SCIENCE CHINA
Information Sciences

CrossMark click for updates

• RESEARCH PAPER •

June 2023, Vol. 66 162404:1–162404:20
https://doi.org/10.1007/s11432-021-3519-1

Reconfigurable spatial-parallel stochastic computing for accelerating sparse convolutional neural networks

Zihan XIA[1], Rui WAN[1], Jienan CHEN[1*] & Runsheng WANG[2]

[1]*National Key Laboratory of Science and Technology on Communications, University of Electronic Science and Technology of China, Chengdu* 611731*, China;*
[2]*School of Integrated Circuits, Peking University, Beijing* 100871*, China*

Received 3 November 2021/Revised 17 January 2022/Accepted 20 March 2022/Published online 17 May 2023

Abstract Edge devices play an increasingly important role in the convolutional neural network (CNN) inference. However, the large computation and storage requirements are challenging for resource- and power-constrained hardware. These limitations might be overcome by exploring the following: (a) error tolerance via approximate computing, such as stochastic computing (SC); (b) data sparsity, including the weight and activation sparsity. Although SC can perform complex calculations with compact and simple arithmetic circuits, traditional SC-based accelerators suffer from the low reconfigurability and long bitstream, further making it difficult to benefit from the data sparsity. In this paper, we propose spatial-parallel stochastic computing (SPSC), which improves the spatial parallelism of the SC-based multiplier to the full extent while consuming fewer logic gates than the fixed-point implementation. Moreover, we present SPA, a highly reconfigurable SPSC-based sparse CNN accelerator with the proposed hybrid zero-skipping scheme (HZSS), to efficiently take advantage of different zero-skipping strategies for different types of layers. Comprehensive experiments show that SPA with up to 2477.6 Gops/W outperforms existing several binary-weight accelerators, SC-based accelerators, and the sparse CNN accelerator considering energy efficiency.

Keywords convolutional neural networks, stochastic computing, sparse neural networks, energy-efficient accelerator, high reconfigurability, spatial parallelism

Citation Xia Z H, Wan R, Chen J N, et al. Reconfigurable spatial-parallel stochastic computing for accelerating sparse convolutional neural networks. Sci China Inf Sci, 2023, 66(6): 162404, https://doi.org/10.1007/s11432-021-3519-1

1 Introduction

Convolutional neural networks (CNNs), a key branch of deep learning [1], have shown success in many intelligent tasks, such as image classification [2,3] and object detection [4]. However, state-of-the-art neural networks improve the accuracy at the cost of billions of operations, hundreds of megabytes of parameters, and intermediate activations [5]. The increasing computing and storage requirements hamper low-power and real-time developments on energy-, memory-, and latency-constrained platforms, including wearable devices and embedded systems. Researchers have proposed various techniques, including exploring the error-tolerance property of neural networks by low-precision computing [6] (e.g., binary-weight neural networks [7], approximate computing [8, 9], and in-memory computing [10, 11]) as well as increasing the sparsity of weights by network pruning with negligible accuracy degradation [12, 13] to overcome this challenge. These algorithmic innovations theoretically reduce the computational complexity and storage requirement. However, specialized hardware to convert these valuable merits into high processing efficiency is explicitly demanded.

As a potential substitute for common binary computing and a paradigm of approximate computing, stochastic computing (SC) [14] provides a power-efficient way to represent numbers in the absence of exact computing. In SC, binary numbers are converted to bitstreams whose values are determined by the frequency of bit '1's. This serial and probabilistic nature enables the extremely low-cost implementation

* Corresponding author (email: Jesson.chen@outlook.com)

© Science China Press 2023

of some complex operations such as multiplications and nonlinear functions. SC is currently successfully applied to low-density parity-check (LDPC) decoder [15] and polar decoder [16].

SC has also been used to accelerate neural networks [17–21]. However, traditional SC-based accelerators usually adopt a fully parallel implementation architecture, assuming target neural networks have fixed configurations, and the area budget is unlimited. Therefore, the lack of reconfigurability impedes their applications on currently versatile deep learning tasks. Moreover, traditional SC-based accelerators suffer from long computational latency due to the serial bitstreams-based arithmetic circuits. This drawback might become salient for large neural networks with a relatively deep calculation path. We believe these problems must be solved before practically applying SC to the acceleration of neural networks.

