AxOSpike: Spiking Neural Networks-Driven Approximate Operator Design

Salim Ullah, Siva Satyendra Sahoo, and Akash Kumar, *Senior Member, IEEE*

***Abstract*—Approximate computing (AxC) is being widely researched as a viable approach to deploying compute-intensive artificial intelligence (AI) applications on resource-constrained embedded systems. In general, AxC aims to provide disproportionate gains in system-level power-performance-area (PPA) by leveraging the implicit error tolerance of an application. One of the more widely used methods in AxC involves circuit pruning of arithmetic operators used to process AI workloads. However, most related works adopt an application-agnostic approach to operator modeling for the design space exploration (DSE) of Approximate Operators (AxOs). To this end, we propose an application-driven approach to designing AxOs. Specifically, we use spiking neural network (SNN)-based inference to present an application-driven operator model resulting in AxOs with better-PPA-accuracy tradeoffs compared to traditional circuit pruning. Additionally, we present a novel FPGA-specific operator model to improve the quality of AxOs that can be obtained using circuit pruning. With the proposed methods, we report designs with up to 26.5% lower PDPxLUTs with similar application-level accuracy. Further, we report a considerably better set of design points than related works with up to 51% better-Pareto front hypervolume.**

***Index Terms*—Accelerator architecture, AxC, arithmetic circuit design, computer arithmetic, FPGAs, operator modeling, SNNs.**

I. Introduction

THE LAST few years have seen rapid strides in bringing artificial intelligence (AI)-based processing into our day-to-day lives. While the more complex processing, such as analytics and large generative AI, are still limited to cloud-based computing, edge AI is becoming increasingly complex owing to applications, such as extended reality (XR) and large language model (LLM) inference. As a result, there is an increased effort across the computation stack—from algorithms to electronic devices–toward enabling complex AI on resource-constrained edge devices. At the algorithm level, spiking neural network (SNN) provides a cheaper alternative to traditional artificial neural networks (ANNs) [1]. In addition to being more biomimetic, SNNs provide a more energy-efficient alternative. The improved energy efficiency usually emanates from avoiding complex multiply-accumulate (MAC) operations. Further, the spike train-based representation of the intermediate features reduces the cost of data movement. However, to enable true event-driven processing of SNNs, the hardware implementation must enable the parallel processing of a large number of neurons. Correspondingly, SNN-based processing can benefit from low-cost implementations of each neuron.

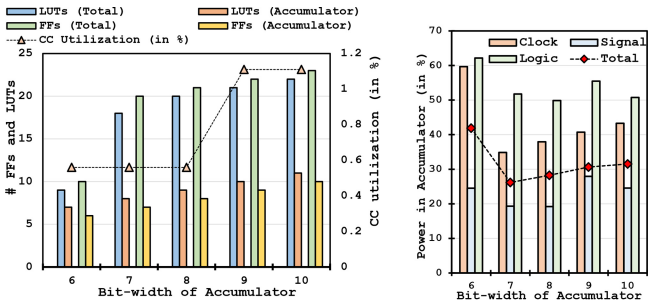


Fig. 1. Characterization results for the hardware implementation of a single neuron and the constituent accumulator on an FPGA for different bit-width accumulators. (a) Resource usage in terms of LUTs, FFs, and CC utilization. (b) Power dissipation of accumulator components (total, clock, logic, and signal) compared to the neuron's components' power dissipation.

The primary arithmetic operations in the neuron of an SNN usually include the accumulation of the membrane potential and the comparison of the membrane potential with a threshold value. In digital hardware, the accumulation involves adding the weight value to the current potential, depending upon the presence/absence of a spike. Fig. 1 shows the cost of implementing a single neuron on a field programmable gate array (FPGA) and the corresponding cost of the accumulator. The results correspond to the characterization of the neuron on an AMD Xilinx Zynq UltraScale+^{TM} MPSoC (ZU3EG A484). The bar-plot groups in the figure correspond to different bit-widths of the accumulator while using signed 4-bit integer weights. Fig. 1(a) shows the resource utilization in terms of flip-flops (FFs), lookup tables (LUTs) and carry-chains (CCs)[1]. The constituent accumulator uses between 44% to 78% of LUTs and between 35% to 60% of the FFs used in the neuron. In Fig. 1(b), the bar plots show the percentage of the power dissipation for each component—*logic, signal, and clock* in the accumulator—compared to the

Manuscript received 2 August 2024; accepted 3 August 2024. This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under the X-ReAp Project under Project 380524764. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Siva Satyendra Sahoo.)*

Salim Ullah and Akash Kumar are with the Chair of Embedded Systems, Ruhr University Bochum, 44801 Bochum, Germany (e-mail: Salim.Ullah@ruhr-uni-bochum.de; Akash.Kumar@ruhr-uni-bochum.de).

Siva Satyendra Sahoo is with the Interuniversity Microelectronics Centre, Leuven, 3001 Leuven, Belgium (e-mail: Siva.Satyendra.Sahoo@imec.be).

Digital Object Identifier 10.1109/TCAD.2024.3443000

[1]Percentage utilization of all CCs in the FPGA.

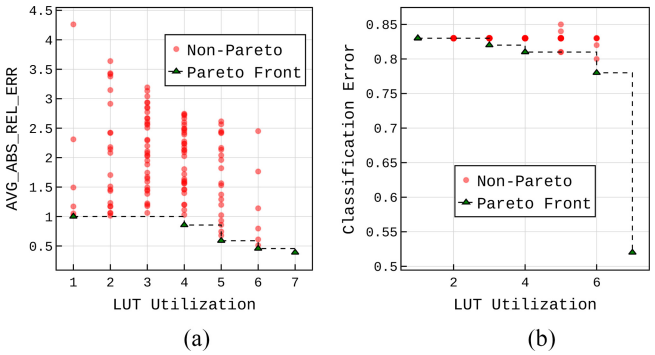


Fig. 2. Exhaustive designs for approximate signed 7-bit adders with overflow using operator model and automated pruning proposed in [9]. *LUT utilization* refers to the number of LUTs used for implementing the approximate operator. (a) Operator-level design space. (b) Application-level design space.

power dissipation of the same component in the neuron. The line plot shows the percentage of total power consumption in the accumulator compared to the neuron. Here, too we observe a considerable fraction of the cost in the accumulator. While Fig. 1 demonstrates the effect of precision scaling on the implementation of the neuron, additional circuit-level methods can be explored for low-cost computer arithmetic.

Approximate Computing (AxC) forms one of the more novel approaches to implementing resource-efficient computing [2]. In general, AxC aims to provide disproportionate gains in power-performance-area (PPA) by leveraging the implicit error tolerance of an application. While the general principle of AxC can be implemented at different abstractions, approximate circuits for arithmetic operations are widely researched as a viable approach for AI workload processing [3], [4]. This can be attributed to the homogeneity of arithmetic units (primarily MACs) being used across a wide spectrum of AI algorithms and the inherent error-tolerant nature of AI applications. In AxC, circuit pruning to generate novel Approximate Operators (AxOs) for computer arithmetic forms a primary method for implementing low-cost hardware [5], [6], [7], [8], [9]. Novel approaches to circuit pruning—both application-specific and otherwise—have been proposed for application-specific integrated circuits (ASICs) and FPGAs. While ASIC-based designs can provide a higher degree of circuit optimizations, FPGAs's capability to dynamically deploy designs with varying PPA-error tradeoffs makes them an attractive option for AxC.

Related works in FPGA-based AxO design include methods ranging from synthesizing ASIC-optimized AxOs for FPGA-based implementations [10], manual pruning in FPGA-optimized accurate operator implementations [8], to automated pruning of accurate operators to generate a library of AxOs [9]. However, all these methods adopt fairly generic operator models and circuit pruning methods. For instance, Fig. 2 shows the design space for a signed 7-bit adder with the AxOs generated through the operator model proposed in AppAxO [9]. This operator model includes removing a subset of LUTs from the accurate operator implementation to realize AxOs. Consequently, the resulting design space comprises 127 $(2^7 - 1)$ AxOs with LUT utilization ranging from 1 to 7. It is worth noting that the accurate 7-bit adder implementation in Fig. 2 processes two 7-bit operands and produces a 7-bit result, which results in arithmetic overflows for some input combinations. This implementation is referred to as *OvErr_BASE* adder.

Fig. 2(a) displays the LUT-error tradeoffs of the complete design space. For this purpose, the outputs of the *OvErr_BASE* approximate adders are compared with an overflow-safe adder that processes 7-bit operands to produce an 8-bit output. The error metric used in the plot, AVG_ABS_REL_ERR, estimates the mean statistics for the error in the sum produced by the operator, compared to the accurate value, for all possible input combinations. AppAxO's approximation methodology does not include any adaptations in the accurate implementation of an operator that accounts for the accuracy degradation during the subsequent circuit pruning. Further, similar to most related works, AppAxO adopts a bottom-up approach to AxO design for an application and can lead to limited benefits for an application.

