This work provides a great exploration of SNN for different network topologies and computation approaches on NAXT. However, this simulator presents 62% of accuracy on MNIST classification task, **(TODO: Add more content)**

.

In our previous work [6], as a design exploration framework for SbS neural networks embedded systems and IoT devices, we reported 99% of accuracy on MNIST classification task with positive additive uniformly distributed noise at 50% of amplitude.

III. BACKGROUND

A. SPIKE-BY-SPIKE NEURAL NETWORKS

As a generative model [3], the SbS model iteratively finds an estimate of its input probability distribution $p(s)$ (i.e. the probability of input node s to stochastically send a spike) by its latent variables via $r(s) = \sum_i h(i)W(s|i)$. An inference population sees only the spikes s_t (i.e. the index identifying the input neuron s which generated that spike at time t) produced by its input neurons, not the underlying input probability distribution $p(s)$ itself. By counting the spikes arriving at a group of SbS neurons, $p(s)$ is estimated by $\hat{p}(s) = 1/T \sum_t \delta_{s,s^t}$ after T spikes have been observed in total. The goal is to generate an internal representation $r(s)$ from the string of incoming spikes s_t such that the negative logarithm of the likelihood $L = C - \sum_\mu \sum_s \hat{p}_\mu(s) log\,(r_\mu(s))$ 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

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

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

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

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

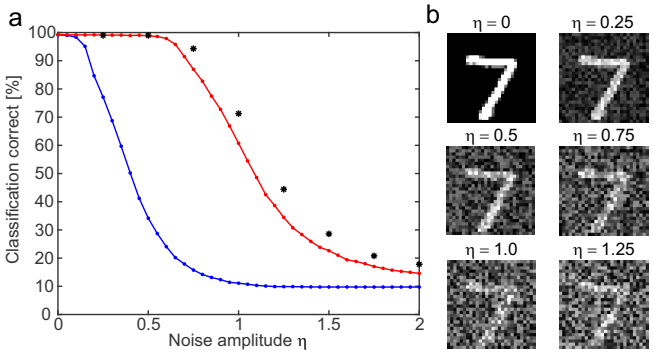

FIGURE 1. (a) Performance classification of SbS NN versus equivalent CNN, and (b) Example of the first pattern in the MNIST test data set with different amounts of noise.

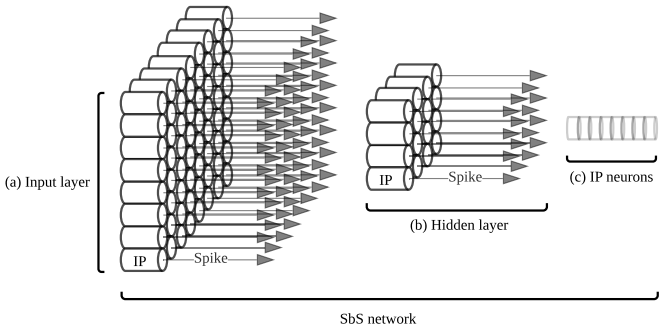

FIGURE 2. (a) Illustrates an input layer with a massive amount of IPs operating as independent computational entities. (b) Illustrate a hidden layer with an arbitrary amount of IPs as independent computational entities. (c) Illustrates a set of neurons grouped in an IP.

An example of the robustness of SbS is presented in **Fig.** 1. It compares the classification performance of a SbS network and a tensor flow network, with the same amount of neurons per layer as well as the same layer structure. We trained on MNIST training data without noise (see [5] for details). It shows the correctness for the MNIST test data set with its 10000 patterns in dependency of the noise level for positive additive uniformly distributed noise. The blue curve shows the performance for the tensor flow network, while the red curve shows the performance for the SbS network with 1200 spikes per inference population. Beginning with a noise level of 0.1, the respective performances are different with a p - level of at least 10^{-6} (tested with the Fisher exact test). Increasing the number of spikes per SbS population to 6000 (performance values shown as black stars), shows that more spike can improve the performance under noise even more.

B. PARALLELIZATION IN SBS NETWORKS

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 [10]. Technically, each IP is an independent computational entity (see **Fig.** 2), this allows to design specialized hardware architectures that can be massively parallelized.

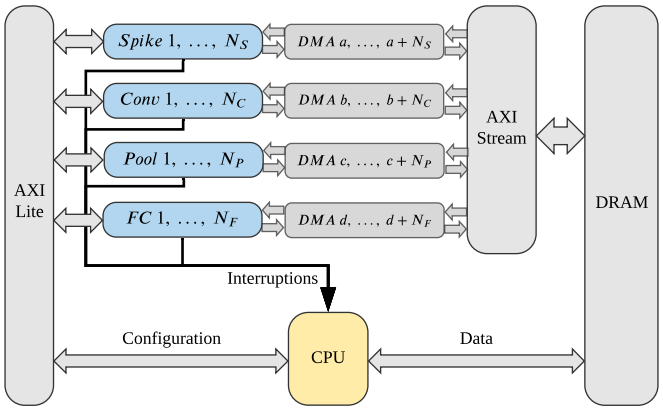

FIGURE 3. System overview of the proposed architecture with scalable number of heterogeneous PUs: *Spike*, *Conv*, *Pool*, and *FC*

IV. SYSTEM DESIGN

A. HARDWARE ARCHITECTURE

As a hardware and software design, the proposed architecture is composed mainly by a CPU and a collection of four heterogeneous PUs: *Spike* (input layer), *Conv* (convolution), *Pool* (pooling), and *FC* (fully connected). Each PU is connected through an AXI-Lite interface for configuration of operational mode, 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 on task or transaction completion which is handled by the software drivers 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.** 3). The dedicated PUs for *Spike*, *Conv*, and *FC* use custom floating-point, and logarithmic computation, while the *Pool* uses standard floating-point.

