



ARTICLE

Resource-Aware Co-Design of Spiking Neural Networks for Intelligent Structural Health Monitoring on Edge FPGAs

Van-Hai Pham^{1,2}, Quang-Thai Pham¹, Van-Vu Luyen¹, Duy-Anh Nguyen^{1,*} and Thanh-Toan Dao¹

¹Department of Electronic and Computer Engineering, University of Transport and Communications, No.3 Cau Giay Street, Hanoi, Vietnam

²Faculty of Electrical and Electronics Engineering, Hanoi Open University, B101, Nguyen Hien Street, Hanoi, Vietnam

*Corresponding Author: Duy-Anh Nguyen. Email: anhnd@utc.edu.vn

Received: 04 June 2026; Accepted: 18 August 2026; Published: 22 October 2026

ABSTRACT: Deploying deep Spiking Neural Networks (SNNs) on resource-constrained Field-Programmable Gate Arrays (FPGAs) for Structural Health Monitoring (SHM) is hindered by the persistent storage of neuronal membrane potentials across timesteps, creating a substantial on-chip memory requirement that standard Convolutional Neural Network (CNN) acceleration techniques do not directly address. We propose a Hardware-Software Co-Design Framework that enables deployment of a Spiking ResNet-18 for automated crack detection on a Xilinx KV260 edge FPGA. The framework formalizes the Membrane Memory Floor (MMF) as a lower bound on on-chip memory for the target deployment architecture, introduces Proportional Channel Scaling (PCS) for systematic topology reduction, applies Memory Type Selection (MTS) to optimally bind state arrays to heterogeneous BlockRAM (BRAM) and UltraRAM (URAM) resources, and develops Channel Folding, a novel High-Level Synthesis (HLS) architecture reducing intermediate buffers by 4×. Evaluated on the SDNET2018 dataset, the framework achieves $85.89 \pm 0.40\%$ accuracy while reducing BRAM utilization by 71.7% and DSP usage by 87%, fitting a deep residual SNN onto a platform where the baseline requires 300% of available memory. Ablation analysis confirms that only the combined application of all techniques satisfies the strict resource constraints. Board-level measurements on the deployed KV260 platform report a dynamic power consumption of only 0.321 W during active SNN inference, demonstrating the suitability of the proposed framework for low-power autonomous SHM at the network edge.

KEYWORDS: Spiking neural network; FPGA; high-level synthesis; edge AI; memory optimization; structural health monitoring; crack detection

1 Introduction

The SHM of civil infrastructure, including bridges, dams, tunnels, and nuclear containment structures, is a critical discipline for ensuring structural safety and enabling cost-effective maintenance of aging infrastructure [1]. Among structural defects, concrete cracking is widely regarded as a primary visual indicator of structural deterioration, making early detection essential for preventing progressive damage [2]. Traditional inspection relies on trained engineers performing periodic visual assessments, a process that is subjective, labor-intensive, and often hazardous in locations such as bridge undersides or post-earthquake structures [3]. Consequently, there is a growing need for autonomous edge intelligence capable of performing continuous, on-site crack detection without reliance on cloud connectivity, which may be unreliable in remote infrastructure environments.

Deep learning, particularly CNNs, has achieved strong performance in automated crack detection [3,4]. Architectures such as ResNet-18 and VGG-16 have demonstrated high detection accuracy on benchmark datasets such as SDNET2018 [5]. However, deploying these models on edge devices faces two critical barriers: (i) high computational complexity for deep CNN architectures such as ResNet-18 and VGG-16 [6], leading to significant power consumption for battery-powered field sensors, and (ii) reliance on cloud-based processing pipelines, which are impractical for remote bridge or dam inspection sites [7,8]. Consequently, SNNs are well suited to long-term edge SHM deployments, where energy efficiency and on-device inference are often more critical than maximizing inference throughput.

The SNNs offer a bio-inspired alternative uniquely suited to SHM edge deployment [9,10]. Unlike CNNs that transmit continuous-valued activations, SNN neurons communicate via discrete binary spikes over time. This event-driven computation exploits data sparsity—particularly relevant for crack detection, where crack textures are sparse against otherwise homogeneous concrete backgrounds—potentially reducing energy consumption on dedicated neuromorphic hardware [11]. Although event-driven computation can exploit the sparse activation patterns produced by crack images, FPGA studies have shown that the resulting energy benefits are highly dependent on the achievable spike sparsity and the underlying hardware implementation [12]. Recent advances in surrogate gradient training have enabled deep Spiking ResNets achieving accuracy comparable to their ANN counterparts [13,14].

FPGAs provide an attractive deployment platform for SNN-based SHM due to their hardware reconfigurability, high parallelism, and energy-efficient inference capabilities [15]. Recent surveys further identify FPGA acceleration as a promising platform for SNN deployment while highlighting the remaining architectural and implementation challenges [16]. Low-cost edge modules such as the Xilinx Kria KV260 (<5 W typical power) can be solar-powered at remote infrastructure sites, enabling continuous autonomous monitoring. However, a critical architectural mismatch remains. While deep SNN models such as Spiking ResNet-18 are necessary for robust crack feature extraction, they introduce a unique memory challenge: neuron membrane potentials (V_{mem}) must persist across multiple timesteps [11]. Unlike CNN activations, which are transient and can be streamed or tiled during inference, SNN membrane states form a persistent on-chip state that scales with neuron count and simulation length, significantly increasing memory pressure for FPGA implementations [11].

Deploying a deep Spiking ResNet-18 for concrete crack detection on a resource-constrained edge FPGA such as the Xilinx KV260 remains challenging due to the limited on-chip memory capacity. In particular, the requirement to maintain neuron membrane potentials across multiple timesteps creates a persistent state that significantly increases the memory footprint of SNN inference. Addressing this limitation is critical for enabling low-power, autonomous SHM at the network edge, reducing reliance on cloud-based processing and supporting continuous structural monitoring. By enabling the complete deployment of a deep SNN on a commodity edge FPGA, the proposed framework supports practical SHM systems that can perform crack detection directly at monitoring sites without cloud connectivity. This deployment model is well suited to long-term structural monitoring, where fixed sensing nodes operate continuously under limited power and communication resources.

1.1 Problem Motivation

Deploying deep SNNs on resource-constrained edge FPGAs remains challenging because neuron membrane potentials must be maintained across multiple simulation timesteps, resulting in substantially higher on-chip memory requirements than conventional CNNs [11]. More broadly, recent neuromorphic hardware reviews have emphasized that achieving competitive edge performance requires carefully balancing computational efficiency, memory organization, and architectural flexibility through hardware–algorithm co-design [17].

HLS is widely used for FPGA-based neural network design, enabling automatic generation of pipelined and dataflow architectures. However, HLS introduces additional intermediate buffers, such as feature maps and streaming FIFOs, which significantly increase on-chip memory usage. This overhead becomes critical in SNNs, where neuron states must be maintained across timesteps.

Unlike CNNs, whose intermediate activations are transient, SNNs require persistent membrane states throughout temporal inference. These states are updated in place across timesteps, so the on-chip memory requirement is determined by the number of neurons and the membrane-state bitwidth, rather than by the simulation length [11].

This challenge becomes critical on FPGA platforms with limited on-chip memory. On commodity edge FPGA platforms, the available on-chip memory is often insufficient to simultaneously accommodate persistent membrane states and HLS-generated intermediate buffers for deep SNN inference. Furthermore, SNN topology and HLS-generated hardware structures are strongly coupled, as architectural choices directly influence the size of feature maps, streaming buffers, and persistent neuron states. Therefore, optimizing the neural network and hardware independently often leads to inefficient on-chip memory utilization.

1.2 Our Novel Contributions

To address the above challenges, we propose a hardware-software co-design framework for deploying deep SNNs on edge FPGAs. The overall concept of the proposed framework is shown in Fig. 1. Starting from a baseline SNN model and FPGA resource constraints, the framework progressively transforms the original network into an FPGA-deployable implementation through software-level and hardware-level optimizations.

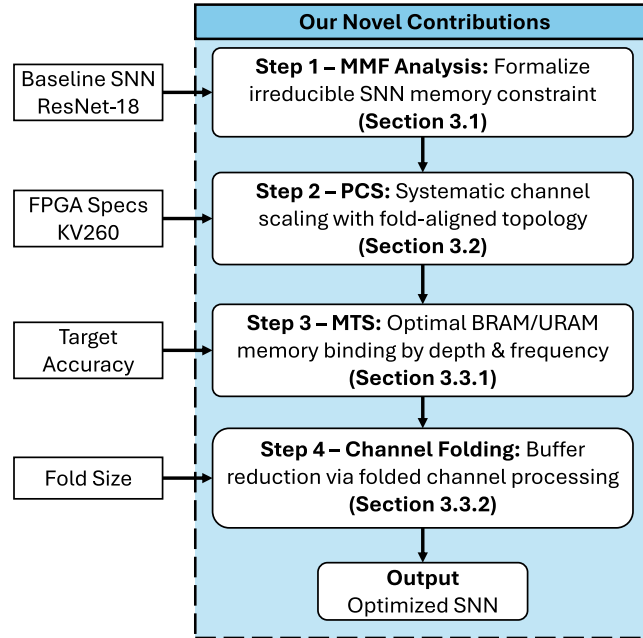


Figure 1: Overview of the proposed hardware-software co-design framework. Starting from a baseline SNN and FPGA constraints, the framework produces an optimized FPGA-deployable model through four stages.

The detailed workflow of the proposed framework is illustrated in Fig. 2. The framework consists of four key mechanisms:

- The Membrane Memory Floor (MMF) formalizes the persistent on-chip memory requirement of deep SNNs under the target FPGA dataflow schedule, providing a basis for analyzing the membrane-state storage bottleneck.
- Proportional Channel Scaling (PCS) systematically adjusts the network topology to satisfy FPGA memory constraints while preserving classification accuracy and maintaining hardware-compatible, fold-aligned channel counts.
- Memory-Type Selection (MTS) maps neuron state arrays to Block RAM (BRAM) or UltraRAM (URAM) according to array depth and access characteristics, improving the utilization balance between available on-chip memory resources.
- Channel Folding processes output channels in groups of size F , reducing intermediate buffer memory by approximately C_{out}/F and lowering DSP utilization through folded computation.

This work demonstrates the practical viability of the proposed framework through the deployment of a deep Spiking ResNet-18 for automated crack detection on a memory-constrained edge FPGA. The hardware-software co-design approach mitigates the persistent membrane-storage bottleneck and significantly reduces on-chip resource requirements while maintaining reliable classification performance.

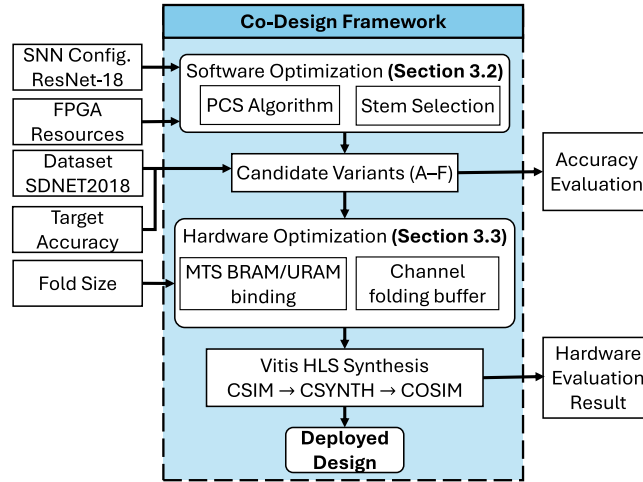


Figure 2: Detailed workflow of the proposed co-design framework. Software optimization generates candidate SNN variants, while hardware optimization maps the selected model to FPGA resources through iterative refinement.