The bottom-up approach usually involves taking a generic operator model and implementing circuit pruning with the model. For instance, Fig. 2(b) shows the AxOs in Fig. 2(a) used as the accumulator in the neurons in an SNN for MNIST digit classification [11]. The Pareto-front w.r.t. the AxOs' LUT utilization and the application's classification error follows a similar pattern to that in Fig. 2(a), with five dominant designs. Since the initial accurate design does not contain any application-specific optimizations, there is limited scope to obtain any application-specific benefits during the design space exploration (DSE) for AxOs using such generic approaches. To this end, we present an application-driven approach to designing AxOs.

Our novel contributions include the following.

1) We present an SNN-driven approximation methodology for designing AxOs. Specifically, we propose a novel approximate operator model that integrates SNN-specific adaptations to obtain improved PPA-error tradeoffs with circuit pruning. With the proposed adaptation, we report designs with up to 26.5% lower PDPxLUTs, while maintaining the same application-level accuracy.
2) We present FPGA-specific pruning-aware optimizations to the operator model. Specifically, we propose novel adaptations to the accurate implementation of signed adders that allow for recovering some of the errors introduced during circuit pruning. With the proposed method, we report up to 102.1% better-Pareto front hypervolume than related state-of-the-art methods.
3) We present an improved circuit pruning method for FPGA-based AxOs. Specifically, we propose a technique that integrates additional Degrees-of-Freedom (DoFs) during circuit pruning, compared to state-of-the-art methods. With this approach, we report a considerably better set of design points than related works with up to 51% higher-Pareto front hypervolume.

The remainder of this article is organized as follows. We present a brief background and survey of the related works in Section II. Section III presents the application and neuron model of the SNN used for evaluating the proposed methods. We present the novel operator models and approximate designs

in Section IV. The experimental evaluation of the proposed contributions is discussed in detail in Section V. Section VI concludes this article with a summary of the presented work and a discussion of the scope for related future work.

II. Background and Related Works

A. Designing Approximate Arithmetic Operators

Recently AxC techniques covering multiple layers of the computation stack, including AxOs have been proposed [12]. Similarly, various works have proposed novel techniques for designing AxOs that utilize the LUT- and CC-based structures in an FPGA more efficiently. For instance, Ullah et al. [13], [14], [15] have presented methodologies for building higher-order AxOs from optimized lower-order AxOs (4×4 multipliers). Similarly, Ullah et al. [8] have presented approximate signed multipliers based on the radix-4 booth algorithm [16]. They limited the approximation to the partial product generation and used manual removal of LUTs, along with truncating input bits to present a few AxO designs. The LUT selection is based on the ranking of LUTs contributing to the critical path delay (CPD) and power dissipation. The CPD usually refers to the maximum delay from any FF output to any FF input. For combinational arithmetic units, it translates to the maximum delay between any of the inputs and any of the outputs. Ullah et al. [9], [17] have provided an automated approach to this pruning methodology for synthesizing both application-specific and application-agnostic AxOs. In another approach, the works presented in [10] and [18] perform FPGA-specific DSE on a set of ASIC optimized AxOs generated using EvoApprox [5]. While this approach reduces the design space considerably, it limits the scope of FPGA-specific optimizations that can be explored using circuit pruning

Broadly, the design methodologies for FPGA-based AxOs can be categorized into the following approaches.

1) *Application Specificity:* While some works integrate the application's behavior during the DSE [5], [9] for AxOs, other works design AxOs considering operator-level error metrics only [8], [17].
2) *Synthesis and Selection:* Selection refers to choosing the appropriate AxOs to be implemented in the FPGA-based accelerator. The selection could be from a set of ASIC optimized AxOs or from FPGA-specific designs [8], [10], [18]. However, the synthesis approach entails integrating the FPGA- and/or application-specific characteristics to design novel AxOs [9], [15], [17].
3) *Manual and Automated DSE:* While some of the related works employ manual optimizations to circuit pruning [8], other works employ automated search methods, including state-of-the-art machine learning algorithms [5], [9], [10]. The difference in both these approaches usually results in a varying number of AxO designs and the corresponding range of PPA-error tradeoffs.

B. Edge AI and SNN

Edge computing forms an essential component of any modern computing ecosystem. To this end, various methods for enabling edge AI on resource-constrained embedded systems are being actively researched. Network pruning, the removal of individual noncritical parameters and filters from a trained ANN, constitutes one such approach [19], [20], [21]. Further, precision scaling of the weights and/or features is widely used to reduce the computation and data movement costs of ANN-inference [22], [23]. However, such generic approaches, including weights clustering, sparse computing [24], etc., do not alter the need for large amounts of data movement and MAC operation considerably. In contrast, SNNs employ an event-driven processing and holds the potential for reducing energy consumption by orders of magnitude. The reduction in computing costs is primarily derived from eliminating multiplication operations between weights and features and by computing only when a spike occurs. Similarly, the representation of the features by a spike train enables reducing the data movement requirements in terms of memory and communication. Further, generic methods, such as network pruning and precision scaling, can also be applied to SNN-based computing.

Hardware acceleration forms one of the major factors that has enabled extracting useful results from AI computing. Similar to ANNs, SNNs can also benefit from hardware acceleration. However, their asynchronous event-driven nature of computing can benefit the most from spatial accelerators rather than GPUs or other thread parallel accelerators. To this end, various ASIC- and FPGA-based accelerators have been proposed for SNNs. FPGA-based implementations of SNN accelerators include SyncNN [25], Gyro [26] and RANC [27]. A more detailed survey of FPGA implementations of SNNs can be found in [28]. One of the common themes across the SNN accelerators is toward enabling the mapping of a high number of parallel neurons. Precision scaling and approximate operators can enable reduced resource consumption of each neuron, thereby allowing a larger number of neurons to work in parallel within the same resource constraints. There has been very little work related to using approximation in SNNs, specifically related to hardware design. Sen et al. [29] have proposed an algorithm-level approximation approach for SNNs. Specifically, the proposed method involves determining spike-triggered neuron updates that can be skipped with little or no impact on output quality. Consequently, the energy consumption owing to the computing and memory access for each of those unnecessary updates can be saved. Our current work focuses more on the design of low-cost hardware arithmetic for the accumulator in the neuron and is complementary to algorithm-level approximations, such as network pruning and AxSNN [29].

C. Summary

For our current work, we focus on the design of AxOs driven by an SNN application. We do not propose any novel SNN architectures, instead focusing on how existing architectures can benefit from AxC and precision scaling. In this context, Table I summarizes the different aspects of designing approximate arithmetic operators across related works.

Application-Specific Operator Model: While some of the related works perform application-specific DSE, the starting

TABLE I
COMPARING RELATED WORKS

Related Work	[5]	[10]	[9]	[8]	[17]	*AxOSpike*
App-driven Operator Model	✗	✗	✗	✗	✗	✓
Pruning-aware Operator Model	✗	✗	✗	✗	✗	✓
FPGA-specific Circuit Pruning	✗	✗	✓	✓	✓	✓
Automated AxO Search	✓	✓	✓	✗	✓	✓

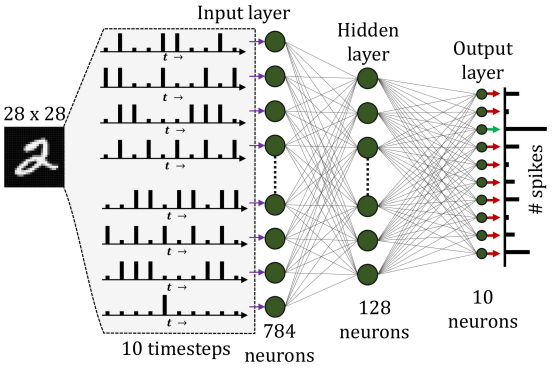


Fig. 3. MNIST digit recognition: ANN versus SNN.

point of the operator model does not include any application-specific information. This bottom-up approach does not leverage application-specific behavior and relies on the search algorithm to integrate the application's error tolerance while synthesizing/selecting novel AxOs.

Pruning-Aware Operator Model: Similarly most related works use generic implementations of the accurate operator as the operator model and do not implement any pruning-aware modifications in the model.

FPGA-Specific Pruning With Automated Search: While a lot of works use FPGA-specific pruning in their automated search methods, they do not fully exploit different DoFs available for pruning and the consequent rewiring during the synthesis of novel AxOs.

To this end, we posit that the design of AxOs can benefit from more complex operator models that integrate information from both the application and the underlying hardware structures of the platform architecture.