The PUs are written in C using Vivado HLS (High-Level Synthesis) tool. In this publication we focus on the optimizations achieved by using reduced custom floating-point and logarithmic computation in the processing unit dedicated for convolution layers.

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 parameter 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. In principle, 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 $\sum_{j=0}^{N-1} W(s_t|j) = 1$ [5]; hence, W employs only positive normalized real numbers. With this, 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; therefore, 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.** 4. 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 $h_\mu^{new} \in \mathbb{R}^N$. The process repeats from the first to the fifth stage for H number of loops. Finally, the S^{new} is dispatched.

The computation of the update dynamics [see **Fig.** 4(d)] operates in two modular stages: *dot-product* and *neuron update*. First, the *dot-product* module calculates $\sum_{j=0}^{N-1} h_\mu(j)W(s_t|j)$, while storing each element-wise multiplication 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. In the following section, we focus on an optimized design for this dot-product module to improve performance while preserving inference accuracy.

C. DOT-PRODUCT HARDWARE MODULE

This module is part of an application-specific architecture optimized to calculate dot-product of arbitrary length, see **Eq.** (2). In this section, we present three pipelined hardware modules using standard floating-point, custom floating-point, and logarithmic computation, respectively.

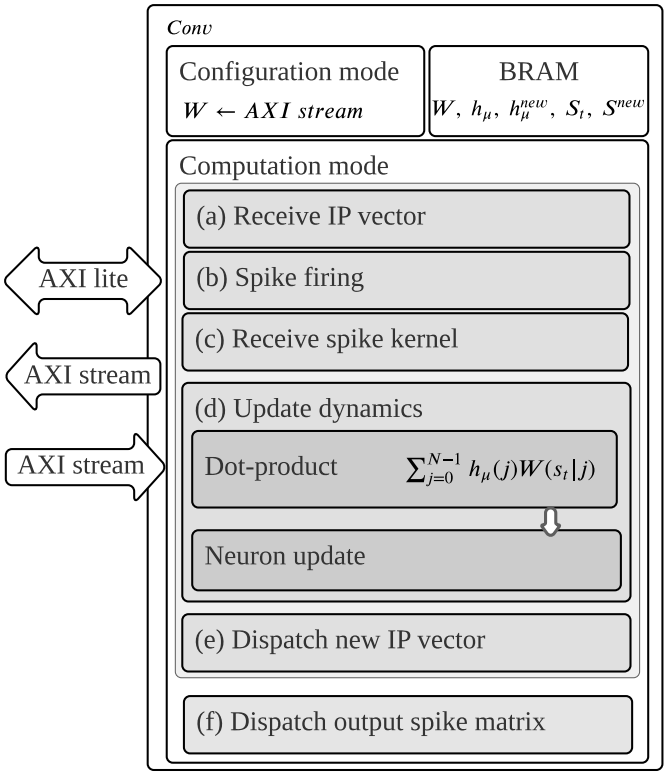

FIGURE 4. The *Conv* processing unit and its six stages: (a) receive IP vector, (b) spike firing, (c) receive spike kernel, (d) update dynamics, (e) dispatch new IP vector, (f) dispatch output spike matrix.

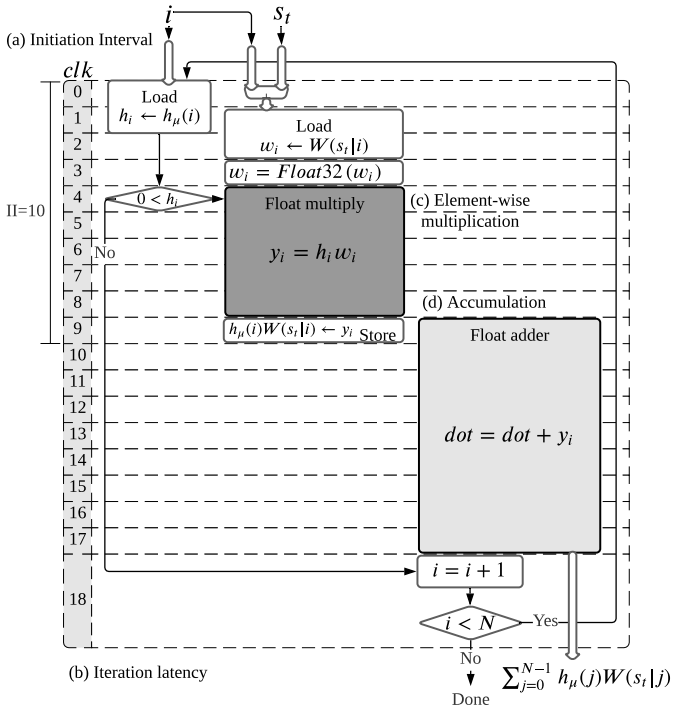

FIGURE 5. Dot-product hardware module using standard floating-point computation. (a) Illustrates the iteration interval of 10 clock cycles. (b) Illustrates the iteration latency of 19 clock cycles. (c) Illustrates the element-wise multiplication block in dark-gray. (d) Illustrates the accumulation block in light-gray.