2 Background and Related Work

2.1 SHM and Edge AI

SHM involves continuous observation of infrastructure using sensor data to detect structural damage and assess integrity [1]. For concrete structures, surface crack detection is a widely used non-destructive evaluation technique. The SDNET2018 dataset [5] provides a standard benchmark containing over 56,000 labeled 256×256 images of bridge decks, walls, and pavements with diverse crack patterns under varying surface and lighting conditions.

Deploying deep learning models for SHM in real-world environments imposes strict edge constraints. Edge devices must operate within a power budget below 5 W, process images locally without reliable cloud connectivity, and tolerate harsh environmental conditions. FPGAs are well suited to these requirements

due to their reconfigurable architecture and deterministic latency. Platforms such as the Xilinx Kria KV260 provide compact edge computing capability with system power below 5 W. However, their limited on-chip memory (648 KB BRAM and approximately 2.3 MB URAM) poses a significant challenge for deploying deep neural networks.

2.2 SNNs and the IF Neuron Model

SNNs represent the third generation of neural networks, processing information through discrete binary spikes over time rather than continuous activations [9]. Among various neuron models, the Integrate-and-Fire (IF) neuron is widely used in deep SNN implementations due to its hardware simplicity, requiring only addition and comparison operations without exponential decay computation [10]. The membrane potential of an IF neuron evolves as Eq. (1):

$$V_{\text{mem}}^{(l)}[t] = V_{\text{mem}}^{(l)}[t-1] + \sum_j w_j^{(l)} S_j^{(l-1)}[t], \quad (1)$$

where $w_j^{(l)}$ denotes the synaptic weight, $S_j^{(l-1)}[t] \in \{0, 1\}$ is the pre-synaptic spike at timestep t , and $V_{\text{mem}}^{(l)}$ is the membrane potential. A spike is generated when the membrane potential reaches the firing threshold:

$$S^{(l)}[t] = \begin{cases} 1, & V_{\text{mem}}^{(l)}[t] \geq V_{th}, \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

After spike generation, the membrane potential is reset according to

$$V_{\text{mem}}^{(l)}[t] = \begin{cases} V_{\text{reset}}, & S^{(l)}[t] = 1, \\ V_{\text{mem}}^{(l)}[t], & \text{otherwise.} \end{cases} \quad (3)$$

In this work, a hard-reset IF neuron is adopted with $V_{th} = 1$ and $V_{\text{reset}} = 0$. Because the membrane potential must be updated and stored across all T timesteps, V_{mem} becomes a persistent state for every neuron during inference, which is the primary source of the FPGA memory challenge.

For crack detection in SHM, the IF model is suitable because crack structures generate sparse spike activity against largely homogeneous concrete backgrounds, enabling event-driven computation and efficient hardware implementation.

The advantages of spike sparsity extend beyond inference. Recent sparsity-aware SNN accelerators have demonstrated that exploiting sparse spike activity significantly improves the energy efficiency of BPTT-based training [18]. This characteristic is particularly attractive for crack detection, where crack patterns typically occupy only a small portion of otherwise homogeneous concrete surfaces.

Training deep SNNs requires surrogate gradient methods to handle the non-differentiability of the spike function [13,14]. In this work, we employ the SpikingJelly framework [19], which implements surrogate-gradient backpropagation-through-time (BPTT) for training deep Spiking ResNets.

2.3 Spiking ResNet-18 Architecture

The Spiking ResNet-18 adapts the standard ResNet-18 architecture [6] by replacing ReLU activations with IF spiking neurons and fusing Batch Normalization (BN) parameters into the preceding convolution weights for hardware efficiency. The network retains the residual structure of ResNet-18 and consists of a stem layer followed by four stages of residual BasicBlocks.

The stem layer processes the RGB input ($256 \times 256 \times 3$) using a $\text{Conv}7 \times 7$ layer with stride 2 followed by max pooling, producing the largest intermediate feature map in the network. On FPGA platforms, this tensor often determines the peak on-chip buffer requirement. The network then proceeds through four stages, each containing two BasicBlocks composed of 3×3 convolutions with IF neurons and residual shortcuts. Between stages, the spatial resolution is reduced by stride-2 downsampling while the channel dimension increases, forming a hierarchical feature representation. Consequently, the size of intermediate feature maps ($C \times H \times W$) decreases with depth, whereas the convolution parameter count grows with channel width ($C_{in} \times C_{out} \times K^2$).

After the final stage, global average pooling reduces the feature map to a C_4 -dimensional vector, which is mapped to the crack/no-crack classes by a lightweight fully connected layer. The key challenge for FPGA deployment arises from the temporal nature of SNN inference: every IF neuron must maintain its membrane potential V_{mem} across T timesteps. Unlike conventional CNN activations that can be discarded after layer computation, these membrane states must persist on-chip throughout the inference process, forming the MMF that constrains FPGA deployment.

2.4 FPGA Memory Architecture

Modern Xilinx UltraScale+ FPGAs provide a heterogeneous on-chip memory hierarchy consisting primarily of Block RAM (BRAM) and UltraRAM (URAM). The key memory resources available on the Kria KV260 platform are summarized in [Table 1](#).

Table 1: Memory resources of the Kria KV260 platform.

Memory	Blocks	Total Capacity	Latency	Granularity
BRAM_18K	288	648 KB	1 cycle	Flexible
URAM	64	2304 KB	Typically 2–3 cycles	4096-depth blocks

BRAMs are flexible dual-port memories suitable for high-throughput streaming buffers such as convolution outputs and intermediate feature maps. In contrast, URAMs provide significantly higher capacity but are most efficient for deep arrays due to their fixed 4K-depth granularity and slightly higher access latency. For SNN inference, membrane potential arrays exhibit widely varying sizes and access patterns across network stages. Therefore, efficient utilization of this heterogeneous memory hierarchy requires selecting the memory type according to both array depth and access frequency, rather than relying solely on the network stage or spatial resolution. This is particularly important for FPGA deployment, as membrane state arrays can range from several thousand to over 65,000 neurons depending on the layer configuration.

2.5 Related Work

Deep learning methods, particularly CNNs, have demonstrated strong performance in automated crack detection for SHM. Early work by Cha et al. [3] applied CNNs for crack detection, while Dung and Anh [4] extended this approach to pixel-level segmentation using fully convolutional networks (FCNs). Islam and Kim [20] further proposed an encoder-decoder architecture based on a VGG backbone for crack segmentation. Despite their high accuracy, these approaches typically rely on computationally intensive CNN backbones such as VGG-16 or ResNet, limiting their deployment on resource-constrained edge platforms. SNNs offer a promising alternative through sparse, event-driven computation that can reduce computational cost and energy consumption.

Beyond vision-based CNN/SNN approaches, the SHM community has also advanced through machine-learning- and optimization-based damage identification frameworks. These approaches have primarily focused on damage identification and prediction accuracy, while recent reviews have summarized the broader integration of machine learning and deep learning in SHM [21]. Unlike these studies, which primarily focus on damage identification algorithms and prediction accuracy, the present work addresses the practical deployment of deep SNNs on commodity edge FPGAs under strict memory constraints. However, to the best of our knowledge, prior work has not explored deploying deep SNN models for SHM on edge FPGA platforms.

Quantization techniques reduce the bitwidth of weights and neuron states, thereby reducing memory requirements. Pruning methods reduce the number of network parameters and can further reduce the resource requirements of SNN models. ANN-to-SNN conversion techniques [22–24] enable deep architectures but typically require long simulation windows (T) to maintain accuracy. However, these methods primarily target precision through quantization, rather than the on-chip memory allocation and buffer management challenges specific to HLS-based FPGA implementations.

Early FPGA-based SNN accelerators such as Minitaur [25] were limited to relatively shallow network architectures. Frameworks like FINN [26] enable efficient FPGA deployment of quantized neural networks but are designed for conventional ANN models with transient activations. Several FPGA SNN implementations store membrane states in off-chip memory, introducing additional latency that is undesirable for edge SHM applications. Custom neuromorphic processors such as Loihi [27] and TrueNorth [28] provide specialized hardware support for SNNs but remain relatively inaccessible for general edge deployment.

More recent FPGA-SNN frameworks such as Spiker [29], E3NE [30], and SpikExplorer [31] have advanced FPGA deployment through improved accelerator architectures, compiler-assisted implementation, and automated hardware design-space exploration. However, these works primarily focus on improving inference efficiency and hardware optimization, while leaving the persistent membrane-state memory overhead of deep residual SNNs on resource-constrained edge FPGAs largely unaddressed.

Recent studies have also explored FPGA-based implementations of neuron models to improve hardware efficiency and computational performance.

No existing work addresses the persistent memory requirement associated with SNN membrane states in the context of deep residual networks on commodity edge FPGAs for SHM applications. Standard CNN tiling fails because membrane states cannot be discarded. Our framework addresses this gap through systematic co-design of network topology and HLS architecture, specifically targeting the practical requirements of autonomous structural monitoring.

2.6 Problem Formulation

Deploying deep SNNs on edge FPGAs is primarily constrained by the persistent storage of neuron membrane potentials. Unlike activations in conventional CNN accelerators, which are transient, membrane states must be retained across all T simulation timesteps, creating a significant on-chip memory demand under the target FPGA deployment architecture.

We formalize this requirement as the MMF, defined as the memory required to store all neuron membrane states maintained by the target deployment schedule:

$$M_{\text{floor}} = \sum_{l=1}^L N_l \cdot \frac{B_{\text{mem}}}{8}, \quad (4)$$

where L is the number of neuron layers, $N_l = C_l \times H_l \times W_l$ denotes the neuron count of layer l , and B_{mem} is the membrane bitwidth. The lower-bound property of M_{floor} follows directly from the recursive membrane dynamics of the IF neuron model in Eq. (1). Since $V_{\text{mem}}[t]$ depends on $V_{\text{mem}}[t-1]$, each neuron must retain its membrane state across timesteps. Consequently, techniques such as spatial tiling, recomputation, or streaming cannot eliminate this storage requirement. Therefore, M_{floor} represents the membrane-state memory floor considered under the target deployment architecture.

We represent the target SNN as $\mathcal{N} = (\mathbf{C}, S_{\text{stem}}, T, B_{\text{mem}})$, where $\mathbf{C} = [C_1, C_2, \dots, C_S]$ denotes the channel configuration of the S network stages, S_{stem} is the stem stride, T is the number of timesteps, and B_{mem} is the membrane bitwidth.

Given a target FPGA with limited resources $\mathcal{R} = (R_{\text{BRAM}}, R_{\text{URAM}}, R_{\text{DSP}})$, our goal is to minimize on-chip BRAM usage while maintaining model accuracy. To this end, we jointly optimize three design variables: the scaled channel configuration $\mathbf{C}'$, the channel folding factor F , and the memory allocation function $\mathcal{T}$ that maps arrays to either BRAM or URAM.