III. SNN MODEL

A. MNIST Digit Recognition

For our current work, we use the classification problem for MNIST digit recognition [11] using fully connected layers only as the network model. Fig. 3 shows the network structure that includes a single hidden layer along with the input and output layers. While the training is usually performed at a higher precision (IEEE FP32), the trained model is then quantized to varying integer precisions, to enable low-cost arithmetic. Although, the network can be trained specifically for an SNN implementation, we have limited the current work to using the trained weights of the ANN, similar to the approach used in SyncNN [25]. Next, we compare and contrast the different aspects of the network for ANN- and SNN-based processing.

1) Input Encoding: In ANNs, the pixel values of the input image are encoded into precision-specific integer values and passed onto the next layer. However, SNNs are tailored to exploit time-varying data, and hence, each pixel value needs to be encoded into a spike train, a sequence of 0s and 1s. Depending upon the type of encoding—rate, latency, or delta—each pixel (feature) is converted into a spike train of a fixed length. We have used rate encoding, which uses the input features to determine the spiking frequency. Fig. 3 shows the hypothetical spike trains across ten timesteps for some input features.

2) Neuron Processing: In ANNs, the incoming features (integer values) to a neuron are multiplied by their corresponding weight values, accumulated across all inputs, and are passed to an activation function like ReLU to determine the features for the next layer. However, in SNNs, the neuron takes the sum of weighted inputs across all input edges, weighted by the input spike value. These values are integrated over time (membrane voltage) and once a constant threshold value is reached, a spike is generated from the neuron, and the membrane voltage is reset. As a result, the output from a neuron is also a spike train, unlike the integer-valued outputs from an ANN's neuron.

3) Output Classification: In the case of ANNs implementing output classification, a Softmax layer is used to determine the image class. However, in the SNN, we can use the count of spikes on the output layer's nodes to determine the appropriate class of the image. As shown in Fig. 3, a well-trained network would exhibit a clear difference in the number of spikes seen at each node after the 10 timesteps.

B. Neuron Model

The Hodgkin–Huxley Neuron model [30] is widely considered the closest to how biological neurons behave. SNN implementations use a wide spectrum of neuron models. The leaky integrate-and-fire (LIF) neuron is the most widely used model in SNN implementations [31], [32]. However, the LIF model also has complex implementation due to many internal states owing to the refractory period and decay of the membrane voltage. As a result, more digital-friendly low-cost implementations have been proposed [33]. For our current work, we have used the basic integrate and fire (IF) model. It does not encode any decay and refractory period-related information and the internal state is only defined by the current membrane voltage. If the voltage (accumulated value) exceeds the threshold value, a spike is generated and the accumulator value is reset. Although simple, the model allows us to focus on the variations in the accumulator operator implementation. However, the accumulator-related AxO exploration can be easily expanded to other neuron models.

IV. APPROXIMATE OPERATOR DESIGN

The accumulator in the SNN neuron accumulates $W-bit$ weights to produce an $N-bit$ output. We denote such

TABLE II
PERFORMANCE COMPARISON OF 4 × 8_8 AND 8 × 8_8 ADDERS

Design	LUTs	CPD [ns]	Power [uW]
4 × 8_8	8	1.36	463.05
8 × 8_8	8	1.43	499.1

accumulators as $W \times N\_N$, where $W < N$, in this article. The actual value of N is an important design decision and defines the upper limits of adders before producing overflows. For example, using a 4-bit adder to accumulate 4-bit weights while producing a 4-bit output (denoted as 4 × 4_4 adder) is susceptible to producing arithmetic overflow frequently. Meanwhile, employing a higher-bit width accumulator, such as 4 × 8_8, would result in less frequent overflows for the accumulation of 4-bit weights. However, the FPGA-optimized implementations of a $W \times N\_N$ adder and an $N \times N\_N$ adder show that both adders produce similar PPA metrics. For example, Table. II compares the LUT utilization, CPD, and dynamic power consumption of 4 × 8_8 and 8 × 8_8 FPGA-optimized adders. In the 4 × 8_8 adder, the 4-bit operand is sign-extended before addition with the 8-bit operand. It can be observed that both implementations have similar performance metrics. Therefore, we have used $N \times N\_N$ operators to accumulate $W-bit$ weights in this work. This design decision also helps implement SNN-specific adaptations to improve the accuracy of proposed adders.

A. Implicit Approximation by Overflow

Fig. 4 depicts the LUTs and carry chains-based representation of our base $N \times N\_N$ design for $N = 4$. In this configuration, the LUTs receive operands in 2's complement form and utilize (1) to determine the values of the output signals *O5* and *O6*. These signals govern the corresponding carry chains in the FPGAs to compute the final sum. Despite being an accurate adder, the base adder is still susceptible to generating incorrect outcomes due to arithmetic overflows. Hence, we refer to it as the *OvErr_BASE* adder.[2] The arithmetic overflows encountered by the *OvErr_BASE* adder can significantly impact the output accuracy of SNNs, which work on the principle of producing a spike when the accumulated weight values reach a threshold value. These arithmetic overflows can result in comparing an incorrect accumulated value with the threshold value

$$O5 = A_x \text{ AND } B_x;\ O6 = A_x \text{ XOR } B_x. \tag{1}$$

The arithmetic overflows occur when the addition of two positive numbers produces a negative result or the addition of two negative numbers produces a positive number. In our current work, we explore the research question of whether an approximate $N \times N\_N$ adder with PPA similar to an $N \times N\_N$ *OvErr_BASE* adder produces more accurate results by controlling overflows. To answer this question, we propose three overflow-safe approximate adders that utilize resources similar to an *OvErr_BASE* adder.

[2]Fig. 2 has used a 7-bit *OvErr_BASE* adder.

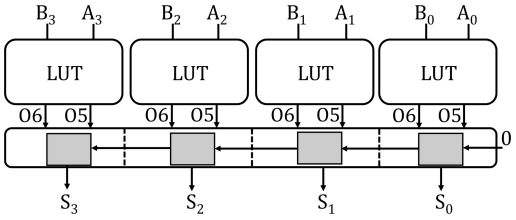


Fig. 4. LUTs and carry chains-based implementation of a 4-bit signed adder: *OvErr_BASE* design.

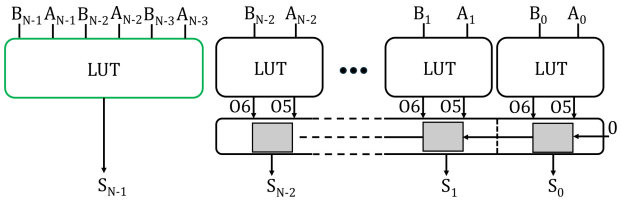


Fig. 5. $N \times N\_N$ *OvCtrl_POS* and *OvCtrl_NEG* adder structure.

However, they introduce deliberate approximations in the addition process to avoid overflows. In the following sections, we will discuss these approximate architectures in detail.

B. Application-Specific Overflow-Safe Approximate Adders

The initial two overflow-safe approximate adders, denoted as *OvCtrl_POS* and *OvCtrl_NEG*, are based on analyzing both operands' sign bit, i.e., the most-significant bit (MSB). Fig. 5 demonstrates a generic view of the two proposed architectures. As shown in Fig. 5, the LUT receiving the sign bits, i.e., A_{N-1} and B_{N-1}, has been detached from the rest of the circuit. If both operands are positive, i.e., the MSB of both operands is 0, the corresponding LUT produces a 0 output. Similarly, if both operands are negative, the LUT produces a 1 as the output. For all other input combinations, the sum's sign bit (S_{N-1}) accurate computation depends on the output carry from the preceding computation. However, the routing of the output carry from the preceding computations to the most significant LUT results in extra routing delays and, therefore, has not been considered in this design. In our proposed *OvCtrl_POS* and *OvCtrl_NEG* architectures, we used LUT's available input pins to provide more bits from the input operands (A_{N-2}, A_{N-3}, B_{N-2}, B_{N-3}) to predict the missing carry. Equation (2) defines the logic the most significant LUT implements for such cases. However, in some cases, the LUT cannot predict the correct sign bit due to the lack of knowledge about other bits of the operands. For instance, Table III presents two examples of a 6-bit adder. The three most significant bits in both examples are the same, but they produce answers with different signs. For such cases, we approximate the sign bit to either 0 (resulting in *OvCtrl_POS* architecture) or 1 (resulting in *OvCtrl_NEG* architecture)

$$\begin{aligned}\text{Sign} = &\left(-2^{N-1}A_{N-1} + 2^{N-2}A_{N-2} + 2^{N-3}A_{N-3}\right) \\ &+ \left(-2^{N-1}B_{N-1} + 2^{N-2}B_{N-2} + 2^{N-3}B_{N-3}\right).\end{aligned} \tag{2}$$

TABLE III
6 × 6_6 *OvCtrl_POS* AND *OvCtrl_NEG* EXAMPLE

Input A						Input B						Result
0	0	0	0	0	0	1	1	1	1	0	0	Negative
0	0	0	1	1	0	1	1	1	1	1	0	Positive

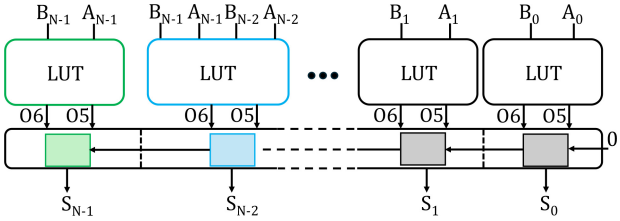


Fig. 6. *N* × *N_N OvRec_AccMSB* adder structure.

