

Efficient Hardware Acceleration of Sparsely Active Convolutional Spiking Neural Networks

Jan Sommer, M. Akif Özkan, Oliver Keszocze *Member, IEEE*, Jürgen Teich, *Fellow, IEEE*

Abstract—Spiking Neural Networks (SNNs) compute in an event-based manner to achieve a more efficient computation than standard Neural Networks. In SNNs, neuronal outputs (i.e. activations) are not encoded with real-valued activations but with sequences of binary spikes. The motivation of using SNNs over conventional neural networks is rooted in the special computational aspects of spike based processing, especially the very high degree of sparsity of neural output activations. Well established architectures for conventional Convolutional Neural Networks (CNNs) feature large spatial arrays of Processing Elements (PEs) that remain highly underutilized in the face of activation sparsity. We propose a novel architecture that is optimized for the processing of Convolutional SNNs (CSNNs) that feature a high degree of activation sparsity. In our architecture, the main strategy is to use less but highly utilized PEs. The PE array used to perform the convolution is only as large as the kernel size, allowing all PEs to be active as long as there are spikes to process. This constant flow of spikes is ensured by compressing the feature maps (i.e. the activations) into queues that can then be processed spike by spike. This compression is performed in run-time using dedicated circuitry, leading to a self-timed scheduling. This allows the processing time to scale directly with the number of spikes. A novel memory organization scheme called *memory interlacing* is used to efficiently store and retrieve the membrane potentials of the individual neurons using multiple small parallel on-chip RAMs. Each RAM is hardwired to its PE, reducing switching circuitry and allowing RAMs to be located in close proximity to the respective PE. We implemented the proposed architecture on an FPGA and achieved a significant speedup compared to other implementations while needing less hardware resources and maintaining a lower energy consumption.

Index Terms—Spiking Convolutional Neural Networks (SNN), Hardware Acceleration, Event-Based Processing, Field-Programmable Gate Array (FPGA).

I. INTRODUCTION

Artificial Neural Networks (ANNs) have become the go-to solution for many machine learning problems [1], [2]. Generally, ANNs start to outperform conventional machine learning approaches when large amounts of training data are available [2]. However, this performance comes at a significant computational cost. For example, the ANN model ResNet-50 requires a total $3.9 \cdot 10^9$ operations to process a single 224×224 image [3]. Generally, a trend can be observed that ANNs increase in size as their classification accuracy improves and the task at hand gets more complex [3]. This places a heavy load on the underlying compute resources in terms on memory, memory bandwidth and processing power. To satisfy non-functional requirements such as power, throughput and latency, careful co-design of the underlying algorithms and the respective processing hardware is necessary [4], [5]. To find more efficient processing systems, inspiration may come from the most efficient cognitive system known to mankind: the human brain. An emerging trend is to implement dedicated hardware to process biologically inspired Spiking Neural Networks (SNNs) [6]–[12]. The term SNN refers to a large set of models

that share one property: the outputs of the neurons (activations) are not encoded with real-valued scalars like in standard NNs, but with sequences of binary events called spikes. What makes SNNs interesting from a computational perspective is their inherent event-driven processing: computations need to be performed only when spikes, i.e., events, occur [13], [14]. To actually achieve a performance advantage compared to standard NNs, three aspects are crucial:

- The neural code determines how information is encoded with binary spikes. The length of the encoding window and the number of spikes required to encode neuronal activations are the most important determinants of the SNN's inference speed and efficiency [15], [16]. In general, the higher the spike sparsity, the better.
- In general, high sparsity in the neuronal output activations requires less computations that need to be performed during inference. While spike sparsity is a nice theoretical property, it is actually very difficult to exploit with standard computer architectures, due to the irregular dataflow associated with it [17].
- The output of a spiking neuron does not depend only on its input but also on its internal state called membrane potential. The real-valued membrane potentials need to be stored and thus increase memory requirements which are already very high to begin with. Strategies have to be deployed to multiplex the membrane potential memory to decrease the overall memory footprint.

To achieve state-of-the-art classification performance on computer vision tasks, established methods from standard NNs have to be adapted to SNNs. These methods are primarily: convolutional layers and pooling layers. To accelerate such Convolutional Spiking Neural Networks (CSNNs) using specialized hardware, most authors propose large spatial arrays of Processing Elements (PEs) [9], [18], [19]. Spatial architectures couple PEs in such way that they can exchange intermediate results without having to access a central memory [5]. Typical implementations use either fixed data path connections between the individual PEs (Systolic Arrays) [10] or Network-on-Chips (NoCs) that feature a highly flexible packet-based interconnect [9], [11]. Systolic arrays are excellent for performing convolutions in cases where dataflow is easily predictable, i.e., in low sparsity situations [20]. NoCs are better at handling unpredictable dataflows since they allow balancing the workload over the different PEs. This comes at the cost of the more expensive NoC communication infrastructure for implementing routers and control circuitry. The major downside of such spatial architectures is that most PEs are left idle if highly sparse activations have to be processed. However, idle PEs still consume power due to leakage and clock switching (the latter only applies for synchronous implementations). Furthermore, idle PEs are not contributing to the end result and are thus wasting chip area. This is complicated by the fact that activation sparsity and the associated irregular dataflow cannot be predicted a priori [4], [21]. Consequently, the non-trivial task of mapping neural operations to PEs must be performed at

The authors are with the Friedrich-Alexander-Universität Erlangen-Nürnberg, 91054 Erlangen, Germany. E-mail: {jan.sommer, akif.oezkan, oliver.keszocze, juergen.teich}@fau.de.

run-time. The main contributions of this paper are as follows:

- A non-spatial hardware architecture optimized for sparse event-based spike processing using a highly efficient neural code.
- A sequential processing scheme that allows the memory for storing the membrane potentials to be multiplexed, keeping the memory footprint low.
- The introduction of a novel memory mapping scheme called *memory interlacing* that allows a highly parallel, fine grained and high-bandwidth distribution of on-chip RAM.
- A queue-based self-timed processing to enable a maximal utilization of PEs, because skipping zero-activations is inherent to the dataflow of the architecture. This allows the execution time to scale directly with the number of spikes. The core idea is to employ less PEs but to maximize their utilization.

II. BACKGROUND ON SPIKING NEURAL NETWORKS

The large variety of SNN-models has arisen due to trade-offs between biological plausibility and model complexity. In this work, we use the integrate-and-fire (IF)-model [22]. The (IF)-model has the least neuro-computational features of real neurons but is very efficient in its implementation [22]. Recent advances have shown the classification performance of SNNs deploying the simple IF-model to be on par with state of the art non-spiking NNs implementations [23], [24]. For example, Sengupta et al. report an error increment of only 0.15% on the CIFAR-10 dataset and an error increment of 0.38% on the difficult ImageNet dataset when using an SNN over a standard NN [23].

A. The Integrate-and-Fire model

Mathematically, the *time discrete* IF-model is described as follows: A binary spike from the previous layer $l-1$ arrives at the synapse i of a neuron j and is weighted with the synaptic weight $w_{i,j}$. The weighted spike is then integrated (i.e. added) into the neurons membrane potential $V_{m,j}^l$. When a membrane potential V_m exceeds the threshold V_t , then the neuron fires a spike itself and V_m is reset to 0. The membrane potential of a neuron j at layer l at each time step t is described as:

$$V_{m,j}^l(t) = \begin{cases} 0 & \text{if } V_{m,j}^l(t-1) > V_t \\ V_{m,j}^l(t-1) + \sum_i w_{i,j} \cdot x_i^{l-1}(t-1) & \text{otherwise} \end{cases} \quad (1)$$

The neuron output x is defined by an all-or-nothing threshold activation function:

$$x_j^l(t) = \begin{cases} 1 & \text{if } V_{m,j}^l(t) > V_t \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Eqns. (2) and (1) have the following implications:

- SNNs are inherently temporal. Their internal state and thus their output is dependent on the time steps t .
- SNNs operate in an event driven manner. They update their internal state V_m only when an event (i.e. a spike) is presented to them.
- The all-or-nothing thresholding leads to a high degree of spike *sparsity*. That means that the occurrence of spikes is a rare event.

The motivation to use SNNs over standard NNs is rooted in the special computational aspects of spike-based processing:

- **No multiplications.** The spikes that encode neuron activations are binary in nature. Weighting the binary activations $\in \{1, 0\}$ does not require an actual multiplication, as the multiplication reduces to: $1 \cdot w = w$ and

$0 \cdot w = 0$. Thus, only adders are required to integrate the weighted spikes on the membrane potential. Adders require significantly less chip area and power compared to multipliers [24].

- **Less compute operations.** The output activations of SNNs are significantly sparser than those of standard ANNs [25]. This sparsity increases in deeper layers of the SNN. Ultimately, this potentially results in less compute operations and thus faster and more energy efficient inference.
- **Less memory accesses.** Even more important than reducing the number of operations is the reduction of memory accesses [26]. Here, SNNs have an advantage since activations are binary and activations are highly sparse, resulting in less data movement.