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

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.** 5, 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 element-wise multiplication [**Fig.** 5(c)] and accumulation [**Fig.** 5(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.** 5(a)], and IL is the iteration latency [**Fig.** 5(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.** 5(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 when needed. Moreover, in the case of floating-point addition [**Fig.** 5(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 necessary. 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.

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

The hardware module to calculate dot-product using custom floating-point computation is shown in **Fig.** 6. In this design, h_μ uses standard floating-point number representation, and $W(s)$ uses a positive custom floating-point number representation where the exponent and mantissa bit width are design parameters. A slight modification in this design yields a hardware module which computes the dot-product using logarithmic computation, this is shown in **Fig.** 7. In this design, $W(s)$ uses a logarithmic number representation where the exponent bit width is a design parameter.

The computation of these 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: *element-wise multiplication* and *accumulation*. Where *element-wise multiplication* is executed either in custom floating-point or logarithmic computation described below.

- Element-wise multiplication.
  - Custom floating-point. As shown in **Fig.** 6(c) in dark-gray, the element-wise multiplication is obtained by adding the exponents and multiplying the mantissa 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 element-wise multiplication has a latency of 5 clock cycles.
  - Logarithmic. As shown in **Fig.** 7(c) in dark-gray, the content values of $W(s)$ are represented in the logarithmic domain, and h_μ in standard floating-point. Hence, the element-wise multiplication is obtained by adding $W(s|i)$ to the exponent of $h_\mu(i)$. In this design the element-wise multiplication has a latency of one clock cycle.
- Accumulation. As shown both **Fig.** 6(d) and **Fig.** 7(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. And then, this denormalized representation is accumulated to obtain the magnitude of the dot-product.

The element-wise multiplication and accumulation is an iterative process, 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 the same initiation interval of two clock cycles.

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

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

- 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 test, 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 resulting dot-product magnitude in a loop until it is in the form of a normalized mantissa where the iteration count represents the negative exponent of the dot-product, each iteration takes one clock cycle.

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

The proposed architecture using custom floating-point and logarithmic computation 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 denormalized representation and one final normalization.

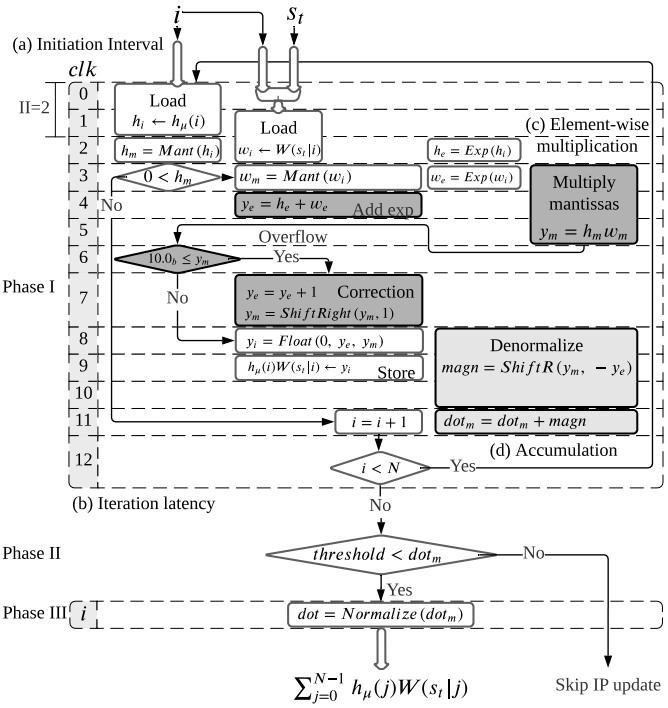

FIGURE 6. Dot-product hardware module using custom floating-point computation. (a) Illustrates the iteration interval of 2 clock cycles. (b) Illustrates the iteration latency of 13 clock cycles. (c) Illustrates the element-wise multiplication blocks in dark-gray. (d) Illustrates the accumulation blocks 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 [11]. 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 an SbS network structure for MNIST classification task as shown in **Fig.** 8. The training of the