The *OvCtrl_POS* and *OvCtrl_NEG* overflow-safe adders have some limitations, which contribute to reducing the application-level accuracy of SNNs. For example, for *OvCtrl_NEG* adder, the result of performing $X - X$ is not equal to zero. To overcome the limitations of *OvCtrl_POS* and *OvCtrl_NEG* architectures, we present the *OvRec_AccMSB* overflow-safe approximate adder. Fig. 6 presents the generic structure of an $N \times N\_N$ *OvRec_AccMSB* adder. Compared to the *OvCtrl_POS* and *OvCtrl_NEG* designs, this architecture is based on the carry chains of the FPGAs. For operands with different signs, this architecture behaves like the base *OvErr_BASE* architecture. However, for operands with the same sign (either both positive or both negative), this architecture employs different functions for the two most significant LUTs. In particular, the second most significant LUT and the associated carry chain element (highlighted by the blue color in Fig. 6) always generate a 0 carry-out and forward it to the carry chain element of the most significant LUT. In the case of positive operands, the most significant LUT (highlighted by the green color) uses the *O6* output to forward a 0 to the associated carry chain element. The carry chain element performs an XOR operation on the *O6* and the carry-in (which is 0) to produce a 0. Similarly, in the case of negative operands, the most significant LUT forwards a 1 to the carry chain element using the *O6* line. The carry chain element performs an XOR operation on the *O6* and the carry-in (which is 0) to produce a 1. When dealing with operands with the same sign, the output S_{N-2} produced by the second most significant LUT and the associated carry chain is approximate. As explained, this is due to the fact that in such cases, the second most significant LUT and the associated carry chain are dedicated to generating and forwarding a 0 carry-out to the following carry chain element.

C. Approximation by Circuit Pruning

1) Pruning-Aware INIT-Value Exploration: The circuit pruning techniques, such as those presented in AppAxO [9], do not account for mitigating the pruning-induced output errors. For instance, Fig. 7 provides an example of an AppAxO-based approximate signed adder that demonstrates this issue. In the shown approximate adder, the LUT that processes

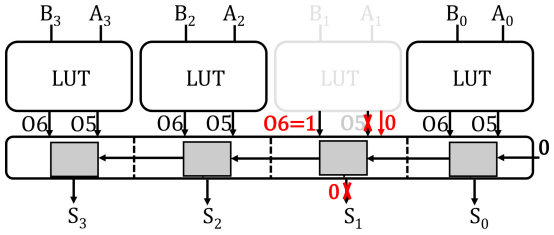


Fig. 7. AppAxO pruning technique-based an approximate 4 × 4_4 signed adder [9].

inputs A_1 and B_1 has been pruned, and as a result, the associated carry chain element does not contribute to the computation of the sum bit S_1 or generate an output carry for the following location. In this approximate design, the output carry generated by the least significant LUT and associated carry chain element (from processing A_0 and B_0) is forwarded to the carry chain element processing inputs A_2 and B_2. However, the propagation of wrong carries can produce AxOs, which are more susceptible to producing arithmetic overflows in general.

In our proposed AxOs generation methodology, referred to as *AxOSpike*, we take advantage of the available LUT input pins to introduce redundancy and mitigate the impact of pruning-induced errors. Fig. 8 shows an example of *AxOSpike*'s error-mitigation technique for the approximate adder presented in Fig. 7. In this technique, every LUT (except for the most significant LUT) also receives the inputs of the proceeding LUT. For example, the least significant LUT receives inputs A_0, B_0, and A_1, B_1. Moreover, the location of every LUT is identified by an N-bit binary string. For example, for the $4 \times 4\_4$ adder in the example, we will use a 4-bit string 1101. The 0 in the binary string identifies that the second least significant LUT has been pruned away. In *AxOSpike*, every LUT also receives the pruning status of the following LUT. For example, the least significant LUT receives E_1 as the fifth input. As the second least significant LUT is pruned (binary string 1101), the E_1 input will be set to 1. With the redundant inputs and information about the pruning status of proceeding LUT, different INIT[3] values can be explored for every LUT to mitigate the impact of pruning-induced errors. In our proposed work, we explored different INIT values to predict the input carry for a carry chain element following a pruned location. For example, for the approximate adder shown in Fig. 8, the least significant LUT will predict the input carry for the carry chain element associated with inputs A_2 and B_2. In this process, the accuracy of output S_0 can be traded to predict the correct carry for the following locations. For the current work, we have selected two INIT values denoted as *v1*(hexadecimal value $X'6666666688888888$) and *v2* (hexadecimal value $X'666606608888F880$). It should be noted that INIT values exploration provides a large design space, and there can be other possible INIT values that provide better-error recovery for the approximate adders.

[3]The function implemented by a LUT is represented by a 64-bit INIT value. Please see Xilinx Configurable Logic Block User Guide for more details.

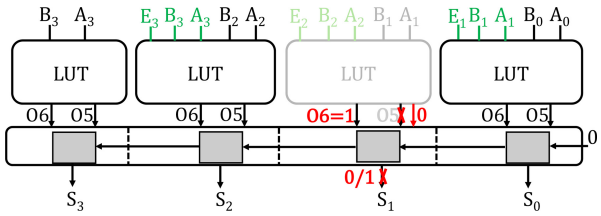
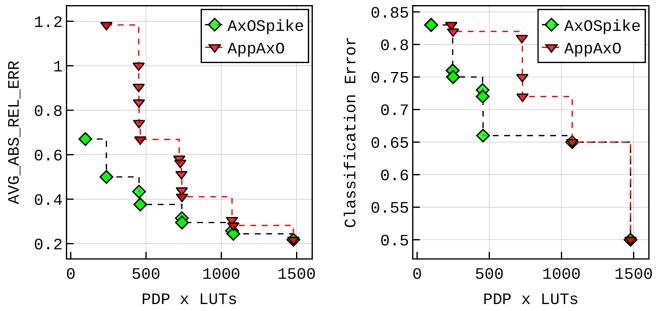


Fig. 8. *AxOSpike*: Redundant inputs-based $N \times N\_N$ adder to mitigate pruning-induced errors.



(a) (b)

Fig. 9. Comparing the pruning-based DSE performance of AppAxO [9] and *AxOSpike* for the same design: 7×7_7 *OvRec_AccMSB* adder. (a) Operator-level Pareto fronts. (b) Application-level Pareto fronts.

2) Enhanced Automated Circuit Pruning: The AxOs that are based on logic pruning usually remove the computational blocks by setting them to a constant value of 0. For example, the output S_1 is truncated to 0 in the AppAxO pruning-based approximate adder in Fig. 7. However, in our experiments, we have identified that truncating computational blocks (LUTs and carry chain elements) and replacing their functionality with a constant 1 can significantly improve the output accuracy in some cases. As observed in Fig. 8, *AxOSpike* provides the opportunity to replace the truncated output S_1 by either 0 or 1. This additional degree of freedom also significantly increases the AxOs design space. For example, for the $4 \times 4\_4$ approximate adder presented in Fig. 7, AppAxO provides $2^4 - 1 = 15$ different approximate versions, whereas *AxOSpike* provides $3^4 - 1 = 80$ approximate versions for the architecture shown in Fig. 8.

The *AxOSpike* pruning technique can be applied to all approximate adders presented in this work. To highlight the efficacy of *AxOSpike*'s generated AxOs, we compare them with AppAxO-generated approximate adders. Fig. 9 shows this comparison on both operator- and application-level for $7 \times 7\_7$ approximate adders. For this comparison, we have considered only the *OvRec_AccMSB* designs. As shown in Fig. 9(a), the nondominated design points provided by *AxOSpike* provide a better-accuracy-performance tradeoff with 24% higher hypervolume of the resulting Pareto front. Similarly, the utilization of these approximate adders for the classification of the MNIST dataset using the SNN model discussed in Section III shows that *AxOSpike*-generated AxOs also contributes to better-accuracy-performance tradeoffs resulting in 51% higher hypervolume.