Despite the interesting properties of SNNs, there are also some problematic aspects that need to be considered when building specialized hardware:

- **Storage of membrane potentials.** Apart from weights and neuron activations, SNNs require an additional data structure: the real-valued membrane potentials of the individual neurons need to be stored and modified during inference.
- **Irregular dataflow.** Activation sparsity cannot be predicted a priori and thus needs to be handled during run-time.
- **Multiple forward passes.** To perform inference on a single sample (e.g. an input image), the entire SNN has to run its forward pass multiple times. This is because it takes multiple time steps for the spikes to propagate through the SNN. How many steps are required depends on (a) the network depth, (b) the neural code and (c) the neuron model.

B. Information Encoding with Binary Spikes

Since spikes are identical to the logical value 1, information is not represented by spike size. Instead, the activation strength is encoded by the timing and/or the amount of the spikes. A lot of effort has been put into researching neural encoding schemes (called neural code) since they are one of the major determinants of an SNNs performance [8], [14], [16], [27], [28]. *Rate coding* is the most used encoding scheme. Here, the activation strength is encoded by the firing rate of the neuron [16]. A high firing rate represents a high activation and vice versa. A neuron decodes an incoming sequence of spikes by performing temporal averaging over a time window to estimate the mean firing rate. With a longer time window, a larger sample size is obtained, resulting in a more accurate fire rate estimate. This makes rate coding a very time consuming process since it takes a considerable amount of time until the estimated mean firing rate has settled to an accurate value [16], [27]. *Time-To-First-Spike* (TTFS) encoding was initially proposed because rate coding could not explain the fast processing in the visual cortex [27]. In TTFS coding, information is encoded in the precise firing times of individual spikes. For this, the time to first spike is measured, i.e. the time that has passed between the arrival of a stimulus and the emission of the spike. A large activation is encoded as an earlier spike transmission and vice versa. TTFS coding enables fast processing since neuronal activations can be encoded with a single spike [29]. Section IV provides a detailed discussion of the neural coded we implemented.

C. Convolutional SNNs

For computer vision tasks, well established techniques like pooling and convolutional layers can be adapted from standard

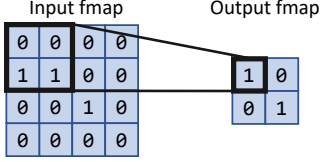


Fig. 1: Max-pooling in SNNs, here with a 2×2 window size.

NNs to SNNs. The general principle is well known from image processing: a 2D-kernel with dimensions $R_l \times R_l$ is convolved over a, typically much larger, 2D input with dimensions $H_l \times W_l$ for each layer l . For simplicity, we consider only quadratic kernels and standard convolutional layers; the basic principles discussed here can be generalized to other kernel shapes or depthwise convolutional layers. The entries of the kernel are the trainable weights. Each output of a neuron (i.e. activation) is a pixel of the resulting 2D output image called a feature map (fmap). A convolutional layer l typically generates a number of C_l output fmaps called channels which can be interpreted as a third dimension. The channels of the input fmap C_{l-1} dictate the number of channels of each kernel K_l . The number of kernels k_l determine the number of output channels C_l , i.e. $C_l = k_l$. For standard spiking convolutional layers, the binary input fmap $\mathbf{X}_{l-1}(t) \in \mathbb{B}^{H_{l-1} \times W_{l-1} \times C_{l-1}}$ is convolved with a number of k_l kernels $K_l \in \mathbb{R}^{R_l \times R_l \times C_{l-1}}$ to which the bias $b \in \mathbb{R}^{C_l}$ is added to get the *update* to the membrane potentials $\mathbf{U}_l(t) \in \mathbb{R}^{H_l \times W_l \times C_l}$. The update to the membrane potential $\mathbf{U}_l(t)$ is added to the membrane potentials $\mathbf{V}_{m,l}(t-1) \in \mathbb{R}^{H_l \times W_l \times C_l}$. The membrane potential $\mathbf{V}_{m,l}(t)$ is then thresholded with $V_t \in \mathbb{R}$ to get the resulting output fmap $\mathbf{X}_l(t) \in \mathbb{B}^{H_l \times W_l \times C_l}$. This results in the following equations, with $*$ denoting the convolution operation:

$$\mathbf{V}_{m,l}(t) = \underbrace{\mathbf{X}_{l-1}(t) * K_l + b_l}_{\mathbf{U}_l(t)} + \mathbf{V}_{m,l}(t-1) \quad (3)$$

$$\mathbf{X}_l(t) = \begin{cases} 1 & \text{if } \mathbf{V}_{m,l}(t) > V_t \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

Pooling can be interpreted as a down-sampling technique to reduce the size of the fmaps. For SNNs, max-pooling is the most used pooling function, due to its simplicity. The pooling operation is performed window-wise in a non-overlapping fashion. Typical window sizes are 2×2 and 3×3 . The implementation of max-pooling for binary fmaps is much simpler than for real-valued fmaps: searching and selecting the maximum in the pooling window is reduced to combining all elements in the pooling window with an or-gate (see Fig. 1). While solutions to perform average pooling have been proposed, they are much more costly to implement and tend to reduce the classification performance of the SNN [24].

III. RELATED WORK

Davies et al. [9] propose the Intel Loihi, an asynchronous Application Specific Integrated Circuit (ASIC). It features a NoC that interconnects 128 PEs called neurocores. Each neurocore has its own memory and processing unit for implementing 1024 spiking neurons as well as an interface to access the NoC. If a neuron emits a spike, the source neuron puts a packet on the NoC which is then routed to the target neuron. The NoC supports only unicast spike communication, i.e. a neuron can only send a spike to a single target neuron. While the communication itself is asynchronous, the

neuronal operations are still conducted in a time discrete (thus synchronous) way. This requires the neurocores to constantly exchange synchronization messages to ensure that they are all in the same algorithmic time step. The NoC allows maximum flexibility in the SNN's connectivity architecture, but it hinders the exploitation of the characteristic dataflow present in pooling- or convolutional-layers. It also lacks the ability to explicitly multicast spike packets. The NoC itself is expensive as it requires a network interface for each neurocore and multiple routers that have to route the packets while preventing collisions and deadlocks.

Wang et al. [18] propose SIES, an FPGA-based accelerator with a 2D systolic array for efficiently calculating convolutions. The core idea of systolic arrays is to read data from memory once, but reuse it in multiple PEs so that less memory accesses are required. SIES uses this highly parallel 2D systolic array only to calculate the *update* of the membrane potential ($\mathbf{U}_l(t)$ as per Equation 3). This increment is then added to the membrane potential of each neuron in a sequential way, which appears to be a major bottleneck. The systolic array architecture does not harvest the high degree of spike sparsity, leaving PEs idle.

Kang et al. [19] propose ASIE, an asynchronous ASIC-based convolutional SNN accelerator that implements event-based processing using the Address Event Representation (AER) protocol. While a 2D fmap in SNNs is basically a $(0,1)$ -matrix \mathbf{M} , it is represented in AER by a list of all addresses (i, j) for which $\mathbf{M}_{i,j} = 1$. The 2D array of PEs is ideally as large as the fmap to be processed because each PE implements a physical neuron. For each address event, only the number PEs defined by the kernel size are actually utilized, leaving most PEs idle. E.g. a 30×30 PE array only utilizes 9 PEs for processing a layer with a 3×3 kernel.

Fang et al. [8] use a technique called High Level Synthesis (HLS) to describe the SNNs dataflow as a network of recursive filters and automatically synthesize it to an FPGA implementation. The convolution operations themselves are performed using a standard, Multiply-Accumulate-based matrix multiplication unit. A deployment of a temporal, non-rate based neural code is deployed to achieve a very energy efficient implementation.

IV. SPIKING NEURAL NETWORK DESIGN

This chapter provides the basis for the hardware implementation of the SNN. The deployment of SNNs is a multi-objective hardware-software co-design effort: the SNN should provide a satisfactory classification result while allowing for fast and energy-efficient processing. An important factor determining the computational performance of SNNs is the neural code. Since recent research suggests TTFS-coding to be the most efficient coding scheme [14], [16], it will be used in this work. To implement TTFS-coding, Rueckauer et al. propose a neuron model where each neuron can fire only a single spike [14]. This is implemented such that a neuron that has already fired cannot fire again. Note that inference on a single sample requires multiple forward passes of the SNN. After inference on a sample is done, the entire SNN is reset so that all neurons are able to fire again. This only-spike-once scheme requires a new neuron model since a single spike has to be enough to push a neurons membrane potential above firing threshold. The main modification is the introduction of the membrane potential slope μ_m . This results in the following system, illustrated in Fig. 2:

$$\mu_{m_j}^l(t) = \mu_{m_j}^l(t-1) + \sum_i w_{i,j} \cdot x_i^{l-1}(t-1) \quad (5)$$

$$V_{m_j}^l(t) = \mu_{m_j}^l(t-1) + V_{m_j}^l(t-1) \quad (6)$$

A neuron output activation x evaluates to 1 if the threshold V_t is crossed and the last time step that a spike has been fired t_{spike} is at 0 indicating that no spike has been emitted so far.

$$x_j^l(t) = \begin{cases} 1 & \text{if } V_{m_j}^l(t) > V_t \text{ and } t_{\text{spike}} = 0 \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

While this only-spike-once methodology results in a very high spike sparsity, it has two inherent downsides:

- The real-valued membrane potential slope has to be stored for each neuron, effectively doubling the memory requirements of the SNN.
- While μ_m must only be updated when spikes occur, V_m of every neuron that has received a spike must be updated every algorithmic time step as a function of μ_m .