The resulting co-design optimization problem is

$$\begin{aligned} \min_{\mathbf{C}', F, \mathcal{T}} \quad & \text{BRAM}(\mathbf{C}', F, \mathcal{T}) \\ \text{s.t.} \quad & \text{BRAM} \leq R_{\text{BRAM}}, \\ & \text{URAM} \leq R_{\text{URAM}}, \\ & \text{Acc}(\mathbf{C}') \geq A_{\text{min}}. \end{aligned} \tag{5}$$

Here, BRAM denotes the total on-chip BRAM consumption of the design, including both membrane arrays and intermediate convolution buffers. The optimization variables are addressed by the three techniques proposed in Section 3: PCS, MTS, and Channel Folding.

2.6.1 Example MMF Calculation

To illustrate how Eq. (4) is computed in practice, we provide an example calculation for the scaled ResNet-18 configuration used in our implementation (channels [16, 24, 48, 96]). For each layer, the number of neurons is given by $N_l = C_l \times H_l \times W_l$, and multiple IF neuron arrays exist per residual block. The resulting membrane arrays and neuron counts are summarized in Table 2.

Table 2: Example calculation of the MMF.

Stage	C_l	$H_l \times W_l$	Arrays	N_l
Stem	16	64×64	1	65,536
Stage 1	16	64×64	4	262,144
Stage 2	24	32×32	4	98,304
Stage 3	48	16×16	4	49,152
Stage 4	96	8×8	4	24,576
Total	–	–	17	499,712

Note: Bold values indicate the total number of membrane arrays and neurons used in the MMF calculation.

For INT8 membrane storage ($B_{\text{mem}} = 8$), the resulting memory requirement is

$$M_{\text{floor}}^{\text{INT8}} = 499,712 \times \frac{8}{8} = 499,712 \text{ bytes} \approx 488 \text{ KB.} \quad (6)$$

This example shows that membrane states alone occupy a large fraction of on-chip memory, motivating the memory-aware co-design strategy proposed in this work.

3 The Proposed Co-Design Framework

3.1 Software-Level Optimization: PCS

As defined in Eq. (4), the MMF (M_{floor}) is proportional to the total number of neurons in the network. Since the neuron count of each layer is given by $N_l = C_l \times H_l \times W_l$, the memory requirement scales linearly with the channel width C_l for a fixed spatial resolution. This observation motivates channel scaling as an effective mechanism to reduce the memory footprint of deep SNNs.

To adapt the network to FPGA memory constraints, we propose PCS, a memory-aware channel scaling strategy. Unlike uniform scaling, PCS applies distinct scaling factors depending on the memory type used for each stage. Early stages are mapped to BRAM to ensure low-latency access, while later stages utilize URAM to exploit its larger storage capacity.

3.1.1 Scaling Factors and Channel Alignment

The early stages of the network are implemented using BRAM in order to avoid access bottlenecks. To derive the scaling factor rigorously, we formulate it as a constrained optimization problem based on the available hardware resources. Let $M_{\text{BRAM}}^{\text{avail}}$ denote the available BRAM capacity after reserving space for system logic, and let M_{buffers} represent the intermediate fold buffers required by the channel folding datapath.

Assume that N_{arrays} membrane arrays are mapped to BRAM, all sharing the spatial resolution of the stem feature map $H_{\text{stem}} \times W_{\text{stem}}$. This assumption holds for the deployed ResNet-18 architecture, where the BRAM-resident membrane arrays correspond to the stem and Stage 1 layers. Since these arrays correspond to the stem and Stage 1 layers, which share the same baseline channel width C_1^{base} , we parameterize the scaled channel width as $\alpha_{\text{BRAM}} \times C_1^{\text{base}}$. To prevent pipeline stalls, the total membrane memory footprint of these arrays must strictly satisfy the on-chip BRAM constraint limit:

$$N_{\text{arrays}} \times (\alpha_{\text{BRAM}} \times C_1^{\text{base}}) \times H_{\text{stem}} \times W_{\text{stem}} \times \frac{B_{\text{mem}}}{8} \leq M_{\text{BRAM}}^{\text{avail}} - M_{\text{buffers}}. \quad (7)$$

By solving this inequality for the upper bound of the scaling factor that maximizes network capacity, the BRAM scaling factor is derived as:

$$\alpha_{\text{BRAM}} = \frac{M_{\text{BRAM}}^{\text{avail}} - M_{\text{buffers}}}{N_{\text{arrays}} \times C_1^{\text{base}} \times H_{\text{stem}} \times W_{\text{stem}} \times (B_{\text{mem}}/8)}, \quad (8)$$

where B_{mem} denotes the membrane-state precision in bits. In this work, INT8 membrane quantization is employed, corresponding to $B_{\text{mem}} = 8$.

Although URAM provides significantly larger storage capacity than BRAM, its multi-cycle access latency makes excessive reliance on URAM undesirable for deeply pipelined SNN inference. Therefore, PCS introduces an additional reduction factor $\beta < 1$ for the URAM-resident stages, allowing later network stages

to be scaled more conservatively than the BRAM-resident stages while exploiting the larger storage capacity of URAM. The corresponding URAM scaling factor is defined as

$$\alpha_{\text{URAM}} = \alpha_{\text{BRAM}} \times \beta. \quad (9)$$

To ensure hardware compatibility with the Channel Folding architecture, which processes channels in groups of size F , as detailed in Section 3.2.2, the scaled channels must be aligned to multiples of the fold size F . The final channel configuration for each stage is therefore computed as

$$C'_s = F \times \text{Round}\left(\frac{\alpha_s \times C_s^{\text{base}}}{F}\right), \quad \forall s \in \{1, \dots, S\}, \quad (10)$$

where $\alpha_s \in \{\alpha_{\text{BRAM}}, \alpha_{\text{URAM}}\}$ depending on the memory assignment of stage s . The complete PCS procedure is summarized in Algorithm 1.

Algorithm 1: Memory-aware PCS

Require: Baseline channels $\mathbf{C} = [C_1, \dots, C_S]$; BRAM budget $M_{\text{BRAM}}^{\text{avail}}$; buffer overhead M_{buffers} ; BRAM arrays N_{arrays} ; fold factor F ; stem resolution $H_{\text{stem}}, W_{\text{stem}}$; membrane precision B_{mem} , URAM reduction factor β

- 1: Compute α_{BRAM} using Eq. (8)
- 2: Compute $\alpha_{\text{URAM}} \leftarrow \alpha_{\text{BRAM}} \times \beta$
- 3: **for** each stage $s = 1$ to S **do**
- 4: **if** stage s uses BRAM **then**
- 5: $\alpha_s \leftarrow \alpha_{\text{BRAM}}$
- 6: **else**
- 7: $\alpha_s \leftarrow \alpha_{\text{URAM}}$
- 8: **end if**
- 9: $C'_s \leftarrow \max(F, F \times \text{Round}(\alpha_s \times C_s / F))$
- 10: **end for**
- 11: **return** Scaled topology $\mathbf{C}' = [C'_1, \dots, C'_S]$

PCS is intended as a practical design-space exploration heuristic rather than a globally optimal solver. The buffer overhead M_{buffers} is estimated from the target folding architecture and may be refined iteratively after fold-factor selection. For the deployed ResNet-18 design, a buffer budget of 100 KB was used during topology exploration and subsequently validated by HLS synthesis.

PCS adapts the network channel configuration according to the memory constraints of the target FPGA. The scaling factors are computed from the available BRAM budget while accounting for the intermediate buffer overhead introduced by the Channel Folding architecture. The resulting channel configuration is further adjusted according to the BRAM/URAM memory assignment and aligned to multiples of the folding factor F , ensuring hardware compatibility before training. Consequently, PCS serves as the software-level optimization step of the proposed hardware-software co-design framework by adapting the network channel configuration to the available on-chip memory resources.

3.1.2 Selection of the URAM Reduction Factor

The reduction factor β controls the channel width of the URAM-resident stages and therefore directly affects the membrane memory footprint and hardware efficiency. Rather than being manually tuned, PCS determines its value through a constrained design-space exploration based on the target FPGA architecture.

Specifically, a set of candidate values is evaluated under the BRAM/URAM resource constraints while preserving fold-aligned channel dimensions required by the Channel Folding architecture. Candidate configurations that violate the available on-chip memory budget are discarded. Among the remaining feasible configurations, the largest value of β is selected to maximize the network capacity while satisfying the hardware resource constraints. The impact of different β values on classification accuracy, BRAM/URAM utilization, execution latency is quantitatively evaluated in [Section 4.2](#), where the selected value ($\beta = 0.75$) is experimentally justified.

3.1.3 Example: PCS Calculation

To illustrate the PCS mechanism mathematically, consider scaling a baseline ResNet-18 configuration $\mathbf{C} = [64, 128, 256, 512]$ targeting the Xilinx ZU5 eV FPGA. Assume the usable BRAM budget is $M_{\text{BRAM}}^{\text{avail}} = 430$ KB, A buffer budget of $M_{\text{buffers}} = 100$ KB is reserved for the subsequent Channel Folding implementation, the stem spatial resolution is 64×64 , and $N_{\text{arrays}} = 5$ membrane arrays are mapped to BRAM. Using the selected fold factor $F = 4$ and the URAM reduction factor $\beta = 0.75$ determined according to the procedure described in [Section 3.1.2](#), we obtain

$$\alpha_{\text{BRAM}} = \frac{430,000 - 100,000}{5 \times 64 \times 64 \times 64} \approx 0.25, \quad (11)$$

$$\alpha_{\text{URAM}} = 0.25 \times 0.75 = 0.1875.$$

[Table 3](#) shows the step-by-step channel alignment, resulting in the final configuration $\mathbf{C}' = [16, 24, 48, 96]$, which satisfies the memory constraint while maintaining a manageable peak feature map size.

Table 3: Example calculation of PCS constraints.

Stage	$\mathbf{C}^{\text{base}}$	Memory	α	Alignment Equation	$\mathbf{C}'$
1	64	BRAM	0.25	$4 \times \text{Round}(0.25 \times 64/4)$	16
2	128	URAM	0.1875	$4 \times \text{Round}(0.1875 \times 128/4)$	24
3	256	URAM	0.1875	$4 \times \text{Round}(0.1875 \times 256/4)$	48
4	512	URAM	0.1875	$4 \times \text{Round}(0.1875 \times 512/4)$	96

3.1.4 Stem Type Co-Design

Besides channel scaling, the design of the stem layer strongly affects the peak feature map size and thus the on-chip memory footprint. [Table 4](#) compares three candidate stem configurations.

Table 4: Stem configuration comparison.

Type	Configuration	Output Spatial Size
Standard (Std)	Conv7 ($s = 2$) + Pool ($s = 2$)	64×64
Aggressive (Aggr)	Conv7 ($s = 4$)	64×64
Multi-conv (Multi)	2×Conv3 ($s = 2$)	64×64

Note: Std, Aggr, and Multi denote the standard stem, aggressive stem, and multi-convolution stem, respectively.

Among these options, the aggressive stem ($\text{Conv}7 \times 7$ with stride 4) is selected because it performs $4\times$ spatial downsampling in a single operation without requiring a MaxPool kernel. This simplifies the HLS control logic while still preserving sufficient texture information for crack detection tasks.

3.2 Hardware-Level Optimization

Although PCS resolves the network-level memory constraint, practical HLS synthesis still introduces additional on-chip memory overhead. To efficiently map the scaled SNN architecture onto the heterogeneous FPGA memory hierarchy, we introduce two hardware-oriented optimizations: MTS and Channel Folding.