TABLE IV
DESIGNS USED FOR EXPERIMENTAL EVALUATION OF *AxOSpike*

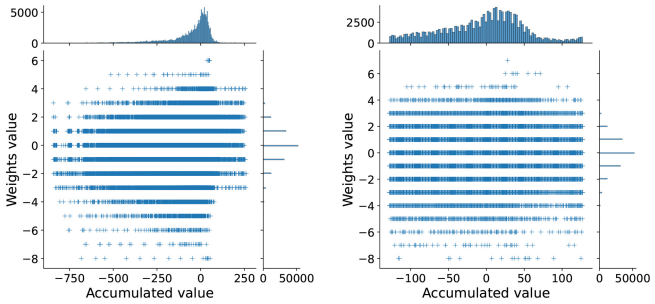
Accumulator Precision	Design Type	Pruning-aware versions	# Op-level characterizations	# App-level characterizations
6x6_6	OvErr_BASE	v1, v2	665 x 2	665 x 2
	OvCtrl_POS	v1, v2	211 x 2	211 x 2
	OvCtrl_NEG	v1, v2	211 x 2	211 x 2
	OvRec_AccMSB	v1, v2	65 x 2	65 x 2
7x7_7	OvErr_BASE	v1, v2	2059 x 2	2059 x 2
	OvRec_AccMSB	v1, v2	211 x 2	211 x 2
8x8_8	OvErr_BASE	v1, v2	6305 x 2	1
	OvCtrl_POS	-	1	1
	OvCtrl_NEG	-	1	1
	OvRec_AccMSB	v1, v2	665 x 2	665 x 2
9x9_9	OvErr_BASE	-	1	1
	OvRec_AccMSB	-	1	1
10x10_10	OvErr_BASE	-	1	1
	OvRec_AccMSB	-	1	1

V. EXPERIMENTS AND RESULTS

A. Experiment Setup

All the arithmetic operators implemented in the current work are designed in VHDL and synthesized for the 7*VX*330*T* device of the Virtex-7 family using AMD Xilinx Vivado 2020.2. The dynamic power is computed by recording the dynamic switching activity for all possible input combinations of the multiplier configurations. For this purpose, we have used the Vivado Simulator and Power Analyzer tools. The behavioral characterization of the operators was based on the results of simulating every possible input combination of the operator, with the implemented design. The SNN model is implemented in C++ and Python using PyBind11. PyTorch and snnTorch [34] were used for ML-related functionality and datasets, and the image-to-spike conversions. The same threshold value has been used across all neurons in an SNN.

Table IV shows the different accumulator designs used in the experimental evaluation. The precision of the accumulator is varied from 6 bits to 10 bits. While the *OvErr_BASE* and *OvRec_AccMSB* designs were analyzed across different precision, the *OvCtrl_POS* and *OvCtrl_NEG* design versions were analyzed only for the 6- and 8-bit operators. Further, we implemented automated circuit pruning for some of the design versions. For the circuit pruning-based operators, we have also explored the impact of utilizing the two INIT values of LUTs, i.e., *v1* ($X'6666666688888888$) and *v2* ($X'666606608888F880$), on improving their output accuracy. Since we use exhaustive sampling, design versions that result in a large number of AxO designs were not used for pruning experiments. However, more intelligent search methods can be implemented in the current framework for the DSE in such large design spaces. The last two columns in the table show the number of designs used for the operator-level and application-level characterization. While the operator-level analysis involves determining PPA and various error metrics, application-level analysis involves only behavioral analysis. As a result, all the PPA metrics mentioned in the subsequent results refer to the hardware characterization of the approximate accumulator design.



(a) (b)

Fig. 10. Joint distribution of operands in the neuron's accumulator for inference on a single image. (a) Distribution assuming zero overflow during accumulation. (b) Distribution when using the 8×8_8 *OvErr_BASE* adder as accumulator.

B. SNN Model Analysis

We have used 4-bit weights for the SNN implemented in our current work. Even with the 4-bit quantization of the weights, the network can reach nearly 98% accuracy, assuming no overflow occurs during accumulation. For instance, Fig. 10(a) shows the distribution of the operand values of the accumulation in all the neurons for a single inference, assuming all additions are overflow-safe. The vertical axis corresponds to the weights and follows the distribution of the 4-bit weights. The horizontal axis, showing the distribution of the current membrane voltage value as the second operand of the accumulator, ranges from -800 to 300. When using the 8×8_8 *OvErr_BASE* adder, the accuracy drops to around 85%. This drop in accuracy due to the overflow is clear from the changed distribution of the operands shown in Fig. 10(b).

The accuracy of the SNN can also vary depending upon the threshold value used in the model, and the number of timesteps used in the input encoding. Fig. 11 shows the variation of the SNN's accuracy using the 8×8_8 *OvErr_BASE* adder for varying threshold values and number of timesteps. While the accuracy varies considerably with the threshold, it stays fairly stable over the different number of timesteps. The boxplot in the figure shows the distribution of the maximum accuracy for the different number of timestep experiments. The maximum accuracy varies by less than 0.1% across the 10 different timestep experiments. The threshold value for the neuron can be viewed as a trainable parameter of the SNN model, with some related works exploring joint optimization of the weights and the threshold value. However, since we have used post-training quantization of the weights, we have employed a sweep of the threshold values to demonstrate how the threshold value can be used to recover the loss in accuracy due to quantization (and approximation) to some extent. For the subsequent experiments related to the SNN behavior analysis, we report the maximum classification accuracy over varying threshold values for 10 timesteps.

C. Application-Specific Operator Modeling

1) Operator-Level Analysis: In our current work, we have presented four different design variants, including the baseline *OvErr_BASE* design, that provide varying error tradeoffs.

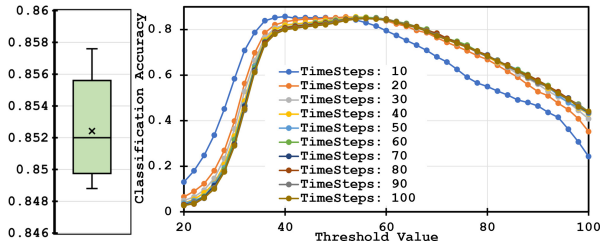


Fig. 11. SNN accuracy variation with threshold value and the number of timesteps using the 8×8_8 *OvErr_BASE* adder as accumulator.

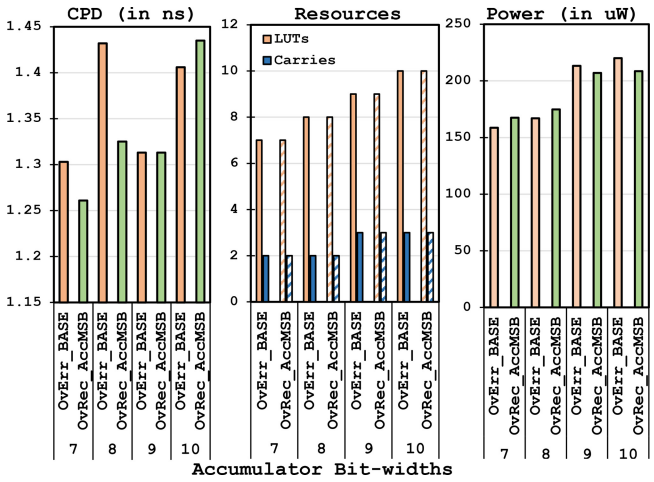


Fig. 12. Comparison of the PPA metrics for the *OvErr_BASE* and *OvRec_AccMSB* adders for different bit-width $N \times N\_N$ adders.