These downsides eliminate a lot of advantages of TTFS encoding. To mitigate this, Han and Roy [28] propose a modified TTFS scheme (further referred to as m-TTFS) that gets rid of μ_m at the cost of a few more spikes. The idea of m-TTFS is that once a neuron has exceeded V_t , it emits a spike *every* algorithmic time step. After a sample has been processed for T time steps, the entire SNN is reset and all neurons can fire again. The m-TTFS code leads to the same number of addition operation as the standard TTFS code, but has the advantage that μ_m is not required anymore [28]. While this m-TTFS encoding reduces spike sparsity compared to standard TTFS-encoding, it is still much more efficient than rate coding because it reduces the number of forward passes that are required for a single inference by orders of magnitude [28].

Spikes are inherently discontinuous and non-differentiable. This renders the well established gradient-based backpropagation learning approach used for standard NNs impossible. The training of SNNs is still a very active research area with no established “best practice”. Three major approaches can be identified:

- **Spike Timing Dependent Plasticity:** STDP describes a large set of bio-inspired learning rules that share one property: the synaptic weights are adapted depending on the firing time difference between neurons in layer l and its preceding layer $l-1$. The basic principle is that the connection strength (=weight) of neurons is increased when they fire together, and that the connection strength is decreased when the time difference between spikes is large [30]. The standard STDP algorithm is unsupervised, i.e. no labeled training data is required.
- **Backpropagation:** this class of approaches try to overcome the non-differentiable nature of SNNs in order to enable spike-based backpropagation. Most approaches (a) try to approximate the derivatives to make gradient learning possible [25], (b) employ differentiable surrogate activations to replace the non-differentiable threshold operation [31] or (c) use a tandem approach where a SNN is performing the prediction and an equivalent standard NN is adapting the weights using backpropagation [32].
- **Conversion:** here the idea is to circumvent the backpropagation problem by training a standard ANN and reusing the trained weights for the SNN. This has the major advantage that the already very mature techniques and toolchains established for standard NNs can be used for training. Deep SNNs trained with conversion methods show the best classification accuracy of all methods discussed here [23], [24].

Here, we use a conversion approach to train the CSNN (see Section VII for details).

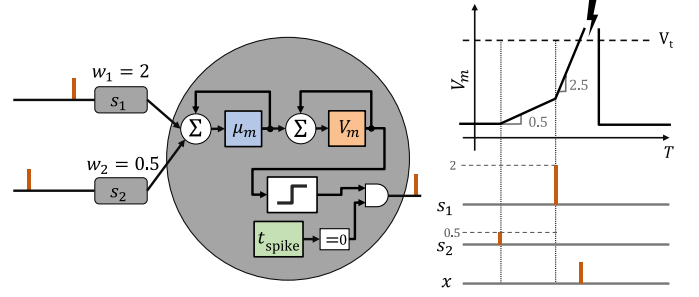


Fig. 2: Behavior of a TTFS encoded IF-neuron over time. The neuron integrates the weighted input spikes on the membrane potential slope μ_m . This slope dictates the rise or fall of the membrane potential V_m over time. When V_m reaches the firing threshold V_t and no spike has been emitted previously, the neuron emits a spike. Due to μ_m , this neuron can spike even from a single input spike.

V. HARDWARE DESIGN

The main observation is that existing implementations employ large arrays of PEs that are then highly underutilized due to spike sparsity. Another common problem is determining the number of PEs for a CNN accelerator (be it spiking or non-spiking). Most implementations like SIES [18] or ASIE [19] deploy a PE array where its ideal size is the size of the 2D fmap that has to be processed. However, the fmap sizes in CNNs change from model to model and from layer to layer. This fact makes maximizing the efficiency of such PE arrays very difficult. This problem of diminishing dimensions is discussed in more detail by Chen et al. [20]. We argue that one dimension is fixed for the most parts of established CNN architectures: the kernel size. Szegedy et al. argue that convolutions with filters sizes larger than 3×3 “might not be generally useful as they can always be reduced into a sequence of 3×3 filters” [33]. Simonyan and Zisserman [34] argue that, for example, two 3×3 convolutional layer have more discriminative power than a single 5×5 layer as the former incorporates two non-linearities while the latter only includes one. Thus, it comes with no surprise that established, well performing architectures like ResNet [35], Inception-V3 [33] or MobileNet [36] all deploy 3×3 kernels for the vast majority of their architecture. For this reason, the proposed architecture is optimized for 3×3 kernels while 1×1 kernels for pointwise layers are also possible. Nevertheless, the techniques discussed here can also be generalized to other kernel sizes. The core idea of this architecture is to employ less PEs but to constantly keep them busy. To ensure that the PEs run at maximum capacity, spikes are represented as address events that are compressed into queues. As soon as all spikes of a queue are processed, the next queue is selected. This allows the processing time to scale with the number of spike events and results in self-timed execution of the SNN. We start by providing a top-level overview of the hardware architecture and then proceed to show how it can be implemented efficiently on either FPGAs or ASICs.

A. Top-level overview

The goal is to maximize the utilization of PEs while minimizing the number of PEs. To achieve this, the individual fmaps are stored in a compressed format. This compressed format is a queue of all spikes in a fmap whereby the spikes are not represented by a logical 1 but by their Address Event Representation (AER). The AER of a spike is simply the

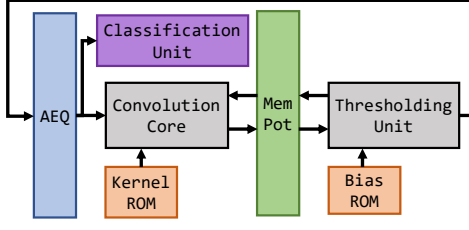


Fig. 3: Top-level architecture that shows the data flow between the different units. The AEQ stores the address events that are read by the convolution core. The convolution core updates the MemPot memory depending on the address events. The thresholding unit adds the bias to the neurons stored in MemPot. Also, the thresholding unit thresholds the neurons to generate the address event that are stored in the AEQ. The classification unit performs implements an Fully Connected layer and performs the final classification.

spike's (i, j) coordinates in the 2D fmap (see Fig 4 for a visual example). To process an fmap, the Address Event Queue (AEQ) must be processed, which has the advantage that the number of processing steps scales directly with the number of spikes. The architecture proposed here consists of six distinct units:

- The AEQ that stores the address events.
- The Membrane Potential memory (MemPot) for storing the V_m of the neurons.
- The convolution unit that receives the address events from the AEQ and updates MemPot.
- The thresholding unit that perform multiple tasks:
 - 1) Threshold the MemPot and write the resulting address events to the AEQ.
 - 2) Perform max-pooling if required.
 - 3) Apply the bias to neurons in MemPot and set them back to 0 if required.
- The Read Only Memory (ROM) for storing the kernel weights K and biases b .
- The classification unit performs the final classification using a small fully connected layer. Its functionality will be omitted here as the focus of this work is on accelerating the convolutional layers.

Fig. 3 provides an overview over the dataflow of the architecture.

B. Performing convolution with address events

Performing convolution with address events requires rethinking the well-known sliding window frame-based convolution. For this, Morales et al. [37] propose an algorithm for event-based convolution. To process an address event at position (i, j) , all neurons affected by this address event have to be updated with the respective kernel weights. The affected neurons are determined by the neighbourhood of the convolution kernel. A 3×3 kernel requires updating the neuron potentials at position (i, j) and all 8 neighboring membrane potentials. To get the same result as with standard sliding-window based convolution, the respective weights of the kernel can be added to the neuron potentials by rotating the kernel by 180° . This is further explained in Fig. 4 and in [37]. Note that this leads to the same result as performing standard sliding window based convolution. However, with this principle, only additions are needed and the number of operations scales with the number of address events in the AEQ.

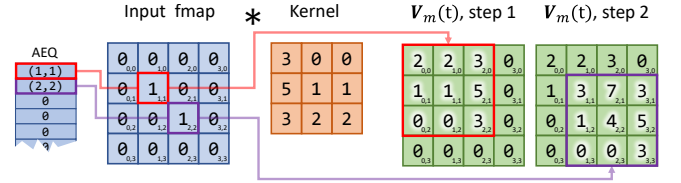


Fig. 4: Convolution with address events. A 2D input fmap is stored as a queue of address events (the AEQ). To perform convolution, the AEQ is processed sequentially. Two address events allow convolution to be performed in two successive steps, whereas sliding window based convolution in this example would require $4 \times 4 = 16$ steps, one step for each pixel position in the input fmap. For each address event, 9 neurons (highlighted in white) can be updated in parallel, due to the 3×3 neighbourhood of the kernel. In V_m , step 1 it is easy to see how the membrane potentials are updated with the kernel contents rotated by updated by 180° . This rotation is necessary to achieve the same result as with sliding-window based convolution (see [37]). Note that V_m is initialized with zeroes. Also, note that only the AEQ is used to store spikes, the 2D Input fmap is only shown to allow for an easy interpretation of the AEQ's content.