3.2.1 MTS

Modern edge FPGAs provide heterogeneous on-chip memory resources, primarily BRAM and URAM, which differ significantly in latency and storage capacity. Efficient memory allocation is therefore essential because membrane arrays vary substantially in both size and access patterns across network stages. Similar BRAM/URAM binding strategies have been widely used in FPGA CNN accelerators such as [26]. In this work, we adapt the concept to persistent SNN membrane-state storage. The proposed MTS strategy is developed and evaluated on Xilinx UltraScale+ devices, where both BRAM and URAM resources are available.

Early layers (Stem and Stage 1) possess large spatial dimensions and are accessed continuously within the inner convolution pipeline. These arrays lie on the critical path and therefore require the single-cycle access latency of BRAM to prevent pipeline stalls. In contrast, deeper layers (Stages 2–4) have smaller spatial resolutions but larger channel depths, resulting in dense arrays that are better suited to the high-capacity URAM. Furthermore, URAM blocks in the Xilinx UltraScale+ architecture are physically organized with a minimum depth of 4096 entries. Using URAM for shallow arrays therefore leads to severe storage underutilization.

To formalize this allocation, let $\mathcal{A}_{\text{HF}}$ denote the set of memory arrays that lie on the critical path and require high-frequency, single-cycle access, e.g., arrays within the inner convolution pipeline. We define a binding function $\mathcal{T}(A_l)$ that maps the membrane array A_l at layer l , with depth $D_l = C_l \times H_l \times W_l$, to the most appropriate on-chip memory resource:

$$\mathcal{T}(A_l) = \begin{cases} \text{URAM}, & \text{if } D_l \geq 4096 \text{ and } A_l \notin \mathcal{A}_{\text{HF}}, \\ \text{BRAM}, & \text{otherwise.} \end{cases} \quad (12)$$

This formulation ensures that deep-stage membrane arrays are migrated to URAM, while BRAM is strictly reserved for latency-critical neuron states ($\mathcal{A}_{\text{HF}}$) and transient buffers. Table 5 demonstrates the MTS assignment for an example architecture.

This strategy migrates deep-stage membrane arrays to URAM while reserving BRAM for latency-critical neuron states and transient buffers, thereby significantly reducing overall BRAM pressure.

3.2.2 Channel Folding Architecture

Even with optimized memory allocation for persistent membrane states, conventional HLS convolution pipelines still require large intermediate buffers. A fully parallel implementation computes all C_{out} output channels simultaneously, requiring temporary storage of size $C_{\text{out}} \times H \times W$ to store convolution outputs and intermediate processing data.

Table 5: MTS assignment for an example architecture.

Array	Depth (D_I)	$A_I \in \mathcal{A}_{\text{HF}}$	Assign
mem_stem	65,536	Yes	BRAM
mem_s1_*	65,536	Yes	BRAM
mem_s2_*	24,576	No	URAM
mem_s3_*	12,288	No	URAM
mem_s4_*	6144	No	URAM
conv_fold	16,384	Yes	BRAM
bn_fold	16,384	Yes	BRAM

Note: The symbol * denotes all membrane arrays within the corresponding stage.

To reduce this transient memory overhead, we introduce the Channel Folding architecture. Instead of computing all output channels in parallel, the hardware processes them in sequential groups defined by a folding factor F . The required intermediate buffer size becomes

$$M_{\text{buf}}^{\text{fold}} = F \times H \times W \times (B_{\text{mem}}/8), \quad (13)$$

where B_{mem} denotes the buffer precision in bits. The spatial buffer is reused across C_{out}/F iterations. Consequently, Channel Folding trades spatial parallelism for reduced on-chip memory usage, decreasing both DSP utilization and buffer size by a factor of C_{out}/F , at the cost of increased sequential execution latency. The folded processing loop is formalized in Algorithm 2.

Algorithm 2: Channel folding layer processing

Require: Input spikes $\mathbf{X}$; weights $\mathbf{W}$; membrane potential $\mathbf{V}_{\text{mem}}$; fold factor F ; threshold V_{th}

Ensure: Output spikes $\mathbf{S}_{\text{out}}$; updated membrane potential $\mathbf{V}_{\text{mem}}$

```

1: for  $i = 0$  to  $(C_{\text{out}}/F) - 1$  do
2:    $c \leftarrow i \times F$ 
3:    $\text{idx} \leftarrow \{c, c + 1, \dots, c + F - 1\}$ 
4:    $\mathbf{Y}_{\text{conv}} \leftarrow \text{Conv2D}(\mathbf{X}, \mathbf{W}[\text{idx}, :, :, :])$ 
5:    $\mathbf{Y}_{\text{bn}} \leftarrow \text{BatchNorm}(\mathbf{Y}_{\text{conv}})$ 
6:    $\mathbf{V}_{\text{mem}}[\text{idx}] \leftarrow \mathbf{V}_{\text{mem}}[\text{idx}] + \mathbf{Y}_{\text{bn}}$ 
7:    $\mathbf{S}_{\text{out}}[\text{idx}] \leftarrow \mathbf{V}_{\text{mem}}[\text{idx}] \geq V_{\text{th}}$ 
8:    $\mathbf{V}_{\text{mem}}[\text{idx}] \leftarrow \mathbf{V}_{\text{mem}}[\text{idx}] \odot (1 - \mathbf{S}_{\text{out}}[\text{idx}])$ 
9: end for

```

3.2.3 Fold Size Selection

The folding factor F determines the balance between inference latency and BRAM consumption. For a convolution layer with C_{out} output channels, the number of sequential iterations introduced by folding is

$$N_{\text{iter}}(F) = \frac{C_{\text{out}}}{F}. \quad (14)$$

To maximize throughput, F should be as large as possible. However, two architectural constraints must be satisfied. First, to avoid control overhead and hardware zero-padding, F must divide the channel configuration of all network stages. The set of valid folding factors is defined as

$$\mathcal{D} = \{f : C_s \bmod f = 0, \forall s \in \{1, \dots, S\}\}. \quad (15)$$

Second, the spatial buffer created by Channel Folding must fit within the remaining BRAM budget $M_{\text{BRAM}}^{\text{avail}}$ after persistent membrane states have been allocated. Combining these constraints, the optimal folding factor is determined as

$$F^* = \max \left\{ F \in \mathcal{D} \mid F \times H_{\text{max}} \times W_{\text{max}} \times (B_{\text{mem}}/8) \leq M_{\text{BRAM}}^{\text{avail}} \right\}. \quad (16)$$

3.2.4 Hardware Mapping Example

To illustrate the proposed hardware optimizations, we continue the scaled network example introduced in [Section 3.1](#). After applying PCS, the resulting channel configuration becomes

$$\mathbf{C}' = [16, 24, 48, 96]. \quad (17)$$

Applying the divisibility constraint in [Eq. \(15\)](#), the set of valid folding factors is

$$\mathcal{D} = \{1, 2, 4, 8\}. \quad (18)$$

Considering the remaining BRAM capacity after membrane state allocation, the optimal folding factor obtained from [Eq. \(16\)](#) is

$$F^* = 4. \quad (19)$$

This configuration effectively balances datapath parallelism and memory consumption while satisfying all architectural constraints.

4 Evaluation Results

4.1 Experimental Setup

Target Platform: Xilinx Zynq UltraScale+ MPSoC (xczu5eV) on Kria KV260 Vision AI Starter Kit, featuring 288 BRAM_18K blocks (648 KB), 64 URAM blocks (2304 KB), 1248 DSP48E2 units, and 117,120 LUTs.

Power Measurement: Board-level power measurements were performed on the deployed Kria KV260 Vision AI Starter Kit using the onboard Texas Instruments INA260 power monitor. Dynamic power was obtained by subtracting the measured idle board power from the active inference board power.

Dataset: SDNET2018 [5], containing over 56,000 labeled images of 256×256 pixels of concrete crack/no-crack samples from bridge decks, walls, and pavements. The dataset was randomly divided into training and test sets using an 80/20 split. To ensure experiment reproducibility, all dataset partitions were generated using a fixed random seed of 42.

Software Environment: The SNN models were implemented in PyTorch [32] and trained using the SpikingJelly SNN framework [19], employing surrogate-gradient BPTT with 20 epochs, batch size 8, learning rate 0.001, and $T = 10$ timesteps. Hardware synthesis was performed using Vitis HLS 2022.2. The overall experimental workflow is illustrated in [Fig. 3](#).

Quantization: Weights and activations are quantized to INT8 using offline post-training quantization (PTQ). Symmetric per-tensor quantization is adopted, where the scale factor for each tensor is computed as $s = \max(|x|)/127$. Activation ranges are calibrated using the first 50 mini-batches (400 images) from the training set through forward inference only, without updating network parameters. During software

evaluation, quantized weights are dequantized back to floating-point values to simulate quantization effects, while the calibrated activation scales are exported for hardware deployment. Convolution computations employ INT32 accumulators for MAC operations, with outputs quantized to INT16 for intermediate storage. Membrane potentials are maintained in INT8 format to minimize on-chip memory usage.

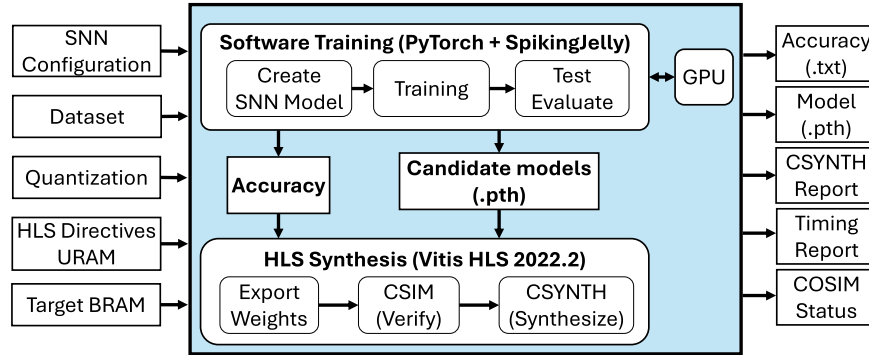


Figure 3: Experimental setup for evaluating the co-design framework. Software training produces the optimized SNN model and accuracy metrics, while HLS synthesis maps the quantized model to FPGA resources and generates synthesis reports.

Data Path: In the deployed HLS design, the accelerator is integrated in the programmable logic and controlled by the processing system through an AXI4-Lite interface. Input and output tensors are transferred through AXI4 memory-mapped interfaces. Inside the accelerator, inference is executed over $T = 10$ timesteps. At each timestep, the input frame is encoded into spikes and propagated through the stem, residual stages, global average pooling, fully connected layer, and argmax operation. The membrane potential arrays and intermediate feature buffers are kept in the on-chip BRAM/URAM memory system during timestep processing. Therefore, the recurrent SNN state is not repeatedly transferred to off-chip memory. For an INT8 input tensor of size $256 \times 256 \times 3$, the input traffic is approximately 192 KB per inference, which is small compared with the persistent on-chip state and buffer storage analyzed in this work. Thus, this work focuses on reducing the on-chip BRAM/URAM footprint rather than optimizing off-chip DDR bandwidth.