Another popular method to address the vast computational and memory burden of neural networks is leveraging the data sparsity, which comes from two folds: the weight sparsity via network pruning and the input activation (IA) sparsity from the commonly adopted rectified linear unit (ReLU) forcing all negative numbers to zeros. A wide variety of accelerators for sparse CNNs have been proposed [22–25]. Eyeriss activates the clock-gating whenever a zero IA is detected in runtime to save energy [22]. However, Eyeriss cannot improve the throughput. EIE (efficient inference engine) [23] leverages the benefits of IA and weight sparsity but is limited only to accelerate fully connected layers (FCLs), which account for only a small part of multiplications in popular CNNs such as AlexNet [2] and VGGNet [26]. SCNN (sparse CNN) is the first one to skip IAs and weights for convolutional layers (CLs) [24]. SCNN proposes the Cartesian product, namely the pixel dimension-first dataflow, wherein a weight vector with size F and an IA vector with size I are delivered to a multiplier array with size $F \times I$. The pixel dimension-first dataflow improves the multiplier utilization, but suffers from heavy partial sums (psums) writeback traffic. Moreover, SCNN only addresses CLs. By contrast, SNAP (sparse neural acceleration processor) [25] transfers the issue of complex writeback traffic to the searching and pairing of valid weight-IA pairs, which are performed by an associative index matching module. However, few existing studies have considered combining efficient designs of low-level arithmetic circuits with high-level zero-skipping architectures to achieve global optimization.

With the above considerations, we present a reconfigurable CNN processor, namely SPA (spatial-parallel stochastic computing for accelerating sparse convolutional neural networks), which is optimized from the microscope (i.e., low-cost arithmetic circuits) to the macroscope (i.e., efficient architectures) to leverage both the SC and data sparsity. Experiment results reveal that the proposed spatial-parallel stochastic computing (SPSC)-based arithmetic circuit exhibits 45.2%–18.9% less area and 38.7%–21.7% less power consumption than its traditional fixed-point counterpart. SPA consumes only 25.71 mW power with a small area of 4.08 mm^2, and achieves an energy efficiency of up to 2477.6 Gops/W. SPA also outperforms several existing CNN accelerators considering energy efficiency. The main contributions of our work are as follows.

(1) We propose SPSC to shorten the computational time of traditional SC to only one clock cycle while maintaining a low area cost. Following our previous work using wire density to mimic neural synaptic plasticity [27], we design a weight-adjustable, low-cost, high-accuracy SPSC-based arithmetic unit, which is further optimized by the proposed bi-directional sequence optimization (BSO).

(2) We introduce a hybrid zero-skipping scheme (HZSS) to use the sparsity of weights and IAs efficiently to accelerate CLs and FCLs, respectively. To the best of our knowledge, we are the first to skip different types of zero data in different types of neural network layers for the SC-based acceleration of CNNs. Furthermore, the serial channel and parallel pixel (SCPP) dataflow is designed to explore the parallelism and be compatible with the SPSC-based two-input vector-vector multiplier (SPSC-TVM). In addition, the magnitude-aware weights pairing (MAWP) algorithm is proposed to find proper weight pairs during compile time to meet the magnitude constraint of our SPSC-TVM.

(3) We propose a reconfigurable architecture with two specialized, loosely-coupled engines for CLs and FCLs. Moreover, the three-level time interleaving technique is proposed to maximize the time overlap between different levels of processing tasks.

2 Stochastic computing basics

In this section, we introduce the basic principles of SC, including the classical system structure of SC and the SC-based arithmetic circuits.