Because all spike events are located inside the AEQ, the clock cycles required to perform the convolution scale directly with the number of spikes, i.e., one clock cycle per event. Due to a high degree of spike sparsity, this leads to a significant speed-up. All 9 neuron membrane potentials can be updated in parallel because there is no data dependency. Thus, a total of 9 PEs are required in the convolution unit. Each PE implements an adder that receives a membrane potential and a kernel $K[\cdot]$ and returns an updated membrane potential. Note that K refers to *all* kernels of the SNN, thus for each convolution the correct kernel must be selected depending on the current layer l , the current input channel c_{in} and output channel c_{out} . We use the following notation to indicate the selection of the correct kernel for the current convolution: $K[c_{out}, c_{in}, l]$. Additional adders are required to calculate the addresses of the affected neurons.

C. Thresholding Unit

The thresholding unit performs three distinct tasks: max-pooling, thresholding and the addition of the bias. Like the convolution unit, the thresholding unit operates in a 3×3 neighborhood and thus processes 9 inputs in parallel. The thresholding unit does not operate in an event-based manner because *all* neurons need to be visited in order to threshold them and to add the bias to them. The processing sequence of the thresholding unit is as follows (also illustrated in Fig. 5):

- 1) Add the scalar bias b to all neuron potentials in the current 3×3 window.
- 2) Threshold all neuron potentials in the 3×3 window with V_t :

No Max-pool: Write the addresses of the neurons in the current window to the AEQ if they exceed the threshold value V_t .

Max-pool enabled: If *any* neuron potential in the window crossed V_t , write the max-pooled address to the AEQ. How the max-pooled address can be calculated will be described later.

- 3) The 3×3 window is moved with a stride of 3, i.e. it moves 3 pixels ahead.

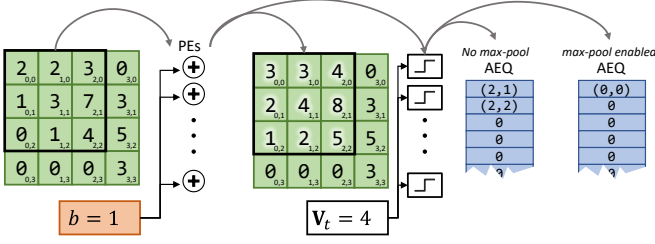


Fig. 5: Functionality of the thresholding unit at the example of a single channel. Nine parallel adders add the bias and 9 parallel comparators do the thresholding in the 3×3 window. The window is moved with a step size of three over all neurons. How max-pooling with address events works will be discussed in more detail later.

- 4) Repeat until all neurons of the current channel have been thresholded.

D. Scheduling strategy

The convolution unit can only perform convolution on a single channel. To process a multichannel convolutional NN with multiple layers, the convolution unit must be applied to all channels and all layers in the correct order. This is a complex scheduling problem: which kernels and biases need to be applied to which address events in which order. The naive solution to this scheduling problem would be to implement as many convolution units as required to perform all operations in parallel. This is not a feasible solution because hardware and memory resources are limited. The goal of the processing strategies proposed here is to keep the required memory resources as low as possible. There are three types of data structures in SNNs:

- The membrane potentials \mathbf{V}_m stored in MemPot are not sparse and require significant memory resources.
- The sparse output activations are stored in the AEQ as address events.
- The weights and biases are stored in uncompressed form.

The key to reducing memory requirements is to reuse, i.e. multiplex, the MemPot memory for each channel. Consider the following example: a layer has 32 channels. Each channel requires 10 kb to store the membrane potentials. Thus, 320 kb of memory are required. However, if each channel is processed one after the other, only 10 kb are required in total. Here, an SNN is processed layer by layer. Each layer of an SNN needs to be simulated for multiple time steps T . To maximize the reuse of MemPot, processing is done in a channel-wise fashion. Each output channel of a layer is simulated for all time steps t , one channel after the other.

The output fmap of each channel is represented by its own AEQ. These AEQs can be implemented in a single dual-port RAM since each individual AEQ is processed sequentially. Algorithm 1 shows the dataflow¹ of the processing scheme where the output address events of a layer $l-1$ are processed to get the output address events of l . We use C_{l-1} and C_l to denote the number of input and output channels of the current layer l . The memory MemPot for storing \mathbf{V}_m is only large enough for a single channel. MemPot is reused for every output channel in order to keep the memory requirements low.

¹The dataflow for standard convolution is shown. The scheme can easily be adapted to support depthwise convolution as well. For didactic reasons and lack of space, we refrain from showing this.

Algorithm 1 Schematic dataflow for processing a layer l

```

1: for  $c_{out} \leftarrow 0$  to  $C_l$  do
2:    $\mathbf{V}_m \leftarrow 0$            ▷ reuse for each output channel  $c_{out}$ 
3:   for  $t \leftarrow 0$  to  $T$  do   ▷ Simulate layer for all time steps  $T$ 
4:     for  $c_{in} \leftarrow 0$  to  $C_{l-1}$  do
5:       ▷ Update  $\mathbf{V}_m$  using the address events from the AEQ
6:        $\mathbf{V}_m \leftarrow \text{ConvolutionUnit}(\text{AEQ}[c_{in}, l-1, t], K[c_{out}, c_{in}, l], \mathbf{V}_m)$ 
7:       end for
8:       ▷ Save the address events from the thresholding unit
9:        $\text{AEQ}[c_{out}, l, t] \leftarrow \text{ThreshUnit}(b[c_{out}], V_t, \mathbf{V}_m)$ 
10:    end for
11:  end for

```

VI. IMPLEMENTATION

This section shows how the top-level architecture described in the previous chapter can be mapped to actual hardware. The top-level architecture poses one essential challenge that needs to be addressed: the membrane potential \mathbf{V}_m is stored in a single memory (MemPot). However, to perform convolution or thresholding, 3×3 membrane potentials need to be accessed in parallel. Dual-port RAMs (such as BRAMs present on most FPGA chips) typically only supports one write and one read access per clock cycle, rendering 9 parallel read/write operations impossible. To solve this problem, a novel memory distribution strategy called memory interlacing is presented here. The idea of memory interlacing is to distribute all elements of MemPot over 9 different RAMs called columns, such that 9 concurrent read/write operations are possible. The elements of the membrane potential need to be placed into the 9 memory columns in a certain fashion: regardless on which position (i, j) the 3×3 window is placed, all 9 elements must come from one memory column each. This memory interlacing scheme is further explained in Fig 6. Each element is addressed uniquely by its address (i, j) and its column $s \in (0, \dots, 8)$. For a more compact notation, $(i, j)[s]$ is used to define the unique address of a neuron. The memory interlacing scheme has multiple positive effects:

- Instead of one large monolithic memory, \mathbf{V}_m is distributed to 9 smaller RAMs. Smaller RAMs tend to be faster and more energy efficient.
- The multiple small RAMs can be distributed closer to the PEs, which further increases speed and energy efficiency.
- The interlaced processing effectively prevents data-hazards.

In the following, we will discuss the hardware implementation of the different modules in detail.

A. Address Event Queue (AEQ)

The AEQ has to store the address events in queues. This is done in an interlacing fashion (see Fig. 7). The AEQ is not only a data structure but also features two independent circuits: one for writing and one for reading the queue columns. The thresholding unit writes the address events and the convolution unit reads them.

Write Logic: The queues in the 9 columns can be filled in parallel. The 9 parallel write accesses are necessary since thresholding is performed in a 3×3 window. The write logic features 9 write counters, one for each column. An address event is only written to a queue if the respective write enable is set.

Read Logic: Since the address events are processed one after the other, the queues are read sequentially, from queue 0 to 8. Therefore, one read counter and the column-select counter $\in (0, \dots, 8)$ to select the correct queue is required. Each

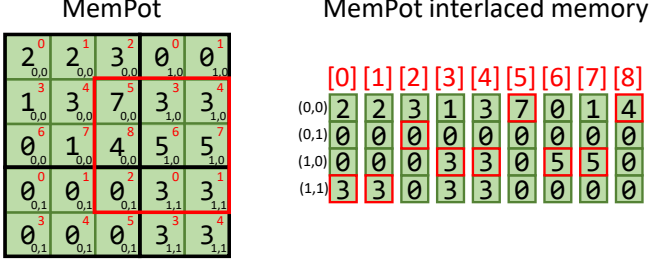


Fig. 6: Memory interlacing scheme (right) and its visual representation (left). All membrane potentials are stored in the 9 memory columns 0 to 8. The elements are distributed in such a way that a 3×3 window (red) always accesses all memory columns in parallel, no matter where it is placed. Each element is uniquely addressed by its address (i, j) and its $s \in (0, \dots, 8)$ displayed in red. For example, the top left memory potential with the value 2 and the address $(0,0)[0]$ is stored in column 0 at address 0,0. Each column can be implemented with a single dual-port RAM.