4.2 Software-Level Design Space Exploration

To evaluate the influence of the URAM reduction factor β , a sensitivity analysis was conducted using three representative configurations ($\beta = 0.50, 0.75$, and 1.00). Table 6 summarizes the corresponding software and hardware characteristics, including classification accuracy, membrane memory footprint, FPGA resource utilization, and inference latency.

Table 6: Sensitivity analysis of the URAM reduction factor β .

β	Accuracy	MMF	BRAM	URAM	Latency	Params
0.50	86.65	432 KB	241	28	1.51 s	186K
0.75	86.39	488 KB	245	44	2.29 s	402K
1.00	85.15	544 KB	245	56	3.35 s	702K

As β increases, the channel width of the URAM-resident stages increases proportionally, leading to larger model capacity but also higher membrane memory requirements and URAM utilization.

The analytical MMF increases from 432 KB at $\beta = 0.50$ to 488 KB at $\beta = 0.75$, and further exceeds the target memory budget at $\beta = 1.00$ (544 KB). Correspondingly, HLS synthesis reports show that URAM utilization increases from 28 to 44 and 56 blocks, while inference latency also grows approximately linearly from 1.51 to 2.29 s and 3.35 s, respectively.

Although the smallest configuration ($\beta = 0.50$) achieves a marginally higher classification accuracy on SDNET2018 (86.65% vs. 86.39%), the difference is only 0.26 percentage points. Following the optimization objective formulated in Section 3.1, PCS selects the largest feasible β satisfying the membrane-memory constraint while maximizing representational capacity. Consequently, $\beta = 0.75$ is adopted because it provides the highest channel capacity within the 500 KB MMF budget and maintains sufficient implementation margin on the target FPGA (44/64 URAM blocks), whereas $\beta = 1.00$ exceeds the analytical memory budget and substantially increases hardware cost with no improvement in classification performance.

The PCS algorithm described in Section 3.1 is applied to generate six network variants with different channel configurations and stem designs. All models are trained using PyTorch with SpikingJelly ($T = 10$, 20 epochs, batch size 8).

Table 7 summarizes the software-level design space exploration results. Reducing model size has limited impact on classification accuracy. For example, Variant A reduces the parameter count from 11.18 to 2.78 M while achieving higher accuracy (89.15%) than the baseline (86.42%). Further channel reduction decreases the peak feature map size, as observed in Variants B and C (256 KB $\rightarrow$ 128 KB). These results indicate that the original baseline is overparameterized for the binary crack-classification task, since several reduced variants achieve comparable or higher software-level accuracy.

Table 7: Software design space exploration: model variant comparison.

Variant	Channels	Stem	Params	Peak FM	Accuracy
Original	64-128-256-512	Std	11.18M	256 KB	86.42%
A	64-128-256	Std	2.78M	256 KB	89.15%
B	32-64-128-256	Std	2.80M	128 KB	87.98%
C	32-64-128	Std	0.70M	128 KB	87.91%
D	16-24-48-96	Aggr	0.40M	64 KB	86.39%
E	16-24-48-64	Multi	0.25M	128 KB	85.33%
F	24-32-64	Aggr	0.19M	96 KB	86.28%

Note: Std, Aggr, and Multi denote the standard stem, aggressive stem, and multi-convolution stem, respectively. Bold values indicate the selected model variant.

Although Variants A, B, and C achieve higher software-level accuracy, the final deployed architecture was selected based on hardware feasibility rather than accuracy alone. The goal of this comparison is to find a compact SNN that can be mapped to the KV260 FPGA under strict on-chip memory constraints. Variant D was selected because it reduces the peak feature map to 64 KB, reduces the parameter count to 0.40 M, and maintains accuracy comparable to the original baseline (86.39% vs. 86.42%). Although the three-stage variants also achieve competitive accuracy, Variant D retains Stage 4 to preserve a deeper residual hierarchy that is closer to the original baseline and provides a more suitable target for demonstrating deployment of a deep SNN on FPGA. Compared with Variant F, Variant D therefore offers a more balanced deployment target: it remains compact enough for memory optimization while preserving the four-stage structure of the baseline. Therefore, Variant D is used as the deployed model for the subsequent FPGA memory optimization and HLS implementation.

As shown in Table 8, the minimum neuron state memory M_{floor} under INT8 quantization is reduced from 3.9 MB to 488 KB, significantly lowering the on-chip memory requirement for FPGA deployment. Over five independent training runs, the baseline achieves an average accuracy of $87.59 \pm 0.78\%$, while Variant D achieves $85.89 \pm 0.40\%$. The moderate accuracy difference (1.70 percentage points) confirms that the 96.4% reduction in parameters and 87.5% reduction in membrane-state memory incur only a limited impact on classification performance. To enable a consistent class-wise comparison under identical conditions, the detailed metrics reported in Table 9 and the subsequent analyses use a representative single run (seed 42) from each model, where the baseline achieves 86.42% and Variant D achieves 86.39%.

Table 8: Software optimization summary: baseline vs. Variant D.

Metric	Baseline	Variant D	Reduction
Parameters	11.18 M	0.40 M	96.4%
Peak Feature Map	256 KB	64 KB	75%
Accuracy (%) ¹	87.59 ± 0.78	85.89 ± 0.40	-1.70
M_{floor} (INT8)	3.9 MB	488 KB	87.5%

Note: ¹Mean $\pm$ standard deviation over 5 independent training runs (seeds: 42, 123, 456, 789, 2024) with identical train/validation split.

Table 9: Classification performance of the baseline model and Variant D.

Metric	Baseline	Variant D
Accuracy	86.42%	86.39%
Balanced Accuracy	57.52%	56.70%
Precision (weighted)	84.80%	84.43%
Recall (weighted)	86.42%	86.39%
F1-score (weighted)	82.51%	82.35%
Sensitivity (Crack Recall)	16.09%	14.47%
Specificity (No-Crack Recall)	98.95%	98.93%
AUC-ROC	0.718	0.733

Because SDNET2018 is an imbalanced dataset containing approximately 85% non-cracked samples and 15% cracked samples, accuracy alone does not fully characterize classifier performance. Therefore, additional metrics including weighted precision, weighted recall, weighted F1-score, balanced accuracy, sensitivity (crack recall), specificity (non-crack recall), and AUC-ROC are reported in Table 9.

The results show that Variant D maintains classification performance comparable to the baseline despite a 96.4% reduction in parameters and an 87.5% reduction in membrane-state memory. In particular, the AUC-ROC slightly improves from 0.718 to 0.733, indicating that the compact architecture preserves discriminative capability across different decision thresholds.

To evaluate the robustness of the proposed design under low-precision deployment, we additionally compare the floating-point (FP32) model with its INT8 post-training quantized counterpart. As summarized in Table 10, INT8 quantization reduces model memory by approximately 75% (1565.3 to 391.3 KB) while causing only a negligible accuracy degradation of 0.02 percentage points (86.35% to 86.34%). Sensitivity and specificity also remain nearly unchanged after quantization, indicating that the proposed model is robust to INT8 deployment.

Table 10: Classification performance before and after INT8 post-training quantization.

Metric	FP32	INT8
Accuracy	86.35%	86.34%
Sensitivity	14.79%	14.61%
Specificity	98.93%	98.94%
Model Memory	1565.3 KB	391.3 KB

Note: INT8 results are obtained using offline post-training quantization with software-based quantization simulation prior to FPGA deployment.

To further examine whether the small accuracy difference (86.42% vs. 86.39%) affects class-wise behavior, Fig. 4 compares the confusion matrices of the baseline model and the selected Variant D.

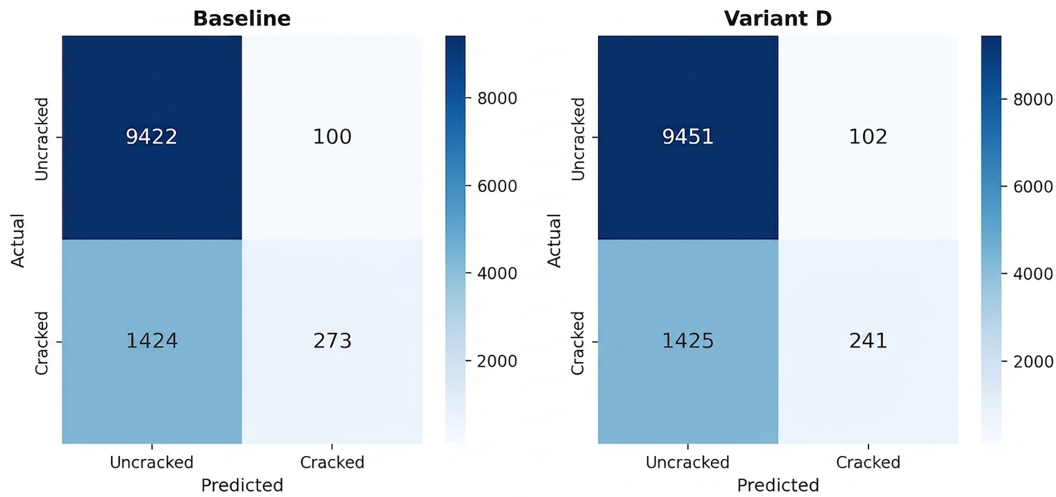


Figure 4: Confusion matrices of the baseline model and the selected Variant D. The optimized model preserves a similar class-wise prediction pattern despite substantial model compression.

The results show that both models exhibit nearly identical classification patterns. For the uncracked class, the number of correctly classified samples changes only from 9422 to 9451. For the cracked class, the number of correctly detected samples changes from 273 to 241. Although Variant D produces 32 additional false negatives, the overall confusion structure remains highly similar to the baseline despite a 96.4% parameter reduction and an 87.5% reduction in membrane memory. These results indicate that the proposed topology reduction preserves the discriminative capability of the network while substantially reducing hardware cost.

To provide a conventional deep-learning reference, we additionally implemented a standard INT8 MobileNetV2 on the SDNET2018 dataset. MobileNetV2 achieved a software classification accuracy of 92.72%, compared with 86.39% for the proposed Spiking ResNet-18. While the CNN provides higher software-level accuracy, it represents a conventional dense computing paradigm. In contrast, the proposed SNN adopts a brain-inspired computational model that represents information using temporal spike trains, providing the foundation for neuromorphic event-driven processing.

4.3 Alternative Hardware Optimization Strategies

During hardware exploration, several common HLS optimizations were evaluated but failed to produce a feasible implementation, as summarized in Table 11. Spatial tiling of the 64×64 feature map caused boundary dependencies and resulted in a timing violation (Worst Negative Slack (WNS) = -4.02 ns). Fully unrolling the output-channel loop led to excessive parallelism and DSP overflow (265%). Mapping additional membrane arrays to URAM also caused URAM overflow (124/64, 193%). In-place BatchNorm introduced read-after-write hazards that prevented efficient pipelining (WNS = -2.2 ns). These results indicate that several CNN-oriented FPGA optimizations are not well suited for deep SNN implementations. Instead, persistent membrane storage becomes the dominant bottleneck, motivating the Channel Folding strategy adopted in this work.

Table 11: Summary of optimization attempts during hardware design exploration.

Optimization	Key Issue
Spatial Tiling	Timing violation (WNS = -4.02 ns)
Explicit Unroll	DSP explosion (265%)
URAM Propagation	URAM overflow (193%)
In-place BatchNorm	RAW hazard (WNS = -2.2 ns)
Channel Folding	BRAM reduced to 245 (85%)

Note: Bold values indicate the feasible optimization strategy adopted in the final design.