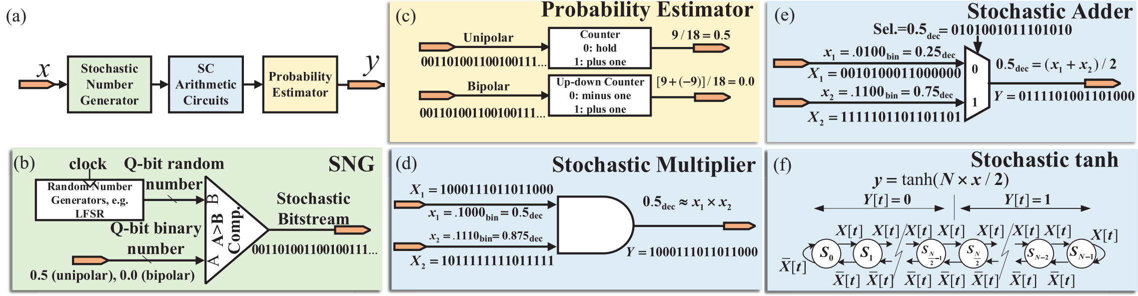


Figure 1 (Color online) Basic designs of the typical SC. (a) Structure of the SC system; (b) typical SNG; (c) typical probability estimator; (d) stochastic multiplier using the AND gate; (e) stochastic adder using the multiplexer; (f) finite state machine-based implementation of the complex function: tanh.

2.1 System structure of stochastic computing

In SC, a binary number is converted to a stochastic bitstream, in which the frequency of '1's determines its value depending on the predefined data range. Popular choices of the data range are $[0, 1]$ (unipolar) and $[-1, 1]$ (bipolar). SC can implement complex arithmetic functions with simple logical gates by processing serial bitstreams rather than binary numbers.

As shown in Figure 1(a), the structure of the SC system comprises stochastic number generators (SNGs) (i.e., the forward conversion), SC arithmetic circuits and probability estimators (i.e., the backward conversion). In Figure 1(b), the SNG comprises a random number generator and a comparator is used to convert binary numbers to stochastic bitstreams. In addition to the classical linear-feedback shift register (LFSR)-based SNGs, many other kinds of SNGs with improved accuracy, such as the Sobol sequence [28] and the Halton sequence [29], are proposed. The SC-based processing is then performed in the domain of stochastic bitstreams. Finally, the probability estimator converts stochastic bitstreams back into binary numbers. As shown in Figure 1(c), for instance, a counter/up-down counter can track the value of the stochastic sequence using unipolar/bipolar encoding. Another advanced implementation is the adaptive digital element-based probability estimator, which adopts a feedback-based design. Details can be found in the work of Brown [14].

2.2 SC-based arithmetic circuits

In SC, the multiplication is performed by an AND gate [14] (unipolar encoding). Typically, for a Q-bit fixed-point number $x \in [0, 1)$, 2^Q cycles are required to generate the stochastic bitstream $X[2^Q]$. The probability of $X[t] = 1$ at cycle t obeys $p_X = x$. Let X_1 and X_2 be two stochastic bitstreams applied to this AND gate. As shown in Figure 1(d), the output bit $Y[t]$ at cycle t is '1' only if two input bits $X_1[t]$ and $X_2[t]$ are both '1's. If two bitstreams are statistically independent, we have $p_Y = p_{X_1} \times p_{X_2}$. Therefore, the AND gate functions as a multiplier and can be orders of magnitude smaller than a typical binary multiplier. Moreover, like many other SC-based arithmetic circuits, the stochastic multiplier is highly error-tolerance because several bit errors in the stochastic bitstream have little impact on the value encoded into it.

The most common implementation of the stochastic adder is using a two-input multiplexer, where the probability of the selection signal is 0.5 [14]. However, unlike the stochastic multiplication, which is closed on the intervals $[-1, 1]$ for bipolar encoding and $[0, 1]$ for unipolar encoding, the stochastic adder cannot perform the exact addition without scaling. For example, in Figure 1(e), the output probability of Y is $(x_1 + x_2)/2$. The calculation sacrifices the resolution by half (i.e., one bit in fixed-point number), so the output signal needs to be re-scaled before performing the subsequent step.