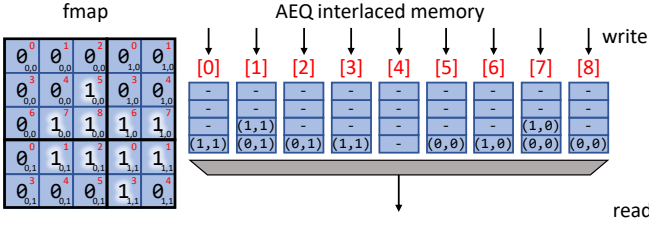


Fig. 7: Interlacing scheme applied to the AEQ. The spikes represented as address events (i, j) and are sorted into the respective column $s \in (0, \dots, 8)$, displayed in red. Note that the binary fmap containing the spikes is only shown as a easy to understand visualization of the AEQ's contents.

entry in the queue not only contains the address event but also two extra bits: the valid bit and the end-of-queue bit. This valid bit indicates if an address event is valid. The valid bit is used to indicate empty queues. The end-of-queue bit indicates the last element of the queue and leads to an increment of the column-select counter. If one queue columns is completely empty, then one clock cycle is wasted by reading an invalid address event and incrementing the column-select counter.

B. Convolution Unit

The functionality of the convolution unit can be split into multiple sub-units: address calculation, kernel permutation, MemPot update calculation and data hazard detection. The circuit of the convolution unit is pipelined into four stages S1 to S4. The general data flow in the convolution unit is as follows:

- Start** Receive address event: When the AEQ receives a read-enable signal, it starts to read out the 9 queues sequentially from a given offset. The convolution unit receives the input address event $(i, j)[s]_{in}$ to uniquely identify the location of an incoming spike.
- S1** Calculate addresses: The convolution unit calculates the addresses of all affected neurons in the 3×3 neighborhood.
- S2** Read MemPot: The convolution unit reads the 9 membrane potentials from the calculated addresses. Also, the permuted kernel will be selected here.

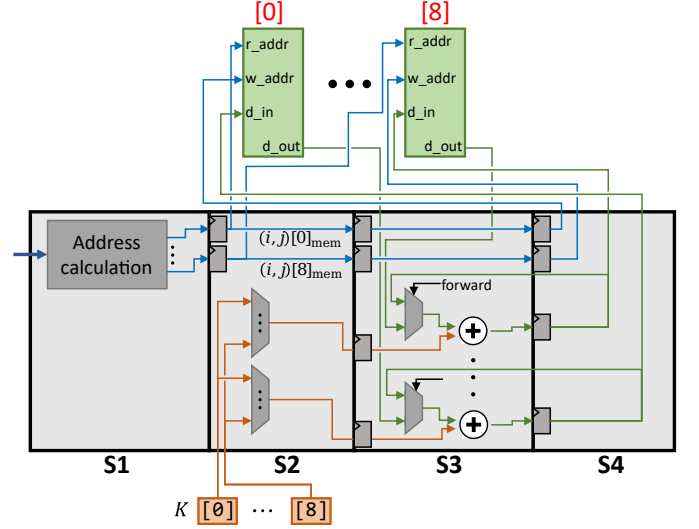


Fig. 8: Pipelined architecture of the convolution unit. The convolution unit is setup in four stages. The core is operating on 9 membrane potentials (green) and 9 kernel weights (orange) in parallel. In S1, the read addresses of all 9 membrane potentials is calculated. These read addresses get passed along the pipeline and become the write addresses in S4. In S2, the read addresses are delivered to the memory columns. It takes one clock cycle for the MemPot memory to deliver the output data. Also, the 9 kernel elements (orange) are sorted in the right order and transported to the PEs in S3. The adders in S3 then update the membrane potentials. In S4, the updated membrane potentials are written back. Note how each PE is permanently connected to the respective memory column of MemPot. For better visibility, only the logic and data paths of two PEs are displayed, the dots \dots indicate the existence of the missing 7 data paths.

- S3** Calculate update: The convolution unit adds the respective kernel weights to the 9 membrane potentials.
- S4** Write back MemPot: The 9 updated membrane potentials are written back to MemPot to the same addresses calculated in S1.

Pipelining increases the parallelism inside the convolution unit because all four stages are executing their respective task in parallel (see Fig. 8). This ensures that all parts of the convolution unit are busy. For example, S3 is updating the membrane potentials of an address event while S2 is already fetching all membrane potentials for the next address event. Furthermore, since the combinational parts of the circuit are divided into four stages, a much higher clock frequency can be achieved. However, pipelining also has some downsides. At the start, it takes four address events until the pipeline is completely filled (wind-up). Only a full pipeline can deliver maximum efficiency. It is therefore important that a constant flow of address events is provided so that the pipeline stages are always saturated. This constant supply can be provided since the address events of a channel are compressed into a queue that is read event by event. A downside of pipelining is the occurrence of *data hazards* caused by data dependencies.

Address calculation: The convolution unit receives an input address event $(i, j)[s]_{in}$ from the AEQ. The address calculation logic must then compute what the addresses of the nine affected membrane potentials in the kernel neighbourhood are so that they can be fetched from MemPot. Here, we use the subscript “in” to refer to the components of the input address

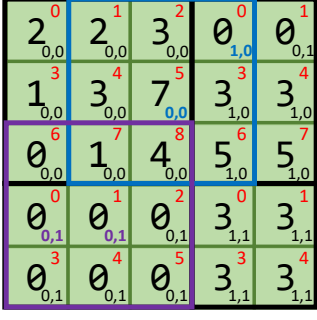


Fig. 9: Example for the address calculation scheme with interlaced memory. Blue kernel: the input address event $(0, 0)[5]_{in}$ comes from $s_{in} = 5$, which defines the center of the kernel. To calculate the address i_{mem} for $s_{mem} = 0$, Eqn. (8) is used to get $i_{mem} = i_{in} + 1 = 1$. Purple kernel: the input address event is $(0, 1)[1]_{in}$, defining the center of the kernel. To calculate the i_{mem} for $s_{mem} = 0$, Eqn. (8) is used to get $i_{mem} = i_{in} = 0$ since $s_{in} = 1$ is not 2 or 5 or 8.

events and “mem” to refer to the components of a neuron’s address in MemPot. Nine different addresses $(i, j)_{mem}$ need to be calculated for each of the 9 MemPot memory columns: $(i, j)[0]_{mem}, \dots, (i, j)[8]_{mem}$. Consider the following example: an input address event has the address $(i, j)[s]_{in}$. For simplicity, consider only the $(i, j)_{mem}$ addresses for column 0 $((i, j)[0]_{mem})$. If the input address event comes from column $s_{in} \in \{2, 5, 8\}$ of the AEQ, then $i_{mem} = i_{in} + 1$ otherwise $i_{mem} = i_{in}$ as described in Eqn. (8). Similarly, Eqn. (9) describes how j_{mem} can be calculated. For the remaining columns s_{mem} , the address calculation logic can be constructed in a similar fashion, which we refrain from showing here due to spacial limitations.

$$i_{mem} = \begin{cases} i_{in} + 1, & \text{if } s_{in} \in \{2, 5, 8\} \\ i_{in}, & \text{otherwise} \end{cases}, \text{ for } s_{mem} = 0 \quad (8)$$

$$j_{mem} = \begin{cases} j_{in} + 1, & \text{if } s_{in} \in \{6, 7, 8\} \\ j_{in}, & \text{otherwise} \end{cases}, \text{ for } s_{mem} = 0 \quad (9)$$

This example for i_{mem} is described visually in Fig. 9. Four adders are needed to calculate $i + 1, i - 1, j + 1, j - 1$ and 9 comparators are required to check from which of the 9 columns in the AEQ the address event came from. Parts of the kernel can be out of bounds of the fmap when the kernel center is directly at the fmap’s edge. To avoid errors, this out-of-bound condition must be detected so that no membrane potential updates occur for the parts in question. Out-of-bounds detection is performed by detecting under/overflows in the address calculation logic. For example, an address event at $(0, 0)[0]$ would cause MemPot column $s_{mem} = 8$ to be addressed with $(-1, -1)$. Since the address calculation logic only supports non-negative integers, this would cause an underflow that can be detected with very little hardware overhead.