4.4 Hardware Optimization Trajectory

Hardware optimization is applied incrementally to resolve resource bottlenecks during FPGA deployment. Table 12 summarizes the evolution of system resources across optimization phases, while Fig. 5 shows the BRAM composition by memory type. In the baseline design, membrane potentials use INT16 precision and all arrays are stored in BRAM, resulting in a total usage of 865 blocks (300%), far exceeding the KV260 capacity. As shown in Fig. 5, membrane storage dominates the memory footprint (484 blocks), followed by intermediate buffers (180 blocks), spike storage (39 blocks), and other structures (162 blocks). Applying INT8 membrane quantization together, with partial URAM mapping for Stage 3–4 membranes reduces BRAM usage to 433 blocks while utilizing 20 URAM blocks. This reduction is mainly due to the combined effect of INT16→INT8 quantization and URAM migration, which decreases BRAM-based membrane storage from 484 to 208 blocks. MTS further migrates Stage 2 membrane arrays to URAM, increasing URAM usage to 44 blocks and reducing BRAM consumption to 385 blocks. At this stage, intermediate buffers become the dominant BRAM consumer (184 blocks).

Table 12: Hardware optimization trajectory.

Phase	BRAM	URAM	DSP	Timing	Latency
Baseline	865 (300%)	0	426	3.6 ns	5.4 s
INT8	433 (150%)	20 (31%)	432	3.6 ns	5.4 s
MTS	385 (133%)	44 (68%)	432	3.6 ns	5.4 s
+Folding	245 (85%)	44 (68%)	55 (4%)	5.7 ns	2.3 s

Note: In this table, the INT8 phase includes membrane quantization and partial URAM mapping for Stage 3–4 membrane arrays, while MTS and Channel Folding correspond to the hardware optimizations proposed in this work. Bold values indicate the final deployed configuration.

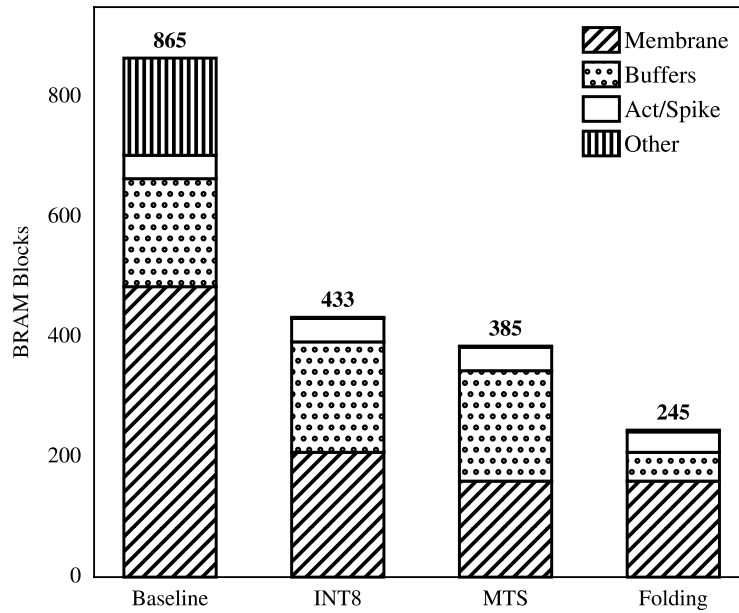


Figure 5: BRAM composition across optimization phases. Each optimization progressively reduces memory footprint, with the final design (245 blocks) fitting within the KV260 limit (288 blocks).

Finally, Channel Folding ($F = 4$) reduces buffer storage from 184 to 48 blocks, bringing total BRAM usage down to 245 blocks (85%), which fits within the KV260 limit. DSP utilization also decreases from 426 to 55 due to sequential channel processing. Interestingly, latency is also reduced (5.4 s $\rightarrow$ 2.3 s). Although folding introduces sequential channel computation, it significantly improves pipeline efficiency and memory bandwidth utilization, leading to higher effective throughput.

Overall, the optimization trajectory reduces BRAM usage from 865 to 245 blocks (72%). The INT8 quantization phase provides the largest single reduction, while MTS and Channel Folding further reduce BRAM consumption to fit within the KV260 resource budget.

4.5 Hardware Implementation Results

Fig. 6 summarizes the final multi-resource utilization of the deployed SNN on the KV260 platform. BRAM usage reaches 245/288 blocks (85.0%), while URAM utilization is 44/64 blocks (68.7%), reflecting the dominant role of on-chip memory in storing membrane states and intermediate buffers. In contrast, DSP utilization is only 55/1248 units (4.4%) because the proposed Channel Folding strategy processes output channels sequentially rather than fully in parallel. LUT usage reaches 86,012/117,120 (73.4%), while flip-flop utilization remains relatively low at 41,428/234,240 (17.7%). Overall, these results indicate that memory resources, rather than compute units, constitute the primary constraint for this deployment.

The sequential processing introduced by Channel Folding directly affects execution latency and throughput. As shown in Fig. 7, the latency for a single inference ($T = 10$) is 2,309 ms (2.3 s) at a clock frequency of 175 MHz. The execution time is dominated by Stage 4 and the stem, which contribute 37.9% (876 ms) and 29.2% (674 ms), respectively. The stem processes large spatial feature maps, while Stage 4 has the largest channel width and is therefore most affected by the sequential channel processing. The remaining stages contribute progressively less: Stage 3 accounts for 17.0% (393 ms), Stage 2 for 10.7% (247 ms), and Stage 1 for 5.3% (123 ms). The Global Average Pooling and fully connected layers introduce negligible overhead (0.04%, 1 ms).

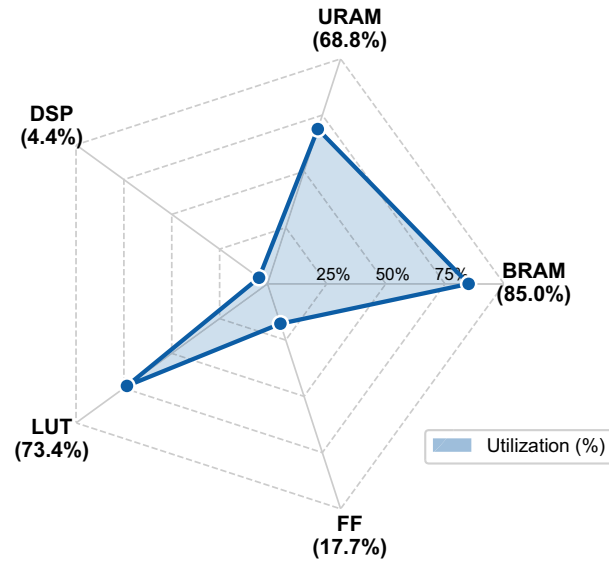


Figure 6: Resource utilization profile of the final design. Block RAM (BRAM), UltraRAM (URAM), digital signal processor (DSP), lookup table (LUT), and flip-flop (FF) utilization are shown as percentages of the available KV260 resources.

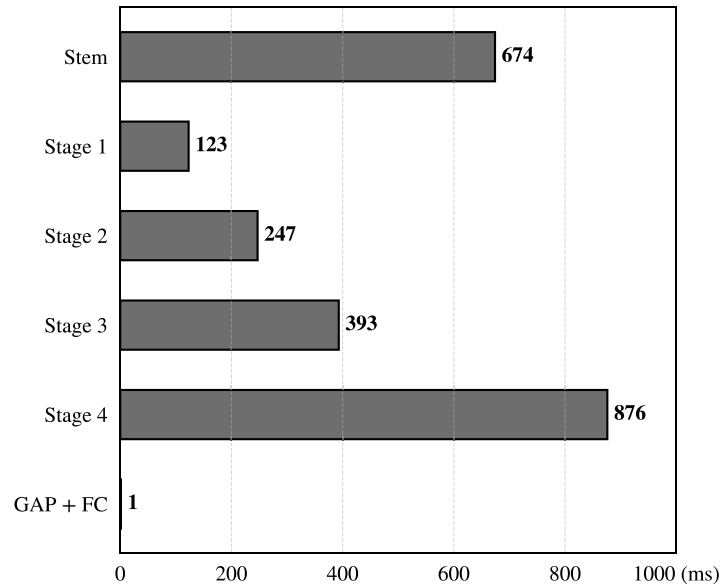


Figure 7: Latency breakdown by network stage ($T = 10$, $f_{\text{clk}} = 175$ MHz). Stage 4 and the stem dominate due to large spatial dimensions and channel counts.

Given this latency, the resulting throughput is approximately 0.43 frames per second. Although this throughput is lower than typical real-time video processing requirements, it is suitable for SHM applications, where inspection images are usually acquired at low sampling rates, e.g., one image per minute.

4.6 Power Consumption

The on-chip power consumption was first estimated using Vivado Power Analysis after place-and-route at approximately 175 MHz. The design achieves timing closure with a WNS of +0.088 ns. As summarized in Table 13, the estimated total on-chip power consumption is 0.860 W, consisting of 0.349 W static power and 0.511 W dynamic power. Logic resources (CLB logic, routing signals, and clock networks) account for 0.279 W of dynamic power, while memory blocks (BRAM and URAM) contribute 0.187 W. DSP power remains relatively small at 0.045 W, consistent with the low DSP utilization observed in the resource analysis.

Table 13: Estimated on-chip power consumption obtained from Vivado Power Analysis.

Component	SNN (Ours)
Logic (CLB + Signals + Clocks)	0.279 W
Memory (BRAM + URAM)	0.187 W
DSP	0.045 W
Total Dynamic	0.511 W
Static	0.349 W
Total On-Chip	0.860 W

Note: Bold values indicate the main power totals.

To validate the estimated power consumption, board-level power measurements were performed on the deployed KV260 platform during active SNN inference using the onboard INA260 power monitor. Power was measured under both idle and active inference conditions, and the dynamic power was calculated as the difference between the two measurements. The measured results are summarized in Table 14.

Table 14: Board-level power measurements on the deployed KV260 platform during SNN inference.

Operating Condition	Measured Power
Idle	3.722 W
Active inference	4.043 W
Dynamic (Active–Idle)	0.321 W

Note: Dynamic power is calculated as the difference between the measured active and idle board power.

The measured dynamic power of 0.321 W is lower than the Vivado-estimated dynamic power. This difference is consistent with the conservative switching activity assumptions employed by Vivado Power Analysis, whereas the measured value reflects the actual runtime behavior of the deployed accelerator. Overall, the board-level measurements confirm that the proposed implementation operates with low dynamic power under real deployment conditions.

4.7 Comparison with State-of-the-Art

Table 15 compares the proposed accelerator with several recent SNN FPGA implementations. Direct comparison across different studies is inherently challenging due to variations in FPGA platforms, target network architectures, and input resolutions. Nevertheless, examining these systems together provides useful insight into the design trade-offs involved in deploying SNNs on resource-constrained edge FPGA devices.

Most prior works focus on lightweight models, such as fully connected networks, LeNet variants, or shallow CNNs, typically operating on heavily downsampled inputs of 28×28 or 32×32 pixels. In contrast, the proposed design targets high-resolution SHM applications, processing 256×256 inputs using a deep ResNet-18 SNN with $T = 10$ timesteps. From a spatial perspective, a 256×256 image contains 64 times more pixels than a 32×32 input. This substantial increase in spatial dimensionality dramatically increases the on-chip memory footprint required to maintain and update membrane potentials. When combined with temporal SNN processing across multiple timesteps, the memory demand becomes even more significant, shifting the primary system bottleneck from computation to memory capacity and bandwidth.