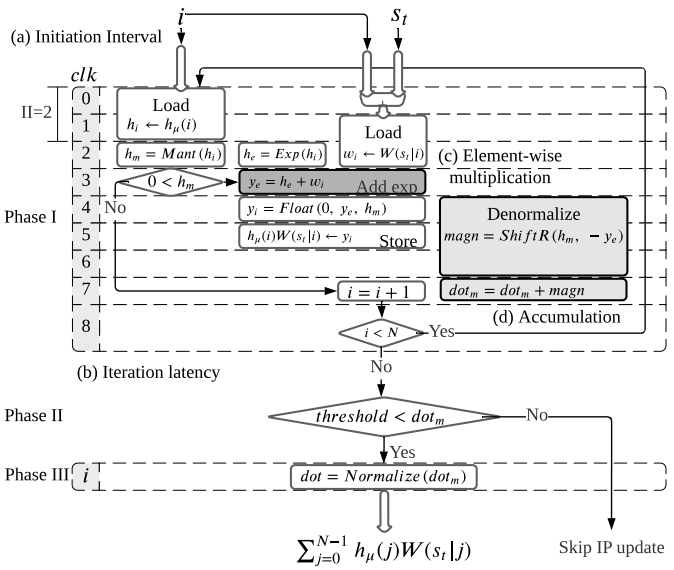

FIGURE 7. Dot-product hardware module using logarithmic computation. (a) Illustrates the iteration interval of 2 clock cycles. (b) Illustrates the iteration latency of 9 clock cycles. (c) Illustrates the element-wise multiplication block in dark-gray. (d) Illustrates the accumulation blocks in light-gray.

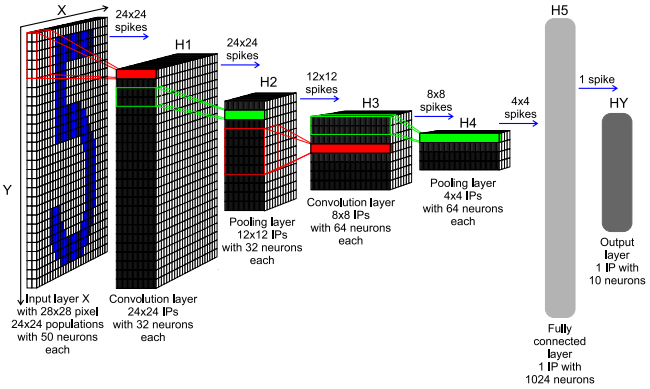

FIGURE 8. SbS network structure for MNIST classification task. Input X: Input layer with 28×28 normalization modules for 28×28 input pixel. From this layer spikes are send to layer $H1$. $H1$: Convolution layer $H1$ with 24×24 IPs with 32 neurons each. Every IP processes the spikes from 5×5 spatial patches 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.

SbS model is performed in Matlab and the resulting synaptic weight matrices are deployed on the embedded system. The SbS network is built as a sequential model using the API from the SbS embedded software framework [6], where the computation of the network is distributed among the hardware processing units and the CPU.

For the evaluation of our approach, we elaborate a design exploration reviewing the computational latency, inference accuracy, noise robustness, resource utilization, and power dissipation. First, we benchmark the performance of SbS network simulation using standard floating-point computation on CPU, and then hardware processing units. Afterwards, we evaluate our dot-product architecture addressing a design exploration using custom floating-point, and then logarithmic computation. Finally, we present a comparison table 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 520 mW.

The SbS network computation on the CPU achieves a latency of 34.279 ms 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.** 9 respectively.

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

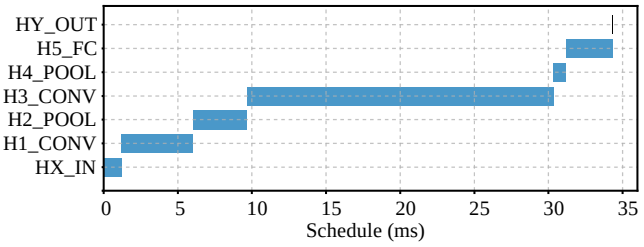

FIGURE 9. Computation on CPU.

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.** 10. 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.

In this deployment, it is distributed the computation workload of *H2_POOL* and *H3_CONV* among two PUs each one, since these are the heaviest pooling and convolution layers

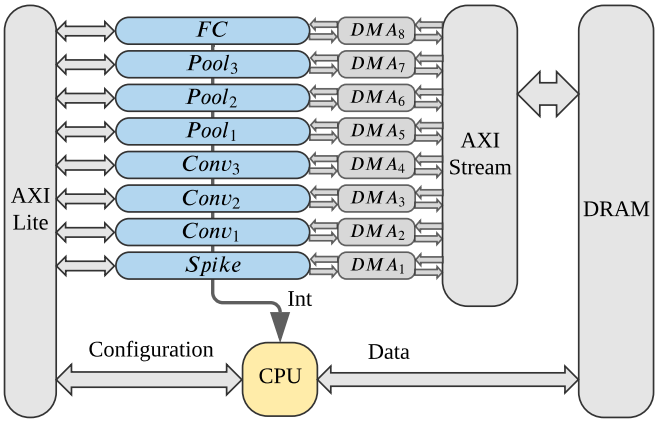

FIGURE 10. System overview of the proposed architecture with 8 processing units.