Kernel permutation: Each of the 9 PEs is essentially an adder, performing the updates of the membrane potentials is connected to one of the 9 memory columns of MemPot. Each PE receives one of the nine kernel elements $K[0]$ to $K[8]$ and a membrane potential as an input. While a channel is processed, the kernel itself does not change. However, the mapping of the 9 kernel elements $K[\cdot]$ to the 9 PEs changes depending on the

location of the input column $s_{in} = 5$ that the address event came from. Reconsider the example in Fig. 9. For the blue kernel, the top left kernel element ($K[0]$, assuming that the kernel is already flipped by 180°) has to be mapped to the PE of MemPot column $s_{mem} = 1$. For the purple kernel however, $K[0]$ has to be mapped the PE of $s_{mem} = 6$. Since there are 9 memory columns there are 9 different permutations of the kernels weights. All 9 possible permutations are calculated in parallel in S2. The correct permutation is then selected with a multiplexer for each PE (see Fig. 8). The hardware cost is relatively low: a total of nine 9-to-1 multiplexers are needed.

Update calculation: During the calculation of the membrane potential update, arithmetic overflows or underflows might occur. When an overflow occurs, a large membrane potential overflows and becomes a negative membrane potential. Underflows are even more critical because strongly negative membrane potentials become very large membrane potentials, generating erroneous spikes. The obvious solution to this problem would be to adapt the bit widths of all data paths so that over/underflows are impossible. However, this would significantly increase the required hardware resources. Instead, saturation arithmetic is used here. If the result of an addition is larger than the maximum, it is clamped to the maximum representable value. In a similar fashion, too small values are clamped to the minimum representable value. Saturation works well for SNNs with m-TTFS coding: A further decrease of an already very negative membrane potential has no effect on the output of the neuron. Similarly, further increasing the membrane potential when it is well above firing threshold does not change the neurons output. To implement saturation arithmetic in hardware, over/underflow detection is required. Over/underflow detection only requires checking a single bit and thus is cheap to implement.

Data hazard mitigation: In this architecture, Read After Write (RAW) data hazards can occur when an updated membrane potential is read from memory that has not yet been calculated or written back. When the RAW-hazard is not handled, the update of an membrane potential will be overwritten. There are two situations where a RAW-hazard can occur: between S2-S4 and between S2-S3. In the case of S2-S4, S2 reads from the same address that S4 is currently writing to. Writing to the memory takes one clock cycle and thus S2 receives an outdated membrane potential. Resolving the S2-S4 hazard is cheap because the updated membrane potential has already been calculated, just not written back yet. The updated membrane potential must only be forwarded directly to S3, bypassing MemPot (see Fig. 8). The S2-S3 hazard occurs when S2 is reading from a memory address for which S3 is currently calculating an update. In this case, forwarding does not work because the update has not yet been calculated. The solution is to stall S2, S1 and the AEQ for one clock cycle so that S3 can finish the calculation. The S2-S3 hazard is then a S2-S4 hazard that can be resolved by forwarding. To implement hazard detection, 9 comparators are required for S3 and 9 comparators are required in S4. The forwarding logic is cheap to implement: only 9 2-to-1 multiplexer are required, one for each PE.

S2-S4 hazards are not a problem because they can be resolved by forwarding. S2-S3 hazards should be avoided because they require parts of the pipeline to be stalled. This reduces throughput and leaves parts of the pipeline unoccupied. The probability of S2-S3 hazards is greatly reduced due to the design of the AEQ interlaced memory. S2-S3 hazards occur when two immediately successive addressing events access overlapping membrane potentials. See for example Fig. 9. If the address event 0,1 column 1 (purple) is processed directly

after event 0,0 column 5 (blue) then a S2-S3 hazard would occur for column 7 and 8 (where the kernels overlap). This cannot happen with the AEQ design because all address events from memory columns 2, 3 and 4 are processed first. Apart from that, the address events are read from the AEQ column-wise, i.e. first all address events from column 0 are read, then column 1 and so on. This is a major advantage because address events from the same column index will never access overlapping membrane potentials. Processing address events from the same column will never result in data hazards. Data hazards can only occur on switching from one column to another.

C. Thresholding Unit

The thresholding unit has to visit every membrane potential in the MemPot. It does so by sliding a 3×3 window over the membrane potential with a stride of 3. The general architecture shows some similarities to the convolution unit. The thresholding unit starts its operation as soon as its clock enable signal is set to 1. A pipelined design with five stages S1 to S5 deployed:

- S1** Calculate addresses: The thresholding unit calculates the addresses of all 9 membrane potentials in the 3×3 window that is currently processed.
- S2** Read MemPot: The thresholding unit reads the 9 membrane potentials from MemPot.
- S3** Add Bias: The scalar bias is added to all 9 membrane potentials.
- S4** Threshold: The updated membrane potentials are compared to the threshold V_t to determine if they fire a spike.
- S5** Write MemPot and AEQ: The updated membrane potentials are written back to MemPot. If a spike is generated in S4, then the respective AEQ-column is written. Also, when enabled, max-pooling is performed here.

It is important to note that no data hazards can occur in the thresholding unit. This is because each membrane potential is read out only once. Fig. 10 gives a schematic overview of the thresholding unit.

It is important to note that no data hazards can occur in the thresholding unit. This is because each membrane potential is read out only once.

Address Calculation: Address calculation is very simple, thanks to the interlaced memory. Only two counters are required, one for the i_{mem} coordinate and one for the j_{mem} coordinate. Addressing all memory columns of MemPot with the same $(i, j)_{\text{mem}}$ address accesses a 3×3 window by design (see for example Fig. 11 where all membrane potentials with the coordinate (0,0) are located in a 3×3 window).

Bias update: The scalar bias is used as an input for all 9 PE (as illustrated in Fig. 5). Saturation arithmetic is used to avoid underflows and overflows, akin to that of the convolution unit.

Thresholding: Thresholding is performed by 9 parallel comparators. Due to m-TTFS encoding, each neuron that has fired already needs to fire again. This is implemented with a spike indicator bit that is stored together with the neurons membrane potential in the MemPot memory. The comparators check two conditions: if the firing threshold is crossed and if the spike indicator is set. If any of these two condition is true, then the spike indicator bit of the respective neuron is set to 1. It is only set back to 0 if a new sample has to be processed.

Writing the AEQ: How the generated address events (indicated with a “out” subscript) are written to the AEQ depends on whether max-pooling is enabled or not. The simplest case is when max-pooling is disabled. In this case, the address event $(i, j)[s]_{\text{out}}$ of a spiking neuron is simply its MemPot address

Algorithm 2 Calculate max-pooled address event

```

1:  $s_{\text{out},i} \leftarrow 0$  ▷ Counts in the sequence 0,1,2,0,1,2,...
2:  $s_{\text{out},j} \leftarrow 0$  ▷ Counts in the sequence 0,3,6,0,3,6,...
3:  $i_{\text{out}} \leftarrow 0$ 
4:  $j_{\text{out}} \leftarrow 0$ 
5: for  $j_{\text{mem}} \leftarrow 0$  to  $j_{\text{max}}$  do
6:   for  $i_{\text{mem}} \leftarrow 0$  to  $i_{\text{max}}$  do
7:     if  $i_{\text{mem}} = i_{\text{max}}$  then
8:        $s_{\text{out},i} \leftarrow 0$ 
9:        $i_{\text{out}} \leftarrow 0$ 
10:      if  $s_{\text{out},j} = 6$  then
11:         $s_{\text{out},j} \leftarrow 0$ 
12:         $j_{\text{out}} \leftarrow j_{\text{out}} + 1$ 
13:      else
14:         $s_{\text{out},j} \leftarrow s_{\text{out},j} + 3$ 
15:      end if
16:    else
17:      if  $s_{\text{out},i} = 2$  then
18:         $s_{\text{out},i} \leftarrow 0$ 
19:         $i_{\text{out}} \leftarrow i_{\text{out}} + 1$ 
20:      else
21:         $s_{\text{out},i} \leftarrow s_{\text{out},i} + 1$ 
22:      end if
23:    end if
24:     $s_{\text{out}} = s_{\text{out},i} + s_{\text{out},j}$  ▷ Max-pooled column
25:  end for
26: end for

```

$(i, j)[s]_{\text{mem}}$. With 3×3 max-pooling enabled, the resulting address event $(i, j)[s]_{\text{out}}$ has to be calculated. Consider the example in Fig. 11. For example, all spikes from the addresses $(0, 1)[0]_{\text{mem}}$ to $(0, 1)[8]_{\text{mem}}$ have to be mapped to a single address event $(0, 0)[3]_{\text{out}}$. The calculation of this mapping is inexpensive to implement in hardware with four counters that run along with the address calculation logic. To avoid expensive division operations, a sequential circuit can be constructed using only adders to perform the address and column index calculation. An algorithmic representation is can be seen in Algorithm 2. The counters $i_{\text{mem}}, j_{\text{mem}}$ are used to calculate the addresses of the 3×3 window applied to MemPot. The counters $s_{\text{out},i}$ and $s_{\text{out},j}$ are used to calculate the s_{out} of the max-pooled address event. To save power, the clock enable of the max-pooling calculation logic can be turned off.