TABLE V
OPERATOR-LEVEL ERROR METRICS VARIATION IN THE DIFFERENT VARIANTS OF THE 6 × 6_6 ADDER

Operator Type / Error Metric	*OvErr_BASE*	*OvCtrl_NEG*	*OvCtrl_POS*	*OvRec_AccMSB*
AVG_ERR	- 0.5	1.5	- 2.5	0
AVG_ABS_ERR	16	9.75	10.25	8
AVG_REL_ERR	0.39	0.9	1.24	0.22
AVG_ABS_REL_ERR	0.39	0.9	1.24	0.22
MAX_ERR	64	32	32	32
MIN_ERR	- 64	- 32	- 32	- 32
PROB_ERR	25	30.47	32.03	37.5

Table V shows the values of the operator-level statistical error metrics of each 6×6_6 adder, compared to the 6×6_7 overflow-safe adder. These error metrics are commonly employed to characterize the output quality of approximate circuits by comparing the approximate outputs with the accurate outputs, indicating the magnitude and frequency of errors [35]. The *OvErr_BASE* design has the largest value of the minimum and maximum error magnitudes along with the lowest-error probability. Similarly, the *OvRec_AccMSB* design has the lowest-average error, average absolute error, average relative error, and the lowest-average absolute relative error, albeit with the highest probability of error. The *OvRec_AccMSB*, *OvCtrl_NEG*, and *OvCtrl_POS* designs have similar maximum and minimum error values. However, in the application-specific analysis, we observed very few benefits with the *OvCtrl_NEG* and *OvCtrl_POS* versions (shown later), especially for higher-bit widths. Therefore, we limit the

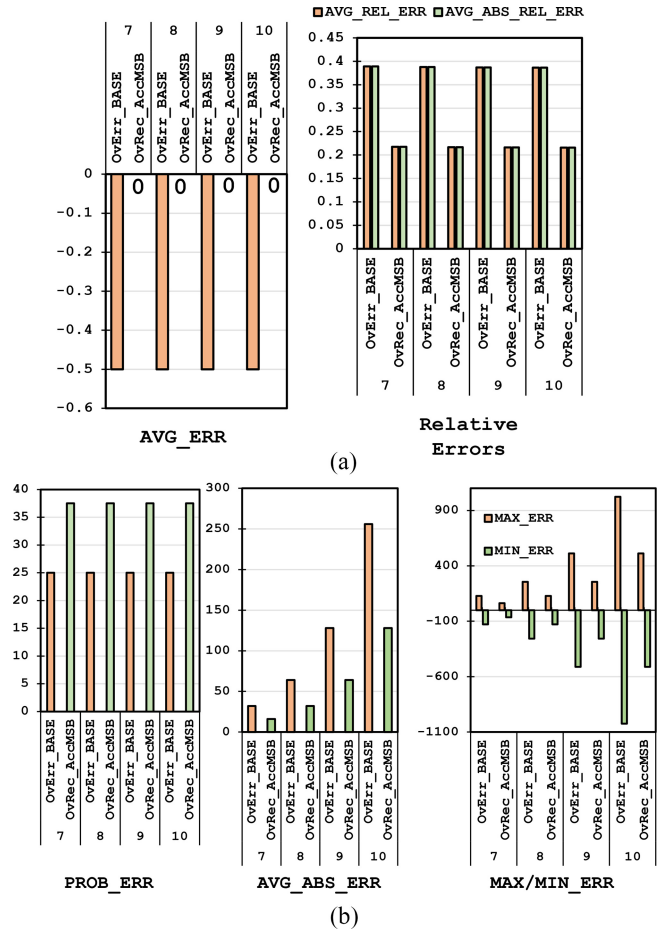


Fig. 13. Comparison of the operator-level error metrics for the *OvErr_BASE* and *OvRec_AccMSB* adders for different bit-width $N\times N\_N$ adders. (a) Average error and relative error metrics. (b) Probability of error, average absolute error, and range of errors.

further analysis to the comparison of the *OvErr_BASE* and *OvRec_AccMSB* versions.

Fig. 12 shows the comparison of the PPA metrics for different bit width of the $N\times N\_N$ *OvErr_BASE* and *OvRec_AccMSB* adders. Expectedly, resource utilization and power dissipation rise with increasing bit widths. Also, the *OvErr_BASE* and *OvRec_AccMSB* designs show similar LUT and CC utilization. However, the power and CPD vary slightly between the two design variants. This can be attributed to the different architecture of the *OvRec_AccMSB* design. Furthermore, the behavior of the designs also varies depending on the sign signals of the operands. The comparison of the operator-level behavior of the *OvErr_BASE* and *OvRec_AccMSB* designs is shown in Fig. 13. The *OvRec_AccMSB* designs exhibit zero average error across different precisions. Except for increased probability of error, the *OvRec_AccMSB* designs exhibit better-error metrics than *OvErr_BASE* across different bitwidth $N\times N\_N$ adders.

2) Application-Level Analysis: We experimented with using the proposed design variants of the $N\times N\_N$ adders as the accumulators in the SNN's neurons. While the *OvErr_BASE* and *OvRec_AccMSB* variants exhibited increasing accuracy with increasing bit-width (N), the *OvCtrl_POS* and *OvCtrl_NEG* versions did not show such patterns. Fig. 14

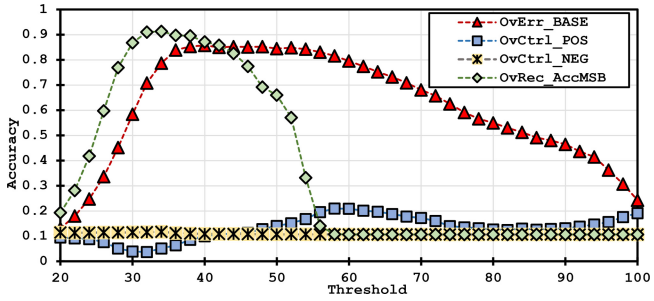


Fig. 14. Variation of classification accuracy with the threshold value for different design variants of 8×8_8 adders used as an accumulator in the SNN's neuron.

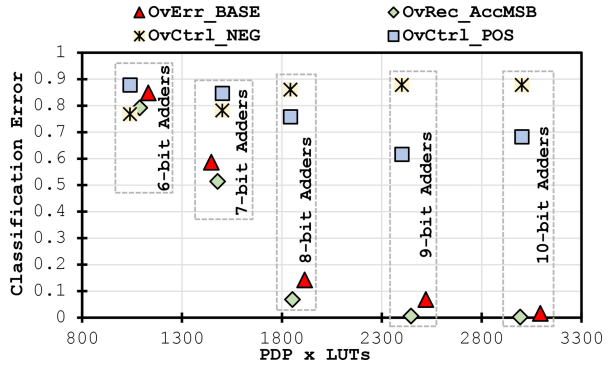


Fig. 15. Comparison of the PPA-error tradeoffs of different $N\times N\_N$ adders when used as accumulators in the SNN's neurons.

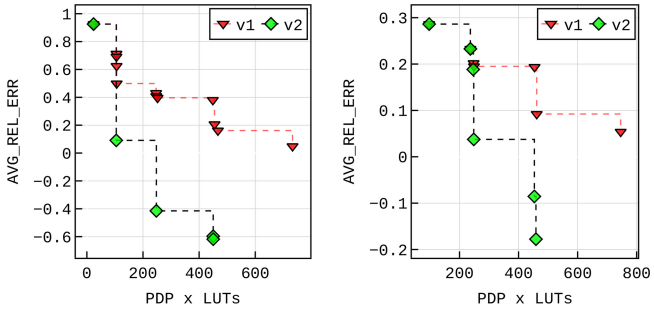


Fig. 16. Comparing the operator-level DSE performance of v2 and v1 for the 7×7_7 adder designs. (a) *OvErr_BASE*. (b) *OvRec_AccMSB*.

shows the variation of the resulting SNN's classification accuracy with different threshold values for the four 8×8_8 adder designs. As can be seen from the figure, the accuracy of the *OvCtrl_POS* and *OvCtrl_NEG* designs achieves a maximum of around 20% accuracy. The comparison of the PPA-error tradeoffs across different bit width operators is shown in Fig. 15. The SNN's classification error decreases with increasing bit width for both *OvErr_BASE* and *OvRec_AccMSB* adders. The proposed *OvRec_AccMSB* design shows lower error than *OvErr_BASE* across all bit widths. The 8- and 9-bit *OvRec_AccMSB* adders result have similar accuracy as the 9- and 10-bit *OvErr_BASE* adders, respectively. This amounts to 26.5% and 20.93% lower PDPxLUTs for similar accuracy, respectively, with the *OvRec_AccMSB* adders.

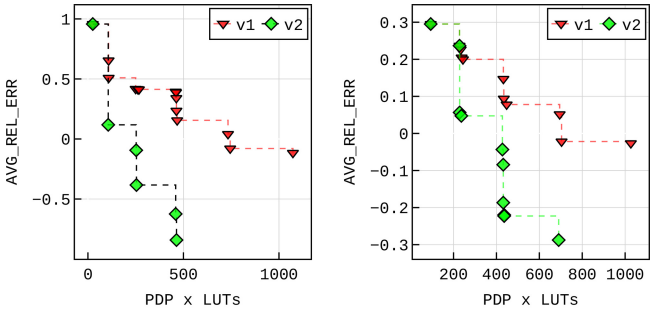


Fig. 17. Comparing the operator-level DSE performance of v2 and v1 for the 8×8_8 adder designs. (a) *OvErr_BASE*. (b) *OvRec_AccMSB*.