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.** 11.

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

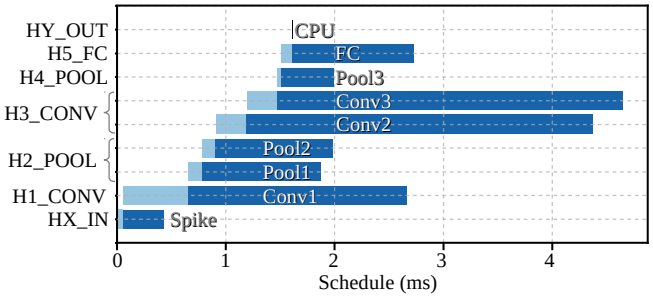

FIGURE 11. Performance of processing units using standard floating-point computation.

In the computation schedule, the following terms are defined: $t_s(n)$ as the start time of the 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. 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).

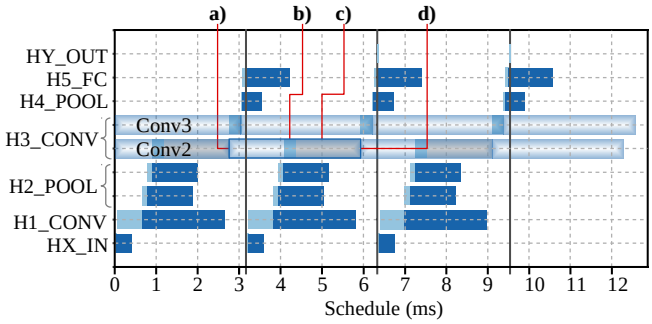

FIGURE 12. Performance bottleneck of cyclic computation on processing units using standard floating-point. a) Illustrates the starting of t_{PU} of *Conv2* on a previous computation cycle. b) Illustrates t_{CPU} of *Conv2* on the current computation cycle. c) Illustrates the CPU waiting time (in gray 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).

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

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

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

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

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.** 12. 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. 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. Furthermore, the noise robustness is measured using input patterns with positive additive equidistributed random noise up to 55% of amplitude as shown in **Fig.** 13. The post-implementation resource utilization and power dissipation are shown in **Tab.** 3 and **Tab.** 4, respectively.

Each *Conv* PU instantiates a BRAM 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 COMPUTATION

In this section, we address a design exploration to evaluate our approach for SbS neural network simulation using custom floating-point and logarithmic computation. First, we examine each synaptic weight matrix in order to obtain 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, noise robustness, 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 each synaptic weight matrix as shown in **Fig.** 11. Since $0 \leq W(s_t|j) \leq 1$ 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).

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

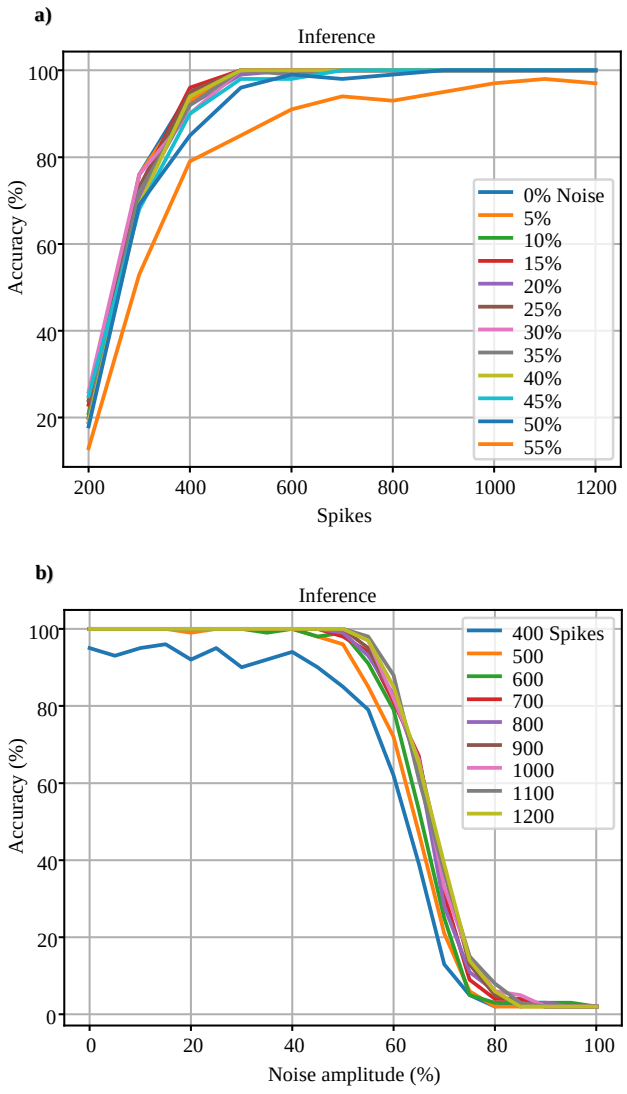

FIGURE 13. Benchmark of accuracy and noise robustness of SbS network simulation on hardware PU using standard floating-point computation on 100 images. (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

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

Applying **Eq.** (10) and **Eq.** (11) to the given SbS network, we obtain -13 as the minimum exponent value of the synaptic weights, and 4-bit needed for its absolute binary representation. The mantissa bit width is a parameter that needs to be determined/tuned by the designer, to perform trade-off between computation accuracy and synaptic memory footprint. In this publication we present a case study with 1-bit mantissa corresponding to the custom floating-point in the next section.