VII. EXPERIMENTS AND EVALUATION

To evaluate the effectiveness of the proposed architecture, we first trained a small CSNN on the MNIST² and the more difficult Fashion-MNIST³ dataset. Training was performed with a conventional CNN using Tensorflow Keras⁴ the clamped ReLU activation function (as described by Rueckauer et al. [14]). In preparation for the deployment in hardware, the CNN was then retrained using quantization-aware training [38]. The weights of the CNN were then converted using the SNN-Toolbox⁵ proposed by Rueckauer et al. [14], [24] and quantized to 8 and 16 bit. The CSNN has a structure of $(28 \times 28\text{-}32\text{C}3\text{-}32\text{C}3\text{-}P3\text{-}10\text{C}3\text{-}F10)$. The notation is as follows. Convolutional layers: $\langle \# \text{channels} \rangle \text{C} \langle \text{kernel size} \rangle$, Max-pooling layers: $P \langle \text{window size} \rangle$, Fully-connected layers: $F \langle \# \text{neurons} \rangle$. Experimentally, it was found that simulating this m-TTFS encoded CSNN for $T = 5$ time steps yielded the best classification accuracy. The input frames consist of integer pixels that need to be binarized before processing in order to get input spikes that are then fed to the CSNN. The binarization can be achieved by thresholding the

²<http://yann.lecun.com/exdb/mnist/>

³<https://github.com/zalandoresearch/fashion-mnist>

⁴<https://keras.io/about/>

⁵<https://snntoolbox.readthedocs.io/en/latest/>

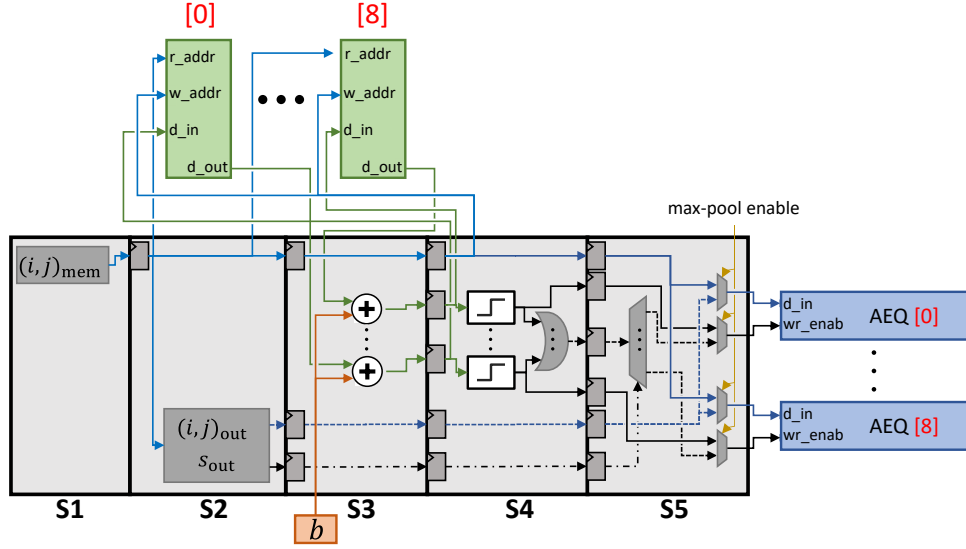


Fig. 10: Pipelined thresholding unit. The MemPot addresses $(i, j)_{\text{mem}}$ are calculated in S1. In S2, the MemPot is read and the max-pooled address and memory column are calculated. In S3, the bias update is calculated. In S4, the thresholding is performed. Also, the result of the bias update is written back. For max-pooling, a 9-to-1 or-gate combines all outputs of the comparators. In S5, the results of the thresholding are written to the AEQ columns. If max-pooling is enabled, then the max-pooled address is forwarded to the data ports of the AEQ columns. The calculated max-pool column is used to select the correct AEQ column's write enable. If max-pooling is disabled, the MemPot addresses $(i, j)_{\text{mem}}$ are connected to all data ports. Also, each comparator is connected to its AEQ column via the write enable. Thus, 9 address events can be written in parallel. Signals related to max-pooling are indicated by dashed lines.

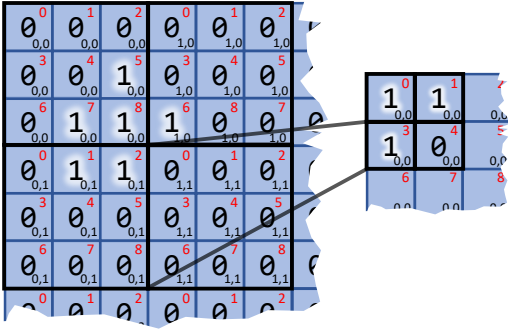


Fig. 11: Max-pooling with interlacing memory columns requires the calculation of the address $(i, j)_{s_{\text{out}}}$ and the respective memory column. For example, all spikes that are generated from address $(0, 1)[0]_{\text{mem}}$ to $(0, 1)[8]_{\text{mem}}$ are pooled to a single address event with the address $(0, 0)[3]_{\text{out}}$.

individual frames. For a simple dataset like MNIST where the background is clearly separated from the object, applying only a single threshold is sufficient. However, this inevitably leads to a loss of information and we thus propose to convert the input frame into binary spikes by applying a set of thresholds $P = (p_1, p_2, \dots, p_{T-1})$. An important property of P is that it is a strictly increasing set to mimic m-TTFS encoding.

The proposed architecture was synthesized for the Xilinx Zynq UltraScale+ XCZU7EV FPGA. The power estimation is performed with the Vivado Power Estimator tool. The general architecture of the accelerator is very compact, therefore the parallelism and thus the latency can be improved by implementing multiple units in parallel. We tested our implementation with multiple degrees of parallelization, referring

to the number of AEQs, MemPot memories, Kernel and Bias ROMs, thresholding units and convolution units. We found that for this CSNN, a parallelization of $\times 8$ yielded the best energy efficiency (see Table I). Table II provides detailed synthesis and utilization results (in terms of LUTs, Flip-Flops (FFs), Block-RAM (BRAM) and dedicated DSPs) and compares them to related work. To evaluate the effectiveness of the proposed approach, we compared the input activation sparsity for each layer with the PE utilization in Table III. Sparsity refers to the number of non-zero activations in relation to all activations. PE utilization measures the clock cycles in which the PEs receive valid address events relative to all clock cycles required to process the CSNN. Note that the PE utilization does not take into account that there might be zero weights that do not lead to an update to MemPot. Table V shows the performance statistics of the architecture proposed here and compares them to other CSNN MNIST implementations. Table IV compares the accuracy on the Fashion-MNIST dataset with related work. Direct quantitative comparisons should be made with caution because the different approaches do not use identical SCNN architectures for their experiments. Nonetheless, the SCNN sizes are comparable enough (all contain three to four trainable layers) to allow for a meaningful qualitative comparison. Fig. 12 provides an overview over how many hardware resources are consumed by the individual units.

VIII. CONCLUSIONS

The SNN hardware implementation strategy developed in this work allows for low-latency and power efficient processing of convolutional SNNs. It supports state of the art SNN architectures thanks to the hardware implementation of max-pooling and neuronal biases. This allows the deployed SNNs to achieve a competitive classification accuracy.

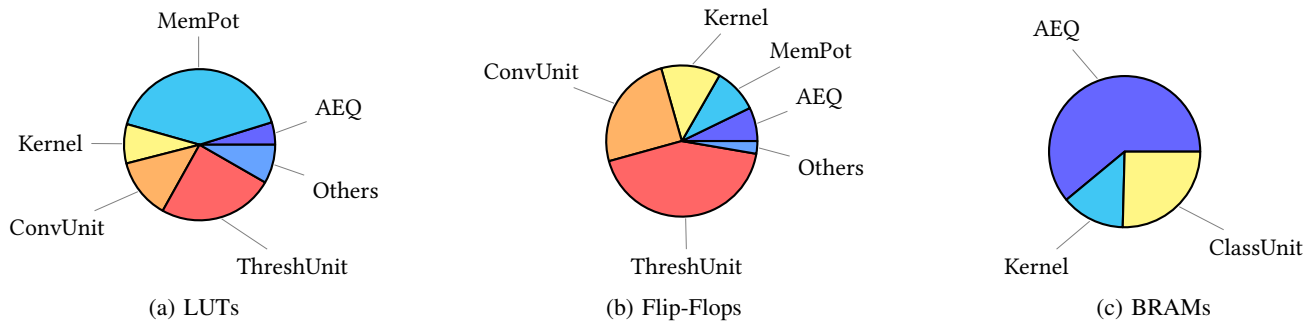


Fig. 12: Utilization of the different FPGA resources. “Others” includes the control unit, the classification unit and the bias ROM. Due to space limitations, units such as the classification unit that implements the final fully connected layer could not be described here. Note that the MemPot memory rows were too small to map efficiently to BRAM, so they were implemented as distributed LUT-RAM.

TABLE I: Performance of different degrees of parallelism, here for the 8 bit implementation. Here, parallelism refers to the number of parallel convolution cores, AEQs, thresholding units, etc.