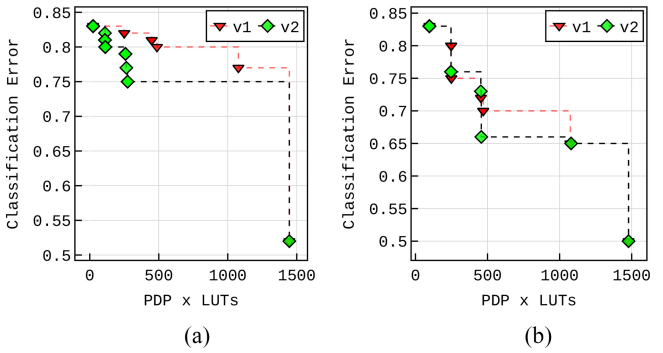


Fig. 18. Comparing the SNN-level performance of *v2* and *v1* for the 7×7_7 adder designs. (a) *OvErr_BASE*. (b) *OvRec_AccMSB*.

D. Pruning-Aware Operator Modeling

In addition to the operator modeling for SNN-driven error recovery, we also propose generic pruning-aware optimization to the operator model. Fig. 16 shows the Pareto front analysis for operator-level results of 7×7_7 adder AxOs generated using circuit pruning. The *v2* plots include the proposed optimizations. As seen in the figure, *v2* results in considerably better designs than *v1* for both *OvErr_BASE* and *OvRec_AccMSB* versions. An analysis of the combined Pareto front shows only 1 and 2 AxOs belong to *v1* compared to 6 and 7 designs from *v2* in Fig. 16(a) and (b), respectively. Similarly, Fig. 17 shows the results of 8×8_8 AxOs. Here too, we observed much improved Pareto front designs with the proposed *v2* operator model. In the corresponding combined Pareto fronts, 1 and 2 designs belong to *v1*, compared to 7 and 11 AxOs with *v2* for *OvErr_BASE* and *OvRec_AccMSB* design variants, respectively.

Fig. 18 shows the application-level Pareto front analysis for 7×7_7 adders. Here too, we observe improved quality of results with the pruning-aware *v2* model. However, with the *OvRec_AccMSB* version in 8×8_8 designs, we observed better-Pareto front designs with *v1*, as seen in Fig. 19(a). Fig. 19(b) compares the resulting hypervolume of the Pareto fronts for application-level analysis. For the 7×7_7 designs, the *v2* version AxOs provide almost similar hypervolume as from the combined Pareto front. While the individual hypervolume of *v2* designs for 8×8_8 AxOs is lower, it still contributes additional design points to the Pareto front. Across

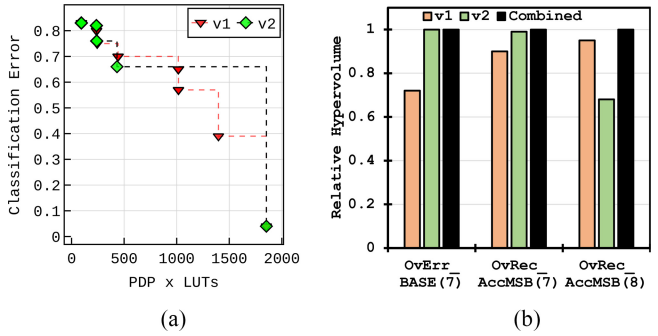


Fig. 19. Application-level results for pruning-aware models of 8×8_8 adder *OvRec_AccMSB*. (a) Comparing Pareto fronts. (b) Comparing Pareto-front hypervolume.

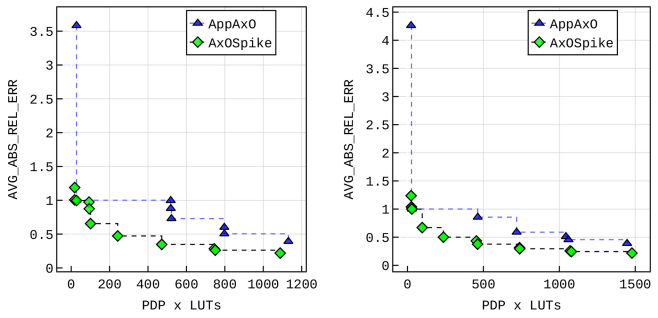


Fig. 20. Comparing the operator-level metrics of approximate 6×6_6 and 7×7_7 adder designs from *AxOSpike* and AppAxO. (a) 6×6_6 adders. (b) 7×7_7 adders.

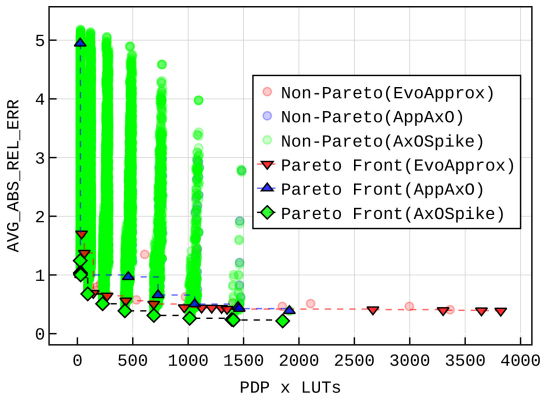


Fig. 21. Comparing the operator-level metrics of approximate 8×8_8 adder designs from *AxOSpike*, AppAxO, and EvoApprox.

experiments for different operators, we observe up to 102.1% improvement in the Pareto from hypervolume due to *v2*.

E. Comparing With State-of-the-Art

For comparing *AxOSpike* with related state-of-the-art works, we chose EvoApprox [5] and AppAxO [9] as the set of relevant works. This approach subsumes the comparison with other related works, such as CoOAx [17], which is based on a similar methodology as AppAxO and approxFPGAs [10], which uses the designs present within the scope of EvoApprox for DSE. Further, 8×8_8 adders are the only relevant designs for the current work provided in the EvoApprox library.

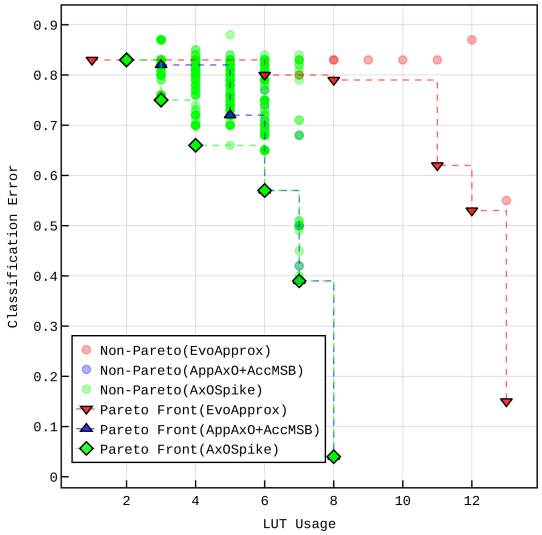


Fig. 22. SNN's classification Error versus LUT Utilization for approximate 8×8_8 adders used as accumulators in the SNN's neuron.

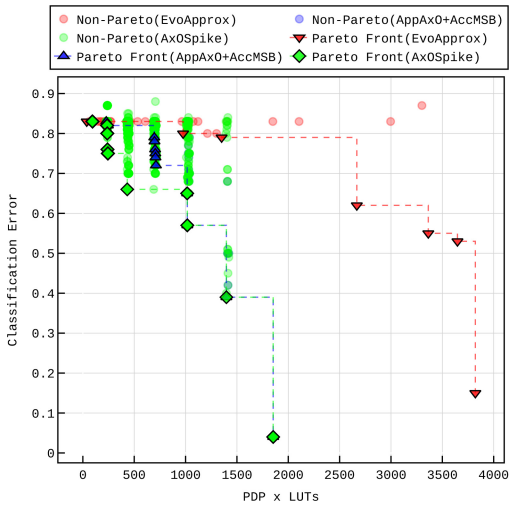


Fig. 23. SNN's classification error versus PDP×LUTs for approximate 8×8_8 adders used as accumulators in the SNN's neuron.

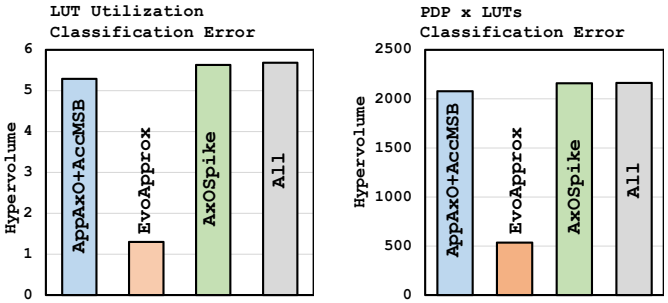


Fig. 24. Comparing the hypervolume for approximate 8×8_8 adders used as accumulators in the SNN's neuron.