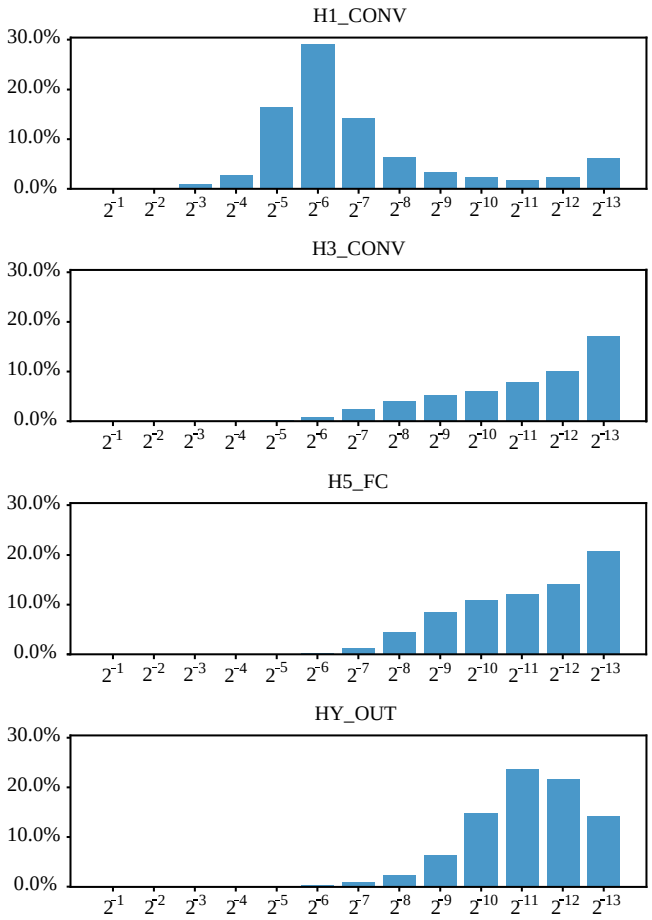

FIGURE 14. $\log_2$-histogram of each synaptic weight matrix showing the percentage of matrix elements with given integer exponent.

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

For this design exploration, we use a custom floating-point representation composed of 4-bit exponent and 1-bit mantissa for the synaptic weight matrix on the proposed dot-product architecture. In this case, each *Conv* PU instantiates a BRAM 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, this does not instantiate BRAM 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.** 15.

With a reduction from 32-bit ot 5-bit in all the synaptic weight matrices, this design exploration achieves a maximum hardware PU latency of 1.309 ms according to **Eq.** (7), and a CPU latency of 1.673 ms. Therefore, applying **Eq.** (8), we obtain a latency of 1.673 ms per spike cycle as shown in **Fig.** 15. In this case, the cyclic bottleneck is in the CPU performance.

This deployment achieves an accuracy of 98.97% correct classification on the 10,000 image test set at 1000 spikes. Furthermore, the noise robustness is shown in **Fig.** 16. The post-implementation resource utilization and power dissipation are shown in **Tab.** 6 and **Tab.** 7, respectively.

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

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

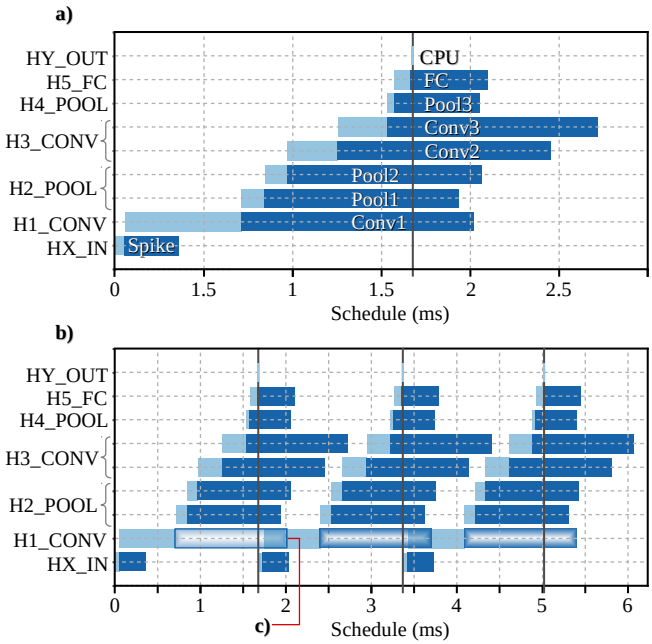

FIGURE 15. Performance on processing units using custom floating-point computation. a) Illustrates computation schedule. b) Illustrates cyclic computation schedule. c) Illustrates the performance of *Conv2* from a previous computation cycle during the preprocessing of *H1_CONV* on the current computation cycle without bottleneck.