Parallelization	$\times 1$	$\times 2$	$\times 4$	$\times 8$	$\times 16$
Throughput [FPS]	3,077	5,908	10,987	21,446	33,292
Efficiency [FPS/W]	3,149	5,006	7,474	10,163	9,148

TABLE II: FPGA Synthesis Results, compared to other FPGA-based SNN implementations.

	Frequency [MHz]	LUT	FF	BRAM [Mb]	DSP
This work (8 bit)	333	19 k	12 k	2.1	32
This work (16 bit)	333	33 k	21 k	3.9	64
Fang et al. [8]	125	115 k	233 k	9.1	1.7 k
Guo et al. [10]	100	53 k	100 k	2.3	
SIES [18]	200	302 k	421 k	6.9	

TABLE III: Sparsity of each layers input activations compared to the PE utilization for each convolutional layer. Here for the very first sample of the MNIST validation dataset.

Convolutional Layer	Layer 1	Layer 2	Layer 3
Input activation sparsity	93%	98%	98%
PE utilization	72%	58%	56%

TABLE IV: Accuracy on the Fashion-MNIST dataset compared to other works.

Work	This work	Guo et al. [10]	Fang et al. [8]
Accuracy [%]	88.9	87.5	89.2
Quantization [bits]	16	32	16

To reduce the memory cost of the membrane potentials, a scheme for neuron multiplexing was introduced. In this scheme, only a small part of the SNN is simulated and only the sparse output spikes of this partial simulation are stored. To achieve a high performance, a novel memory distribution scheme called memory interlacing was introduced. Memory interlacing allows for a highly-parallel fine-grained distribution of memory units close to the PEs. To exploit the high degree of sparsity, the spike events are compressed into queues. As a result, the processing time scales with the number of occurring spikes. Queue-based processing ensures that PEs are utilized as much as possible, even though there still is room to further improve the PE utilization. We found that in our

CSNN, there were multiple channels inside the convolutional layers that never generated spikes. Thus, pruning such “dead” layers could lead to further improvements. The PEs are highly pipelined to achieve a high clock frequency and to improve parallelism. In contrast HLS-based approaches like the one of Fang et al. [8] or S2N2 [39], our approach is agnostic to the CSNN’s architecture and can thus be implemented on an ASIC as well. As the hardware utilization of a single convolution unit is so small, multiple convolution units can be implemented in parallel, allowing easy scaling of throughput. The prototype developed in this work shows a very promising performance, however, the target SNN used here is relatively small, developed for a relatively simple benchmarking dataset. In the future we plan to implement larger SNNs and also compare our results to non-spiking implementations.

ACKNOWLEDGMENTS

REFERENCES

- [1] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0893608014002135>
- [2] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: <https://doi.org/10.1038/nature14539>
- [3] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for machine learning: Challenges and opportunities,” pp. 1–8, 2017. [Online]. Available: <http://arxiv.org/pdf/1612.07625v5>
- [4] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, “Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 15–28.
- [5] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [6] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, “Spiking neural networks hardware implementations and challenges,” *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, no. 2, pp. 1–35, 2019.
- [7] H. Mostafa, B. U. Pedroni, S. Sheik, and G. Cauwenberghs, “Fast classification using sparsely active spiking networks,” in *IEEE International Symposium 2017*, 2017, pp. 1–4.
- [8] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu, “Encoding, model, and architecture systematic optimization for spiking neural network in fpgas,” in *Proceedings of the 39th International Conference on Computer-Aided Design*, Y. Xie, Ed. New York, NY, USA: ACM, 2020, pp. 1–9.
- [9] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

TABLE V: Performance comparison for different computing platforms on the MNIST dataset, sorted by efficiency [FPS/W] in descending order.

	Type	Quantization [bits]	Throughput [FPS]	Latency [ms]	Power [W]	Efficiency [FPS/W]	Accuracy [%]
This work	FPGA	8	21 k	0.04	2.1	10163	98.3
This work	FPGA	16	21 k	0.04	2.9	7208	98.2
Fang et al. ^a [8]	FPGA	16	2124	0.52	4.5	471	99.2
Loihi ^a [9]	ASIC		671	1.5	3.8	178	98.0
Jetson ^a	SoC		211	75.8	14.0	15	99.2
RTX 5000 ^a	GPU		864	18.5	61.2	14	99.2
Guo et al. [10]	FPGA	32			0.7		98.9
ASIE [19]	ASIC				0.001		98.0
SIES [18]	FPGA						99.2
S2N2 [39]	FPGA						98.5

^a Performance results taken from [8].

- [10] S. Guo, L. Wang, S. Wang, Y. Deng, Z. Yang, S. Li, Z. Xie, and Q. Dou, "A systolic snn inference accelerator and its co-optimized software framework," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '19, New York, NY, USA: Association for Computing Machinery, 2019, p. 63–68. [Online]. Available: <https://doi.org/10.1145/3299874.3317966>
- [11] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. 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 Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.
- [12] S. Narayanan, K. Taht, R. Balasubramonian, E. Giacomini, and P.-E. Gaillardon, "Spinalflow: An architecture and dataflow tailored for spiking neural networks," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 349–362.
- [13] A. Yousefzadeh, T. Serrano-Gotarredona, B. Linares-Barranco, M. A. Khoei, S. Hosseini, P. Holanda, S. Leroux, O. Moreira, J. Tapson, B. Dhoedt, and P. Simoens, "Asynchronous spiking neurons, the natural key to exploit temporal sparsity," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 668–678, 2019.
- [14] B. Rueckauer and S.-C. Liu, "Conversion of analog to spiking neural networks using sparse temporal coding," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2018, pp. 1–5.
- [15] S. Thorpe, A. Delorme, and R. van Rullen, "Spike-based strategies for rapid processing," *Neural Networks*, vol. 14, no. 6-7, pp. 715–725, 2001.
- [16] W. Guo, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems," *Frontiers in neuroscience*, vol. 15, p. 638474, 2021.
- [17] J. H. Shin, A. Shafiee, A. Pedram, H. Abdel-Aziz, L. Li, and J. Hassoun, "Griffin: Rethinking sparse optimization for deep learning architectures," 2021.
- [18] S.-Q. Wang, L. Wang, Y. Deng, Z.-J. Yang, S.-S. Guo, Z.-Y. Kang, Y.-F. Guo, and W.-X. Xu, "Sies: A novel implementation of spiking convolutional neural network inference engine on field-programmable gate array," *Journal of Computer Science and Technology*, vol. 35, no. 2, pp. 475–489, 2020.
- [19] Z. Kang, L. Wang, S. Guo, R. Gong, S. Li, Y. Deng, and W. Xu, "Asie," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 16, no. 4, pp. 1–22, 2020.
- [20] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
- [21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.
- [22] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE transactions on neural networks*, vol. 15, no. 5, pp. 1063–1070, 2004.
- [23] A. Sengupta, Y. Ye, R. Wang, C. Liu, and K. Roy, "Going deeper in spiking neural networks: Vgg and residual architectures," *Frontiers in neuroscience*, vol. 13, p. 95, 2019.
- [24] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in neuroscience*, vol. 11, p. 682, 2017.
- [25] C. Lee, S. S. Sarwar, P. Panda, G. Srinivasan, and K. Roy, "Enabling spike-based backpropagation for training deep neural network architectures," *Frontiers in neuroscience*, vol. 14, p. 119, 2020.
- [26] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.
- [27] S. Panzeri, N. Brunel, N. K. Logothetis, and C. Kayser, "Sensory neural codes using multiplexed temporal scales," *Trends in neurosciences*, vol. 33, no. 3, pp. 111–120, 2010.
- [28] B. Han and K. Roy, "Deep spiking neural network: Energy efficiency through time based coding," in *Computer Vision – ECCV 2020*, ser. Lecture Notes in Computer Science, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds. Cham: Springer International Publishing, 2020, vol. 12355, pp. 388–404.
- [29] B. Rueckauer and S.-C. Liu, "Temporal pattern coding in deep spiking neural networks," in *2021 International Joint Conference on Neural Networks (IJCNN)*, July 2021, pp. 1–8.
- [30] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural networks : the official journal of the International Neural Network Society*, vol. 111, pp. 47–63, 2019.
- [31] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, 2019.
- [32] J. Wu, Y. Chua, M. Zhang, G. Li, H. Li, and K. C. Tan, "A tandem learning rule for effective training and rapid inference of deep spiking neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021.
- [33] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 2818–2826. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.308>
- [34] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.
- [36] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: <http://arxiv.org/abs/1704.04861>
- [37] R. Tapiador-Morales, A. Linares-Barranco, A. Jimenez-Fernandez, and G. Jimenez-Moreno, "Neuromorphic lif row-by-row multiconvolution processor for fpga," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 1, pp. 159–169, 2019.
- [38] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference." [Online]. Available: <http://arxiv.org/pdf/1712.05877v1>
- [39] A. Khodamoradi, K. Denolf, and R. Kastner, *S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks*. New York, NY, USA: Association for Computing Machinery, 2021, p. 194–205. [Online]. Available: <https://doi.org/10.1145/3431920.3439283>