Many interesting SC circuits are built upon finite state machines [14]. These designs usually have a saturating counter of which the state is controlled by every single bit $X[t]$ in the input stochastic bitstream X. Figure 1(f) depicts the state transition graph of the stochastic tanh function, where the input stochastic bitstream X is bipolar encoding. The output $Y[t]$ is '1' whenever the state is in the right half of N possible states. N is usually an even design parameter and impacts the computing accuracy. Compared with its fixed-point counterpart, the stochastic tanh requires much fewer hardware resources.

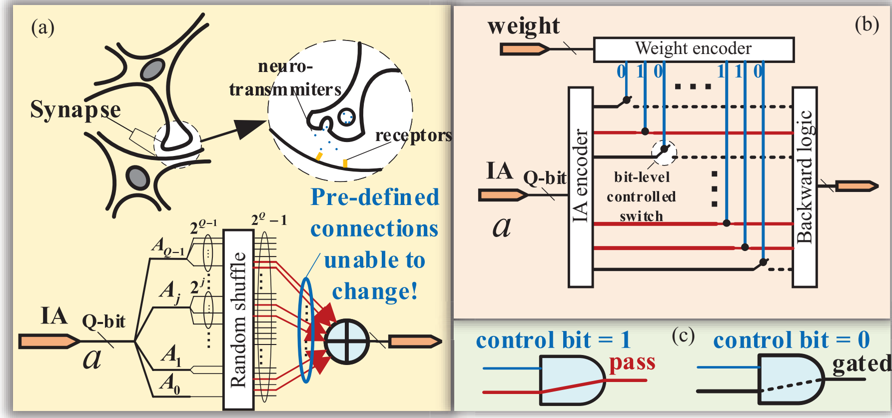


Figure 2 (Color online) Review of the NSPC-based multiplier and possible solutions to its limitations. (a) NSPC-based multiplier, which mimics the synaptic strength by the density of wires; (b) generalization of NSPC and the bit-level control to the wires generated by IA encoder; (c) AND gate implementation of the bit-level switch.

3 Exploration from neural synaptic plasticity-inspired computing

In this section, we review our previous work, neural synaptic plasticity-inspired computing (NSPC), analyze the limitations of the previous NSPC, and propose practical solutions to these limitations.

Inspired by the biological essence of human neurons and SC, Xia et al. [27] proposed NSPC to mimic the connections among neurons with wires and to perform the low-cost and approximate multiplication. The high efficiency of biological neural systems comes from ubiquitous connections in the human brain. Neural synaptic plasticity can adjust the connection strength. NSPC associates the magnitude of weights in artificial neural networks with the connection strength in biological neural networks. The mechanism simulates the biological fact by the density of circuit wires, which is directly proportional to the strength of the connection strength and the magnitude of weights.

Now we introduce the main idea of NSPC. As shown in Figure 2(a), similar to SNGs, NSPC also generates a sequence where every element has the same weight '1'. The difference is that NSPC generates this sequence within only one clock cycle. For a Q-bit fixed-point IA a, 2^Q-1 activation wires are generated at once. In detail, for the j-th bit A_j, 2^j duplicate wires are produced. Therefore, the frequency, or the probability from a statistical perspective, of '1's in this sequence is $p_{\mathrm{IA}} = a/(2^Q-1)$. Then, these activation wires are randomly shuffled. The NSPC-based multiplier performs the multiplication by selecting these generated activation wires in parallel. For instance, suppose the weight is $w \in [0,1]$, then $f(w) \times (2^Q-1)$ activation wires are randomly selected, where $f(\cdot)$ is a normalization operation. Finally, these wires are added together and shifted, and then the binary result is obtained.

Although the NSPC-based multiplier can reduce the computing time of the traditional SC to only one clock cycle and save logical cost compared with fixed-point multipliers, it suffers from limited reconfigurability. That is, NSPC can only perform the multiplication in which one operand is constant (i.e., constant coefficient multiplier). Therefore, the lack of reconfigurability and the large chip area caused by the fully parallel implementation prevent the NSPC-based accelerator from applying to versatile neural networks.