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

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

3) Design exploration for dot-product using logarithmic computation

For this design exploration, we use a 4-bit integer exponent for logarithmic representation of the synaptic weight matrix. In this case, each *Conv* processing unit implements the

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

PU	Power (mW)
Conv	82
FC	66

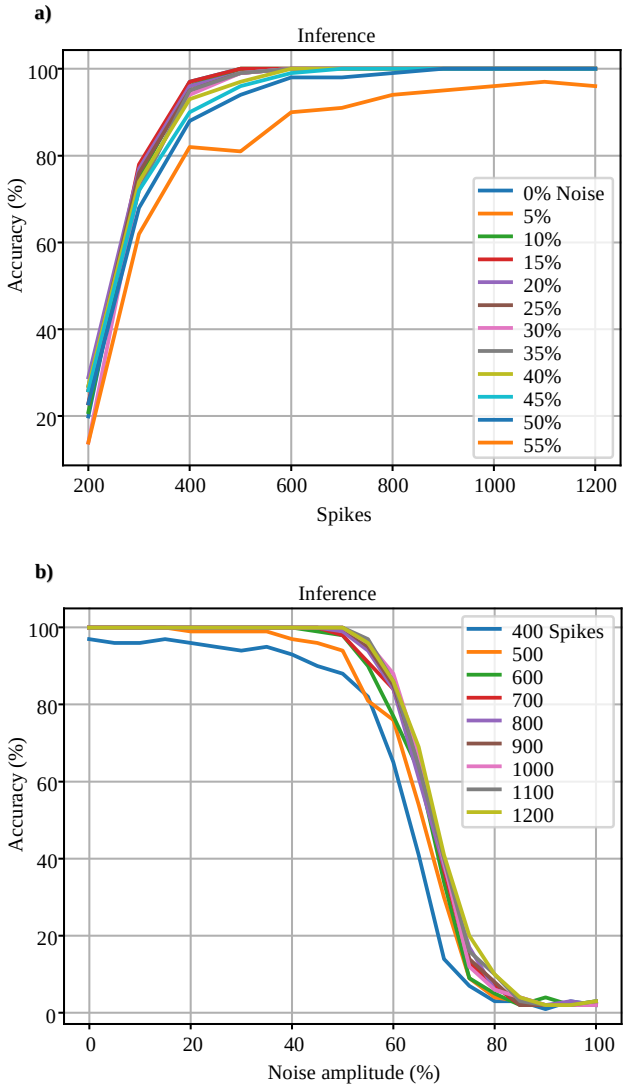

FIGURE 16. Accuracy and noise robustness of SbS network simulation on hardware PU using custom floating-point computation on 100 images. (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

proposed dot-product architecture and a BRAM 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.** 17.

With a reduction from 32-bit ot 4-bit in all the synaptic weight matrices, this design exploration achieves a maximum

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

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

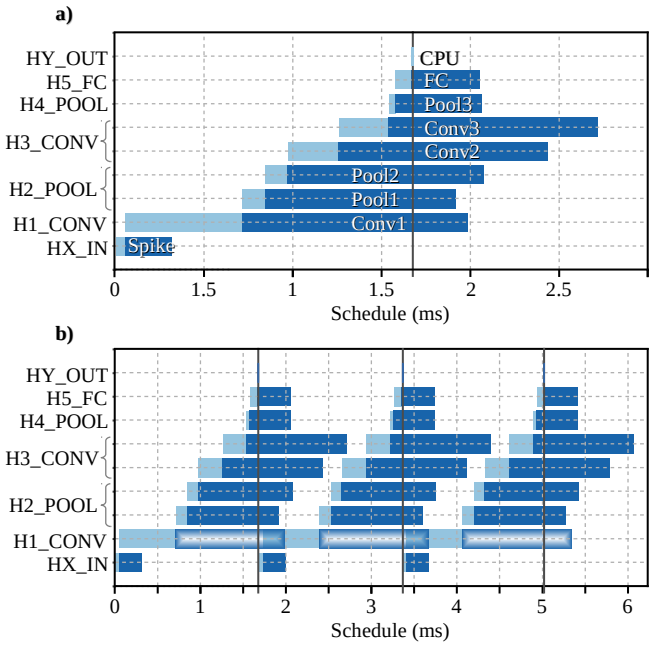

FIGURE 17. Performance of processing units using logarithmic computation. a) Illustrates computation schedule. b) Illustrates cyclic computation schedule.

hardware PU latency of 1.271 ms according to **Eq.** (7), and a CPU latency of 1.673 ms. Therefore, applying **Eq.** (8), we obtain a latency of 1.673 ms per spike cycle as shown in **Fig.** 17. In this case, the cyclic bottleneck is in the CPU performance.