Despite this substantially heavier workload, the proposed system exhibits the lowest reported total on-chip power consumption among the compared implementations (0.860 W). For example, the reported power is significantly lower than that of STI-SNN (1.53 W) and the PYNQ-Z2 implementation (1.54 W), and much lower than SeaSNN (4.996 W). In addition, the design uses only 55 DSP slices. This low compute resource usage is largely enabled by the proposed Channel Folding strategy, which serializes channel processing to reduce active parallel compute resources and associated power consumption.

At the same time, it is important to acknowledge the resulting trade-off in inference efficiency. The proposed design achieves an efficiency of 2.19 GOPS/W, which is lower than the 24.93 GOPS/W reported by Nimbekar et al. [33] and 13.46 GOPS/W by Wang et al. [34]. This difference largely reflects the nature of the underlying workloads. Accelerators targeting 32×32 inputs can typically buffer entire feature maps within localized on-chip memory, enabling extensive parallelization of compute units and therefore maximizing computational efficiency. In contrast, processing 256×256 feature maps on the KV260 platform leads to a more memory-bound architecture. By employing Channel Folding, the proposed design intentionally trades peak computational efficiency (GOPS/W) for the ability to deploy a deeper, high-resolution SNN within the strict memory constraints of an edge FPGA device.

Table 15: Comparison with state-of-the-art SNN FPGA accelerators.

Work	FPGA	Input	Model	Power (W)	Frequency (MHz)	DSP	Inference Efficiency (GOPS/W)	Time Steps
SeaSNN [35]	ZU4EF	28^2	FC (20-10-10)	4.996	200	223	420	–
PYNQ-Z2 [33]	PYNQ-Z2	32^2	ResNet-18 and VGG-11	1.54	100	17	24.93	8
STI-SNN [34]	ZCU102	32^2	SCNN5	1.53	200	–	13.46	1
Frontiers [36]	Zynq7035	32^2	LeNet-large1	1.82	100	–	–	10
Ours	KV260	256^2	ResNet-18 SNN	0.860	175	55	2.19	10

Note: Bold values indicate the proposed design.

Overall, this comparison highlights a different design objective from many prior works. Rather than maximizing theoretical compute efficiency on small benchmark inputs, the proposed architecture demonstrates the practical feasibility of executing deeper SNN models for high-resolution structural inspection tasks within the power and resource limits of commercial edge FPGA platforms.

5 Discussion

5.1 Memory Bottleneck and the Necessity of Hardware-Software Co-Design

Deploying SNNs on resource-constrained FPGA platforms such as the AMD KV260 introduces a significant on-chip memory constraint. Unlike conventional CNN accelerators where intermediate activations can often be overwritten after each layer, SNNs must preserve membrane potentials across multiple timesteps during temporal processing. As a result, deep SNN implementations require persistent storage for a large number of neuron states, substantially increasing the overall on-chip memory demand. Our analysis shows that resolving this constraint requires a coordinated hardware-software co-design approach, as neither model-level compression nor hardware restructuring alone is sufficient.

To provide a hardware reference, a standard INT8 MobileNetV2 was synthesized on the Xilinx Kria KV260 using the same Vitis HLS implementation flow adopted throughout this work. The synthesized design required 2314 BRAM blocks, exceeding the available on-chip BRAM capacity of the target FPGA. Consequently, placement and routing could not be completed, preventing post-route dynamic power estimation. This comparison provides a hardware reference for a conventional CNN under the same deployment methodology. Unlike conventional CNNs, deep SNNs require persistent neuron-state storage across multiple simulation timesteps, introducing unique architectural challenges for FPGA implementation. The proposed co-design framework addresses these challenges and enables the successful deployment of a deep brain-inspired SNN on a resource-constrained edge FPGA.

At the software level, design space exploration was performed to identify a network configuration with a reduced intrinsic memory footprint. The selected configuration, Variant D in [Table 7](#), significantly compresses the model, reducing the peak feature map size from 256 to 64 KB and the total number of parameters from 11.18 to 0.40 M. Importantly, over five independent training runs, this substantial reduction in model size leads to an average accuracy difference of only 1.70 percentage points ($87.59 \pm 0.78\%$ vs. $85.89 \pm 0.40\%$). These results indicate that careful architectural scaling can effectively reduce memory requirements while largely preserving model accuracy.

However, even after this model-level optimization, the hardware implementation initially required 865 BRAM blocks, which greatly exceeds the 288 BRAM blocks available on the KV260 platform. To further reduce memory usage, several hardware-oriented techniques were introduced and evaluated through an ablation study, as shown in [Fig. 8](#). When applied individually, INT8 membrane quantization reduces BRAM usage to 481 blocks, corresponding to a 44.4% reduction; MTS reduces BRAM usage to 681 blocks, corresponding to a 21.3% reduction; and Channel Folding reduces BRAM usage to 569 blocks, corresponding to a 34.2% reduction. Although each technique contributes a meaningful reduction in memory usage, none of them alone can satisfy the hardware constraint. Only when all three techniques are applied together does the BRAM requirement decrease to 245 blocks, corresponding to a total reduction of 71.7%. Notably, this combined reduction is smaller than the sum of the individual reductions, indicating that interactions exist between quantization, memory mapping, and pipeline restructuring. This observation highlights the importance of coordinated hardware-software co-design when targeting memory-constrained FPGA platforms.

5.2 Trade-Offs between Spatial Parallelism and Memory Optimization

The final hardware implementation reveals a clear architectural trade-off between computational parallelism and memory utilization. To fit the design within the KV260's on-chip memory capacity, Channel Folding was introduced to reduce the number of output channels processed simultaneously. In this

configuration, the convolution pipeline processes only a fraction of the output channels per iteration, with a folding factor of $F = 4$, which significantly reduces the size of intermediate feature buffers stored on-chip.

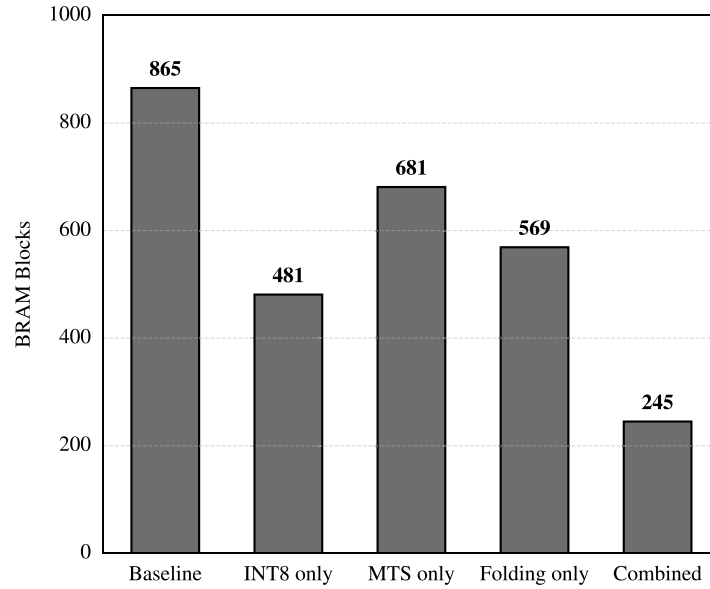


Figure 8: Ablation study showing BRAM usage when each optimization technique is applied independently.

While this restructuring effectively mitigates the memory requirement, it also reduces spatial parallelism in the accelerator. As a result, the utilization of compute resources remains relatively low, with only 55 out of 1248 DSP blocks used (4%). In contrast, memory resources remain heavily utilized, with BRAM usage reaching 245/288 blocks (85%) and URAM usage reaching 44/64 blocks (68%). This resource utilization profile indicates that, for this design, the hardware implementation is primarily constrained by memory capacity rather than compute resources.

The reduced level of parallelism also increases the inference latency. The final implementation achieves an inference latency of approximately 2.3 s. Although this latency would not be suitable for high-frame-rate vision systems, it remains acceptable for SHM applications, where inspection images are typically captured at relatively low sampling rates. In this context, the trade-off between memory footprint and latency is consistent with the operational requirements of the target application.

While the proposed framework substantially reduces on-chip memory requirements, the overall LUT utilization remains relatively high (73%). This reflects the fact that the proposed optimizations primarily target the persistent memory bottleneck of deep SNN deployment rather than general logic utilization. Consequently, reducing memory consumption does not necessarily translate into proportional reductions in LUT usage. Although the relatively high LUT utilization may increase routing complexity during implementation, the reported resource utilization remains within the available logic capacity of the target KV260 platform, enabling successful mapping of the complete deep SNN architecture onto the device.

The current framework was validated on a Xilinx UltraScale+ platform and a binary crack-classification task using SDNET2018. In addition, the PCS formulation was derived for the deployed ResNet-18 architecture, where early stages are mapped to BRAM and deeper stages are mapped to URAM. Extending the framework to other FPGA families, network topologies, and multi-class SHM tasks remains future work.

5.3 Limitation of Crack Classification Performance

Although the proposed framework achieves an overall classification accuracy of 86.39%, the confusion matrix reveals an important limitation in crack detection performance. Specifically, the model exhibits high specificity for intact concrete surfaces but relatively low sensitivity to cracked samples, resulting in a crack recall of 14.47%. For practical SHM applications, minimizing missed crack detections is often more critical than maximizing overall accuracy, making crack recall an important performance indicator alongside overall classification accuracy. Therefore, the current classification performance should be interpreted by considering both the overall accuracy and the class-specific evaluation metrics.

This behavior can be partially explained by the strong class imbalance of the SDNET2018 dataset, in which intact concrete images substantially outnumber cracked samples. Such a class distribution tends to bias the classifier toward the majority class, leading to high specificity but reduced sensitivity to crack images. Since the primary objective of this work is to investigate the proposed hardware-software co-design methodology for deploying deep SNNs on resource-constrained FPGAs, we adopted the original dataset distribution and did not investigate imbalance-aware training strategies.

Therefore, the reported hardware efficiency should be interpreted together with this classification limitation, as improving crack sensitivity remains an important direction for future investigation.

6 Conclusion

This paper presents a hardware-software co-design framework for deploying a deep Spiking ResNet-18 on the resource-constrained Xilinx KV260 edge FPGA. The framework addresses the fundamental SNN memory bottleneck, namely the persistent storage of membrane potentials, through a coordinated multi-level optimization strategy, enabling a high-complexity model to be mapped onto a platform with limited on-chip memory.

The proposed approach combines MMF analysis for memory quantification, PCS for software-level model compression, MTS for heterogeneous BRAM/URAM mapping, and Channel Folding to reduce intermediate feature buffers.

Experimental results on the SDNET2018 dataset show that the framework reduces BRAM utilization by 71.7%, from 865 to 245 blocks, and DSP usage by 87%, while achieving accuracy comparable to the original baseline ($85.89 \pm 0.40\%$ vs. over five runs). Further $87.59 \pm 0.78\%$ more, board-level measurements on the deployed KV260 platform report a dynamic power consumption of only 0.321 W during active SNN inference, validating the low-power characteristics of the proposed implementation for battery-powered edge deployment. This result suggests that the original baseline is overparameterized for the binary crack-classification task, allowing a substantially smaller SNN architecture to retain similar software-level performance while enabling FPGA deployment. Ablation analysis further confirms that although individual techniques provide partial memory reduction, only their combined application satisfies the strict memory constraints of the target platform.