To solve the generalization problem, firstly, the generation of activation wires can be viewed as a predefined function, which is performed by the IA encoder. Then, we can equip each activation wire generated from the IA encoder with a bit-level switch controlled by the control bit generated from the weight encoder, as depicted in Figure 2(b). In this way, the circuit can perform bit-level control over every activation wire according to the variable weight. If the control signal is '1', then the activation wire can pass. Otherwise, it will be gated. For example, suppose the activation wires are A_2, A_1, A_0 and the control bits are 1, 0, 1, then the wire A_1 will be gated, resulting in A_2, 0, A_0. Figure 2(c) shows that a two-input AND gate can serve as the 1-bit switch in digital circuits. One of the inputs is the activation wire bit corresponding to the IA, and another is the control bit corresponding to the weight. In the end, backward logic such as adders is able to convert the survived wires into the binary domain. Through this analysis, we can craft a new reconfigurable stochastic multiplier.

In this way, the activation wire density varies dynamically with the varying magnitude of weights.

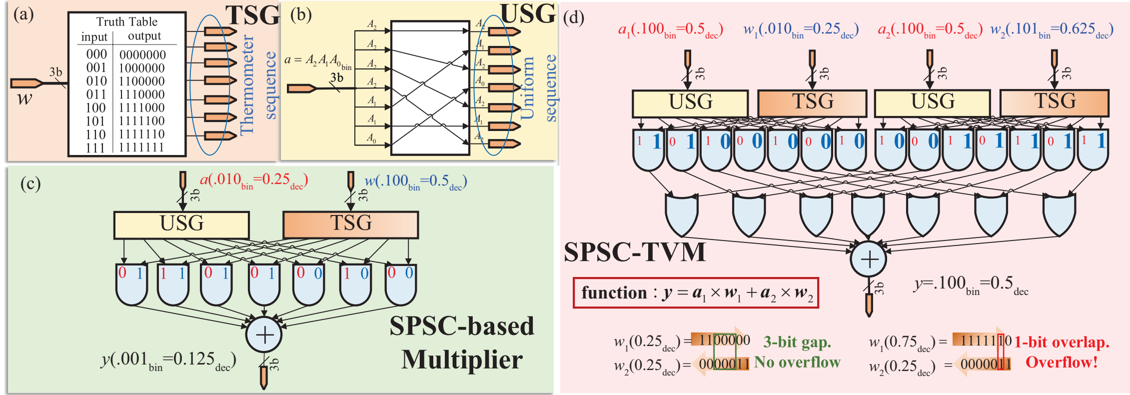


Figure 3 (Color online) 3-bit example for the proposed SPSC multiplier and its optimizations. (a) TSG converting the binary number to the thermometer sequence; (b) USG converting the binary number to the uniform sequence; (c) SPSC-based multiplier; (d) optimized SPSC-TVM and its constraint on weights.

Since the magnitude of weights is mapped to the synaptic strength [27], the above method also uses the wire density to simulate the neural synaptic plasticity.

4 Proposed high-accuracy and low-cost SPSC computing unit

In this section, a high-accuracy, low-cost SPSC-based multiplier with optimized encoding schemes is presented. Then, the hardware details and the mathematical proof as well as the error analysis of the SPSC-based multiplier are presented. After that, the BSO is proposed to reduce the logical overhead. Finally, an efficient weight pairing algorithm is proposed to reduce failure-pair cases as much as possible while satisfying the magnitude constraint of our SPSC-TVM.

4.1 Multiplier with spatial-parallel encoding schemes

Following the preceding analysis in Section 3, the remaining task is to find efficient and accurate encoding schemes for the IA and weight, and then design the corresponding backward logic.

Firstly, to design the weight encoder, as depicted in Figure 3(a), we adopt the thermometer coding from Zhang et al. [20], referred to as thermometer-sequence generator (TSG) in this paper, to convert the fixed-point weight $w \in [0, 1)$ to the thermometer sequence, where '1's appear at the head of the sequence. The TSG converts the Q-bit unsigned fractional weight w to $2^Q - 1$ bits within only one cycle. And the probability of '1's in the thermometer sequence is $(w \times 2^Q)/(2^Q - 1)$.