1) Operator-Level: Fig. 20 shows the Pareto front analysis of the 6×6_6 and 7×7_7 approximate adders generated with the AppAxO methodology and those using *AxOSpike*. As can be seen in the figures, *AxOSpike* generates designs with better-PPA-error tradeoffs than AppAxO. With the combined Pareto front analysis, we do not observe any designs from AppAxO contributing to the Pareto front. However, this is expected as the enhanced automated circuit pruning of *AxOSpike* subsumes that of AppAxO. Even in the case of 8×8_8 designs, including the designs from EvoApprox, all the dominant points in the combined Pareto front are the result of *AxOSpike*'s proposed modifications, as seen in Fig. 21. In addition to showing the efficacy of the presently proposed methods, it also shows the importance of integrating FPGA-specific optimizations into the operator model. Unlike *AxOSpike* and AppAxO, the designs in the EvoApprox library are optimized for ASIC implementation. and fail to leverage the FPGA-based structures for any error recovery.

2) Application-Level: For the application-level comparison with related state-of-the-art works, we used the 8×8_8 designs from EvoApprox. Additionally, we combined the pruning methodology of AppAxO along with our proposed *OvRec_AccMSB* design for another comparison point - AppAxO+AccMSB. Fig. 22 shows the Pareto fronts of the candidate designs while considering the LUT usage of the adders along with the SNN's classification accuracy. Similar to the operator-level results, we observe higher-quality designs generated using *AxOSpike*. The combined Pareto front shows a total of 42 AxO designs, with EvoApprox and AppAxO+AccMSB contributing just 1 and 3 designs, respectively. A similar analysis while considering the adders' PDPxLUTs metrics, shown in Fig. 23, shows 1 and 4 designs in the combined Pareto front resulting from EvoApprox and AppAxO+AccMSB, respectively, out of a total of 98 points. Fig. 24 shows the comparison of the Pareto front hypervolume corresponding to Figs. 22 and 23. As evident, *AxOSpike* results in much better designs than EvoApprox. Further, the results show that the proposed *OvRec_AccMSB* design can be combined with any complementary circuit pruning methodology, such as AppAxO, to provide better-quality designs than state-of-the-art methods.

VI. Conclusion

With the rising complexity of edge AI, novel approaches spanning across the computing stack need to be developed. SNNs and approximate arithmetic form two such emerging computing paradigms at the algorithm and the circuit layer, respectively. SNN-based processing is inherently error tolerant and AxC should be able to leverage such error resilience most effectively. However, a bottom-up approach to designing AxOs can lead to limited benefits. In this current article, we present novel methods of integrating both application-specific and hardware platform-specific information into the operator model, thereby enabling the search for considerably better-quality designs with automated circuit pruning. With the proposed techniques, we report designs with up to 26.5% lower PDPxLUTs with similar application-level accuracy. Further, we report a considerably better set of design points than related works with up to 51% higher-Pareto front hypervolume. The current work can be extended to include more complex SNN neuron models and can benefit from more intelligent automated DSE, especially for larger bit-width operators.

REFERENCES

[1] J. D. Nunes, M. Carvalho, D. Carneiro, and J. S. Cardoso, "Spiking neural networks: A survey," *IEEE Access*, vol. 10, pp. 60738–60764, 2022.

[2] W. Liu, F. Lombardi, and M. Shulte, "A retrospective and prospective view of approximate computing [point of view]," *Proc. IEEE*, vol. 108, no. 3, pp. 394–399, Mar. 2020.

[3] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–33, Mar. 2016.

[4] P. Stanley-Marbell et al., "Exploiting errors for efficiency: A survey from circuits to applications," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–39, 2020.

[5] V. Mrazek et al., "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2017, pp. 258–261.

[6] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *Proc. 52nd Annu. Design Autom. Conf.*, New York, NY, USA, 2015, pp. 1–6.

[7] B. S. Prabakaran et al., "DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 917–920.

[8] S. Ullah, H. Schmidl, S. S. Sahoo, S. Rehman, and A. Kumar, "Area-optimized accurate and approximate softcore signed multiplier architectures," *IEEE Trans. Comput.*, vol. 70, no. 3, pp. 384–392, Mar. 2021.

[9] S. Ullah, S. S. Sahoo, N. Ahmed, D. Chaudhury, and A. Kumar, "*AppAxO*: Designing application-specific approximate operators for FPGA-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 3, pp. 1–31, 2022.

[10] B. S. Prabakaran, V. Mrazek, Z. Vasicek, L. Sekanina, and M. Shafique, "ApproxFPGAs: Embracing ASIC-based approximate arithmetic components for FPGA-based systems," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[11] L. Deng, "The MNIST database of handwritten digit images for machine learning research [Best of the Web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[12] S. Ullah, S. S. Sahoo, and A. Kumar, "CLAppED: A design framework for implementing cross-layer approximation in FPGA-based embedded systems," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, 2021, pp. 475–480.

[13] S. Ullah, S. Rehman, M. Shafique, and A. Kumar, "High-performance accurate and approximate multipliers for FPGA-based hardware accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 2, pp. 211–224, Feb. 2022.

[14] S. Ullah et al., "Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[15] S. Ullah, S. S. Murthy, and A. Kumar, "SMApproxlib: Library of FPGA-based approximate multipliers," in *Proc. 55th Annu. Design Autom. Conf.*, New York, NY, USA, 2018, pp. 1–6.

[16] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.

[17] S. Ullah, S. S. Sahoo, and A. Kumar, "CoOAx: Correlation-aware synthesis of FPGA-based approximate operators," in *Proc. Great Lakes Symp. VLSI*, 2023, pp. 671–677.

[18] V. Mrazek, M. A. Hanif, Z. Vasicek, L. Sekanina, M. Shafique, "AutoAx: An automatic design space exploration and circuit building methodology utilizing libraries of approximate components," in *Proc. 56th Annu. Design Autom. Conf.*, New York, NY, USA, 2019, pp. 1–6.

[19] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2016, *arXiv:1510.00149*.

[20] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2017, *arXiv:1608.08710*.

[21] S. Vadera and S. Ameen, "Methods for pruning deep neural networks," *IEEE Access*, vol. 10, pp. 63280–63300, 2022.

[22] V. Camus, C. Enz, and M. Verhelst, "Survey of precision-scalable multiply-accumulate units for neural-network processing," in *Proc. IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2019, pp. 57–61.

[23] "Mixed precision." TensorFlow. Accessed: Jun. 9, 2021. [Online]. Available: https://www.tensorflow.org/guide/mixed_precision

[24] G. Xiao, C. Yin, T. Zhou, X. Li, Y. Chen, K. Li, "A survey of accelerating parallel sparse linear algebra," *ACM Comput. Surv.*, vol. 56, no. 1, pp. 1–38, Aug. 2023.

[25] S. Panchapakesan, Z. Fang, and J. Li, "SyncNN: Evaluating and accelerating spiking neural networks on FPGAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 15, no. 4, pp. 1–27, Dec. 2022.

[26] F. Corradi, G. Adriaans, and S. Stuijk, "Gyro: A digital spiking neural network architecture for multi-sensory data analytics," in *Proc. Drone Syst. Eng. Rapid Simul. Perform. Eval., Methods Tools*, New York, NY, USA, 2021, pp. 9–15.

[27] J. Mack et al., "RANC: Reconfigurable architecture for neuromorphic computing," *Trans. Comp.-Aided Des. Integr. Circuits Sys.*, vol. 40, no. 11, pp. 2265–2278, Nov. 2021.

[28] M. Isik, "A survey of spiking neural network accelerator on FPGA," 2023, *arXiv:2307.03910*.

[29] S. Sen, S. Venkataramani, and A. Raghunathan, "Approximate computing for spiking neural networks," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2017, pp. 193–198.

[30] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *J. Physiol.*, vol. 117, no. 4, pp. 500–544, 1952.

[31] N. Brunel and M. C. W. Van Rossum, "Lapicque's 1907 paper: From frogs to integrate-and-fire," *Biol. Cybernet.*, vol. 97, no. 5, pp. 337–339, 2007.

[32] L. É. Lapicque, "Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization," *J. Physiol. Pathol. Gén.*, vol. 9, pp. 620–635, 1907.

[33] A. Carpegna, A. Savino, and S. Di Carlo, "Spiker: An FPGA-optimized hardware accelerator for spiking neural networks," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2022, pp. 14–19.

[34] J. K. Eshraghian et al., "Training spiking neural networks using lessons from deep learning," *Proc. IEEE*, vol. 111, no. 9, pp. 1016–1054, Sep. 2023.

[35] S. Ullah and A. Kumar, *Approximate Arithmetic Circuit Architectures for FPGA-Based Systems*. Cham, Switzerland: Springer, 2023, pp. 27–40. [Online]. Available: https://www.springerprofessional.de/en/approximate-arithmetic-circuit-architectures-for-fpga-based-syst/24072054