While the resulting design sacrifices spatial parallelism, the inference latency of 2.3 s remains suitable for static or low-frame-rate SHM scenarios, where fixed cameras periodically acquire structural images for inspection rather than continuous video streams. Consequently, the proposed framework is not intended for high-frame-rate applications. The current throughput is primarily constrained by the aggressive Channel Folding strategy adopted to satisfy the limited BRAM capacity of the target FPGA. On FPGA platforms with larger on-chip memory resources, the folding factor could be reduced and additional datapath parallelism introduced through loop unrolling, enabling higher throughput while preserving the proposed hardware-software co-design methodology.

Beyond reducing on-chip memory consumption, the proposed framework establishes the feasibility of deploying deep SNNs on commodity edge FPGAs, providing a practical foundation for low-power, fully on-device SHM systems.

Overall, this work demonstrates that coordinated algorithm-hardware optimization enables practical deployment of deep SNN models on memory-limited edge FPGA platforms.

Acknowledgement: The authors would like to express sincere gratitude to the University of Transport and Communications (UTC), Vietnam, for its academic support during the completion of this research.

Funding Statement: This work was supported by the Ministry of Education and Training (MOET) under Grant No. B2025-GHA-02.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Van-Hai Pham and Duy-Anh Nguyen; data curation, Van-Hai Pham; formal analysis, Van-Hai Pham; investigation, Van-Hai Pham; methodology, Van-Hai Pham and Duy-Anh Nguyen; software, Quang-Thai Pham and Van-Vu Luyen; validation, Quang-Thai Pham and Van-Vu Luyen; visualization, Quang-Thai Pham and Van-Vu Luyen; funding acquisition, Thanh-Toan Dao; project administration, Thanh-Toan Dao; supervision, Duy-Anh Nguyen and Thanh-Toan Dao; writing—original draft, Van-Hai Pham, Quang-Thai Pham, Van-Vu Luyen, and Duy-Anh Nguyen; writing—review and editing, Duy-Anh Nguyen and Thanh-Toan Dao. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The dataset used in this study is publicly available from the SDNET2018 dataset at <https://doi.org/10.1016/j.dib.2018.11.015>. The complete software and hardware implementation, including training scripts, INT8 quantization tools, Vitis HLS source code, FPGA deployment drivers, and board-level power measurement scripts, is publicly available at https://github.com/thai2612/Spiking_Resnet18_for_FPGA_KV260.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Farrar CR, Worden K. An introduction to structural health monitoring. *Phil Trans R Soc A*. 2007;365(1851):303–15. doi:10.1098/rsta.2006.1928.
2. Koch C, Georgieva K, Kasireddy V, Akinci B, Fieguth P. A review on computer vision based defect detection and condition assessment of concrete and asphalt civil infrastructure. *Adv Eng Inform*. 2015;29(2):196–210. doi:10.1016/j.aei.2015.01.008.
3. Cha YJ, Choi W, Suh G, Mahmoudkhani S, Büyüköztürk O. Autonomous structural visual inspection using region-based deep learning for detecting multiple damage types. *Comput Aided Civ Infrastruct Eng*. 2018;33(9):731–47. doi:10.1111/mice.12334.
4. Dung CV, Anh LD. Autonomous concrete crack detection using deep fully convolutional neural network. *Autom Constr*. 2019;99(4):52–8. doi:10.1016/j.autcon.2018.11.028.
5. Dorafshan S, Thomas RJ, Maguire M. SDNET2018: an annotated image dataset for non-contact concrete crack detection using deep convolutional neural networks. *Data Brief*. 2018;21:1664–8. doi:10.1016/j.dib.2018.11.015.
6. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; 2016 Jun 27–30; Las Vegas, NV, USA. p. 770–8. doi:10.1109/CVPR.2016.90.
7. Li E, Zeng L, Zhou Z, Chen X. Edge AI: on-demand accelerating deep neural network inference via edge computing. *IEEE Trans Wirel Commun*. 2020;19(1):447–57. doi:10.1109/TWC.2019.2946140.
8. Satyanarayanan M. The emergence of edge computing. *Computer*. 2017;50(1):30–9. doi:10.1109/MC.2017.9.
9. Maass W. Networks of spiking neurons: the third generation of neural network models. *Neural Netw*. 1997;10(9):1659–71. doi:10.1016/S0893-6080(97)00011-7.

10. Roy K, Jaiswal A, Panda P. Towards spike-based machine intelligence with neuromorphic computing. *Nature*. 2019;575(7784):607–17. doi:10.1038/s41586-019-1677-2.
11. Tavanaei A, Ghodrati M, Kheradpisheh SR, Masquelier T, Maida A. Deep learning in spiking neural networks. *Neural Netw*. 2019;111(10):47–63. doi:10.1016/j.neunet.2018.12.002.
12. Li Z, Lemaire E, Abderrahmane N, Bilavarn S, Miramond B. Efficiency analysis of artificial vs. spiking neural networks on FPGAs. *J Syst Archit*. 2022;133(9):102765. doi:10.1016/j.sysarc.2022.102765.
13. Neftci EO, Mostafa H, Zenke F. Surrogate gradient learning in spiking neural networks: bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Process Mag*. 2019;36(6):51–63. doi:10.1109/MSP.2019.2931595.
14. Wu Y, Deng L, Li G, Zhu J, Shi L. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Front Neurosci*. 2018;12:331. doi:10.3389/fncom.2018.00031.
15. Zhang C, Li P, Sun G, Guan Y, Xiao B, Cong J. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*; 2016 Aug 8; San Francisco, CA, USA. New York, NY, USA: ACM; 2016. p. 326–31. doi:10.1145/2934583.2934624.
16. Karamimanesh M, Abiri E, Shahsavari M, Hassanli K, Schaik A, Eshraghian J. Spiking neural networks on FPGA: a survey of methodologies and recent advancements. *Neural Netw*. 2025;186(11):107256. doi:10.1016/j.neunet.2025.107256.
17. Frenkel C, Bol D, Indiveri G. Bottom-up and top-down approaches for the design of neuromorphic processing systems: tradeoffs and synergies between natural and artificial intelligence. *Proc IEEE*. 2023;111(6):623–52. doi:10.1109/jproc.2023.3273520.
18. Yin R, Moitra A, Bhattacharjee A, Kim Y, Panda P. SATA: sparsity-aware training accelerator for spiking neural networks. *IEEE Trans Comput Des Integr Circuits Syst*. 2023;42(6):1926–38. doi:10.1109/tcad.2022.3213211.
19. Fang W, Chen Y, Ding J, Yu Z, Masquelier T, Chen D, et al. SpikingJelly: an open-source machine learning infrastructure platform for spike-based intelligence. *Sci Adv*. 2023;9(40):ead1480. doi:10.1126/sciadv.adi1480.
20. Islam MMM, Kim JM. Vision-based autonomous crack detection of concrete structures using a fully convolutional encoder–decoder network. *Sensors*. 2019;19(19):4251. doi:10.3390/s19194251.
21. Khatir A, Capozucca R, Khatir S, Magagnini E, Thanh LC, Riahi MK. Advancements and emerging trends in integrating machine learning and deep learning for SHM in mechanical and civil engineering: a comprehensive review. *J Braz Soc Mech Sci Eng*. 2025;47(9):419. doi:10.1007/s40430-025-05697-5.
22. Rueckauer B, Lungu IA, Hu Y, Pfeiffer M, Liu SC. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Front Neurosci*. 2017;11:682. doi:10.3389/fnins.2017.00682.
23. Sengupta A, Ye Y, Wang R, Liu C, Roy K. Going deeper in spiking neural networks: VGG and residual architectures. *Front Neurosci*. 2019;13:95. doi:10.3389/fnins.2019.00095.
24. Li Y, Zeng Y. Efficient and accurate conversion of spiking neural network with burst spikes. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*; 2022 Jul 23–29; Vienna, Austria. p. 2485–91. doi:10.24963/ijcai.2022/345.
25. Neil D, Liu SC. Minitaur, an event-driven FPGA-based spiking network accelerator. *IEEE Trans Very Large Scale Integr Syst*. 2014;22(12):2621–8. doi:10.1109/TVLSI.2013.2294916.
26. Blott M, Preusser TB, Fraser NJ, Gambardella G, O'Brien K, Umuroglu Y, et al. FINN-R: an end-to-end deep-learning framework for fast exploration of Quantized neural networks. *ACM Trans Reconfig Technol Syst*. 2018;11(3):1–23. doi:10.1145/3242897.
27. Davies M, Srinivasa N, Lin TH, Chinya G, Cao Y, Choday SH, et al. Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro*. 2018;38(1):82–99. doi:10.1109/MM.2018.112130359.
28. Merolla PA, Arthur JV, Alvarez-Icaza R, Cassidy AS, Sawada J, Akopyan F, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*. 2014;345(6197):668–73. doi:10.1126/science.1254642.
29. Carpegna A, Savino A, Carlo DS. Spiker: an FPGA-optimized hardware accelerator for Spiking neural networks. In: *Proceedings of the 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*; 2022 Jul 4–6; Nicosia, Cyprus. p. 14–9. doi:10.1109/isvlsi54635.2022.00016.

30. Gerlinghoff D, Wang Z, Gu X, Goh RSM, Luo T. E3NE: an end-to-end framework for accelerating spiking neural networks with emerging neural encoding on FPGAs. *IEEE Trans Parallel Distrib Syst.* 2021;33(11):3207–19. doi:10.1109/tpds.2021.3128945.
31. Padovano D, Carpegna A, Savino A, Carlo DS. SpikeExplorer: hardware-oriented design space exploration for Spiking neural networks on FPGA. *Electronics.* 2024;13(9):1744. doi:10.3390/electronics13091744.
32. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: an imperative style, high-performance deep learning library. *arXiv:1912.01703.* 2019. doi:10.48550/arxiv.1912.01703.
33. Nimbekar A, Katti P, Li C, Al-Hashimi BM, Acharyya A, Rajendran B. Hardware-software co-optimised fast and accurate deep reconfigurable spiking inference accelerator architecture design methodology. In: *Proceedings of the 2024 IEEE 37th International System-on-Chip Conference (SOCC)*; 2024 Sep 16–19; Dresden, Germany. New York, NY, USA: IEEE; 2024. p. 1–6. doi:10.1109/SOCC62300.2024.10737858.
34. Wang K, Yang C, Yu C, Ang YS, Wang B, Wang A. STI-SNN: a 0.14 GOPS/W/PE single-timestep inference FPGA-based SNN accelerator with algorithm and hardware co-design. *arXiv:2506.08842.* 2025. doi:10.48550/arxiv.2506.08842.
35. Geng S, Wang Z, Liu Z, Zhang M, Zhu X, Dan Y. Hardware implementation of FPGA-based spiking attention neural network accelerator. *PeerJ Comput Sci.* 2025;11(10):e3077. doi:10.7717/peerj-cs.3077.
36. Wu J, Lu L, Wang Y, Li Z, Chen C, Li Q, et al. Efficient spiking convolutional neural networks accelerator with multi-structure compatibility. *Front Neurosci.* 2025;19:1662886. doi:10.3389/fnins.2025.1662886.