Secondly, for the IA encoder, inspired by the low-discrepancy sequence adopted and optimized in [21], we design a customized wire shuffle method, called uniform sequence generator (USG), with a deterministic shuffle pattern, as shown in Figure 3(b). Like NSPC, j-th bit of the Q-bit fixed-point input a is extended to 2^j wires, where $j \in [0, Q-1]$ is the bit index. In total, the IA a generates $2^Q - 1$ wires. And the frequency of '1's in the uniform sequence is $(a \times 2^Q)/(2^Q - 1)$. Specifically, these wires are shuffled according to the following rule.

USG rule. the j-th bit A_j first appears at position $2^{(Q-j-1)}$, and thereafter in every $2^{(Q-j)}$ bits.

An advantage of USG is that it does not require any circuit resources except wires because for arbitrary inputs with a given bitwidth, the wire connection pattern is deterministic.

Figure 3(c) illustrates the design of the SPSC-based multiplier. The TSG and USG convert the weight w and IA a to the thermometer sequence and uniform sequence, respectively. Here, TSG and the USG are interchangeable. Then, the i-th bit of the thermometer sequence and the i-th bit of the uniform sequence are fed into the AND gate. There are a total of $2^Q - 1$ AND gates. Finally, an adder tree severing as the backward logic with $2^Q - 1$ 1-bit inputs is used to get the binary result.

To prove the mathematical correctness of the proposed SPSC-based multiplier, we define the computing goal as

$$y = (a \times 2^Q) \times w, \tag{1}$$

where both $a \in [0,1)$ and $w \in [0,1)$ are unsigned fixed-point numbers. For the Q-bit a, we have

$$a = (A_{Q-1}A_{Q-2}\cdots A_0)_{\text{bin}} = \sum_{j=0}^{Q-1} 2^{j-Q} \times A_j. \tag{2}$$

Therefore, Eq. (1) can be rewritten as

$$y = \sum_{j=0}^{Q-1} 2^{j-Q} \times A_j \times (w \times 2^Q) = \sum_{j=0}^{Q-1} (2^j \times w) \times A_j \simeq \sum_{j=0}^{Q-1} \text{round}(2^j \times w) \times A_j. \tag{3}$$

The final term of (3) is an approximate term of (1), which is also the hardware result of the proposed SPSC-based multiplier. To prove this, we give the following property of USG.

USG property. the number of times A_j appears within the first $w \times 2^Q$ bits in the uniform sequence is round$(2^j \times w)$.

To prove this property, we define an auxiliary variable h as

$$h = \begin{cases} 1, & w \times 2^Q - 2^{Q-j-1} = 2^{Q-j} \times (w \times 2^j - 0.5) \geqslant 0, \\ 0, & \text{otherwise}. \end{cases} \tag{4}$$

According to the **USG rule**, if the j-th bit A_j appears at least once within the first $w \times 2^Q$ bits, then we have $h = 1$. Otherwise, A_j is not covered by the first $w \times 2^Q$ bits and thus $h = 0$.

For the remaining $w \times 2^Q - 2^{Q-j-1}$ bits, the number of times A_j appears is computed by

$$\text{floor}((w \times 2^Q - 2^{Q-j-1})/2^{Q-j}) = \text{floor}(w \times 2^j - 0.5). \tag{5}$$

Therefore, the total number of times A_j appears within the first $w \times 2^Q$ bits is (4) + (5), which equals round$(2^j \times w)$. The **USG property** has been proved. Moreover, based on the established **USG property**, the hardware result of the proposed SPSC-based multiplier is obviously equal to the final term of (3).

The computing error of the SPSC-based multiplier results from the round function in (3), which can be formulated as the sum of independent round-off errors. Hence, the theoretical upper bound of the absolute computing error is $\sum_{j=0}^{Q-1} 1/2 = Q/2$.