This deployment achieves an accuracy of 98.84% correct classification on the 10,000 image test set at 1000 spikes. Furthermore, the noise robustness is shown in **Fig.** 16. 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

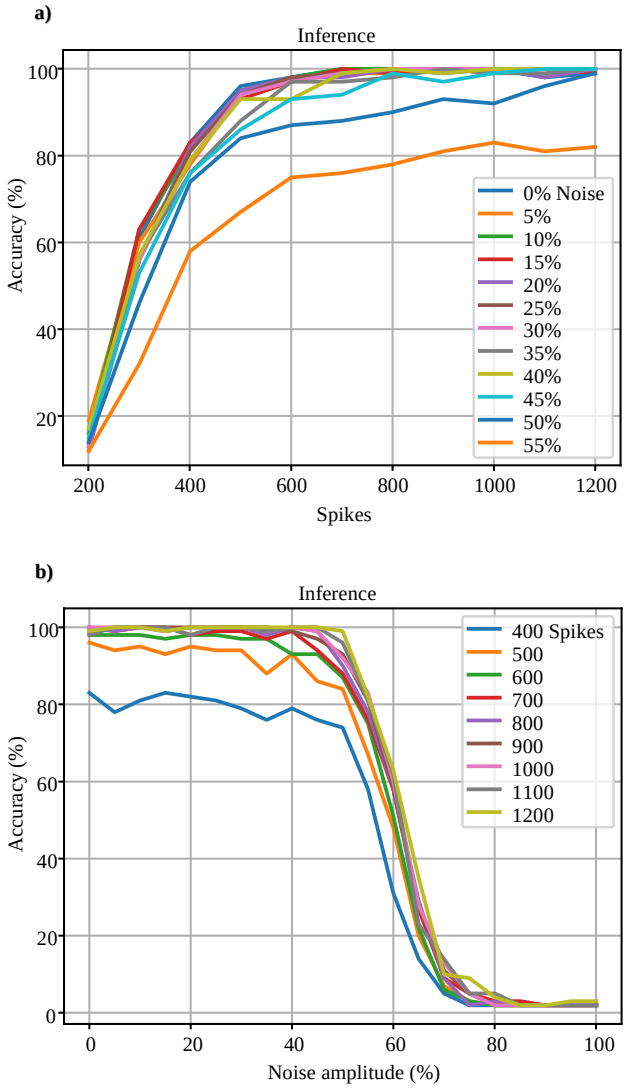

FIGURE 18. Accuracy and noise robustness of SbS network simulation on hardware PU using logarithmic computation on 100 images. (a) Illustrates the accuracy vs number of spikes at given noise amplitudes. (b) Illustrates the accuracy vs noise amplitude at given number of spikes.

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 spike cycle $T_{SC} = 34.279ms$. As a second benchmark, the network simulation on processing units using 8-bit custom floating-point for synaptic storage and standard floating-point computation achieves an accuracy of 98.98% with a latency $T_{SC} = 3.183ms$. As the benchmark result, we get a 10.77x latency enhancement and an accuracy degradation of 0.32%.

As a demonstration of the proposed architecture, the SbS network simulation on hardware PUs with synaptic representation using 5-bit custom floating-point and 4-bit logarithmic achieve 20.49x latency enhancement and accuracy of 98.97% and 98.84%, respectively. As a result, we accuracy degradation of 0.33% and 0.46%, respectively. Moreover, in the case of adding 50% of noise amplitude to the input images, the SbS network simulation presents an accuracy degradation of 0.67% and 4.08%, respectively. The experimental results of the design exploration are summarized in **Tab.** 11.

VI. CONCLUSIONS

In this publication, we present a hardware architecture for non-quantized SbS network simulations based on optimized dot-product computation using custom floating-point and logarithmic number representation. To reduce latency and power dissipation, the proposed architecture performs element-wise multiplication by adding integer exponents, and denormalized product accumulation. To reduce memory footprint, the synaptic vector uses a precise custom floating-point or logarithmic representation. To preserve accuracy and noise robustness, the neuron vector uses non-quantized standard floating-point representation.

The proposed hardware architecture is demonstrated on a Xilinx Zynq-7020 with a deployment of NMIST classification task. In this architecture, the SbS network computation on hardware processing units using 5-bit custom floating-point and 4-bit logarithmic achieve 20.49x latency enhancement and accuracy of 98.97% and 98.84%, respectively. These results represent less than 0.5% of accuracy degradation. Moreover, in the case of adding 50% amplitude of positive additive uniformly distributed noise to the input images, the SbS network simulation presents an accuracy degradation of less than 1% on the custom floating-point, and less than 5% on the logarithmic computation.

In conclusion, the proposed dot-product architecture preserves accuracy and noise robustness while reducing latency, memory footprint, and power dissipation of non-quantized SbS network computation.

ACKNOWLEDGMENTS

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

REFERENCES

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

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

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

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 dot-product computation, each entry 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.

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

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

[6] 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.

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

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

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

[10] ——, "Massively parallel fpga hardware for spike-by-spike networks," *bioRxiv*, 2019. [Online]. Available: https://www.biorxiv.org/content/early/2019/06/14/500280

[11] U. Xilinx, "Zynq-7000 all programmable soc: Technical reference manual," 2015.

DAVID ROTERMUND

KLAUS R. PAWELZIK

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.

ALBERTO GARCIA-ORTIZ