4.2 SPSC-TVM with BSO

Although the aforementioned SPSC-based multiplier reduces the computing time of traditional SC to only one clock cycle, it still suffers from a relatively high logical cost. The number of gates of the Q-bit SPSC-based multiplier is given by

$$G_{\text{SPSC-Mult}}(Q) = (2^Q - 1) + ((2^Q - 1 - Q) \times 5) + (G_{\text{TSG}}(Q)). \tag{6}$$

In (6), the first term is the number of AND gates functioning as bit-level switches, the second term corresponds to the adder tree with $2^Q - 1$ 1-bit inputs, and the third term is the cost of the TSG which can be ignored as demonstrated in Subsection 6.2. Considering (6), the second term (i.e., the adder tree) dominates the overall logical cost, especially when Q is increasing.

Actually, the logical function of an adder with two 1-bit inputs can be realized by a simple OR gate if the carry bit is ignored (i.e., without the overflow effect). With this consideration, we propose the BSO as shown in Figure 3(d). To calculate $a_1 \times w_1 + a_2 \times w_2$, OR gates are used to replace the first stage of the adder tree rather than directly deploying costly adders to sum a total of $2 \times (2^Q - 1)$ bits. Moreover, we leverage a spatial property of the thermometer sequence to overcome the overflow effect of OR gates. All of the '1's appear at the head of the thermometer sequence. Therefore, two thermometer sequences can be arranged in a tail-to-tail manner. In this way, there are no overflow effects as long as the number of '1's in the two thermometer sequence is less than 2^Q. We call this special constraint the magnitude constraint. The definition of the overflow is $(w_1)_{\text{dec}} + (w_2)_{\text{dec}} > (2^Q - 1)/2^Q$, where $(w_1)_{\text{dec}}$ and $(w_2)_{\text{dec}}$ are the decimal value of two unsigned fixed-point weights, Q is the bitwidth of the weight. The proposed arithmetic circuit in Figure 3(d) is called SPSC-TVM.

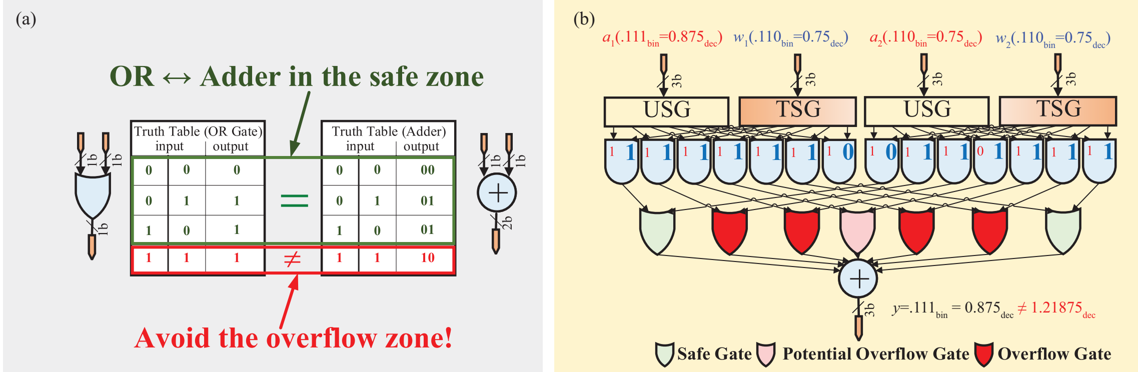


Figure 4 (Color online) Visualization of the overflow effect. (a) OR gate level overflow; (b) SPSC-TVM level overflow. Safe gates will never overflow, given the current two weights. The potential overflow gate does not overflow given the current two activations, but may overflow in other combinations of activations. The overflow gate already overflows in the current case, and each OR gate leads to the loss of 1'b1.

To better understand the overflow effect and its impact on the computing accuracy of SPSC-TVM, we give a simple case study illustrated in Figure 4. The essence of the overflow effect is that using a single OR gate cannot obtain the same results as a two 1-bit inputs adder when both inputs are 1'b1, leading to a loss of 1'b1, as shown in Figure 4(a). In Figure 4(b), we show how the overflow of OR gates causes the computing accuracy loss of SPSC-TVM. Here, given $w_1 = 0.75$ and $w_2 = 0.75$, the five middle OR gates fall into the potential overflow zone. Given the combination of $a_1 = 0.875, a_2 = 0.75$, four out of the five OR gates overflow. Finally, the final computation result deviates from the exact value by 28.2%. Furthermore, according to our numerical experiment, with the bitwidth increasing from 4 bits to 6 bits, the MAE (%) of SPSC-TVM without the overflow is 8.6%, 4.4%, and 2.2%, respectively. However, the MAE (%) with the overflow is 11.1%, 9.4%, and 8.8%, respectively. Moreover, the computing accuracy loss will accumulate and propagate along the deep computing path of CNNs, thus leading to intolerable performance loss. Please note that although we can use USG to encode both activations and weights from a statistical perspective, the combination of USG and TSG has better computing accuracy (MAE (%) is 28.1% versus 4.4% given 5-bit precision) and enables the BSO, which significantly reduces the cost of the adder tree.

The proposed SPSC-based multiplier has some similarities with the design in [21] in terms of encoding patterns. However, Ref. [21] does not explicitly give the implementation of the fully parallel version of their design. Moreover, the SPSC-based multiplier is only a basic element in our design SPSC-TVM, which mainly reduces the cost of the expensive high fan-in adder tree via the proposed BSO.

4.3 Magnitude-aware weights pairing

In this subsection, we propose the MAWP algorithm to make weights sent to SPSC-TVMs satisfy the magnitude constraint due to BSO. At the same time, the weight pairing scheme needs to reduce the inserted zeros as much as possible.

In order to minimize the computing error, we adopt the weight scaling technique from Kim et at. [17]. We scale up the weights in one layer (or one kernel) before the multiplication and then scale back down the result after the accumulation. Suppose that the data range of the weights is limited to $[-1/s, 1/s]$ where $s > 0$. Then, all the weights are scaled up by s times. For example, if the weights are $[-0.4, 0.1, 0.2]$, and then we have $s = 2.5$ and the scaled weights are $[-1, 0.25, 0.5]$. Moreover, to enable the process of signed weights, we use the sign-magnitude format to represent the two's complement numbers. In detail, one bit is for the sign (given by the controller), and the other bits are for the absolute value of weights. For instance, 0.875 and -0.125 are represented by 4'b0.111 and 4'b1.001, respectively. The positive weights and negative weights are paired with the proposed MAWP, respectively. After that, positive pairs are sent to SPSC-TVMs serially, and then the negative pairs.

SPA adopts a tile-based dataflow as illustrated in the following Subsection 5.3, decreasing the search scope to a weight workload containing T_C weights rather than as large as all weights in one layer, increasing the difficulty to find valid weight pairs. To mitigate this problem, we introduce the MAWP algorithm shown in Algorithm 1 to obtain the optimal pairing mode under the scope of a weight workload. Here, we only consider the positive part of weights because the negative part is the same. The Q-bit fractional

Algorithm 1 Algorithm of the proposed MAWP

Input: The bitwidth of weights: $Q \in \mathbb{Z}^+$; a set of positive integer weights: $W[i] \in \mathbb{Z}^+, 0 \leqslant i \leqslant T_C - 1$;
Output: A (x by 2) array W_P storing pairs of weights; // x is a variable depending on the value of input weights.
1: Initialize the upper bound of the sum of a weight pair: $U \Leftarrow 2^Q - 1$;
2: Sort the weight list from largest to smallest: $W_S \Leftarrow \mathrm{Sort}(W)$;
3: $x \Leftarrow 0$; flag $\Leftarrow$ False; // Set a flag to indicate the state of pairing.
4: **while** $T_C > 0$ **do**
5: **for** $i = 1$; $i < T_C$; $i++$ **do**
6: **if** $((W_S[0] + W_S[i]) \leqslant U) \bigcup (T_C > 1)$ **then**
7: $W_P[x] \Leftarrow [W_S[0], W_S[i]]$;
8: Remove $W_S[0], W_S[i]$ from W_S, and move remaining elements to the left to squeeze two bubbles;
9: $T_C \Leftarrow T_C - 2$; flag $\Leftarrow$ True; **Break**; // Successfully find two weights and then break the "for" loop.
10: **end if**
11: **end for**
12: **if** flag == False **then**
13: $W_P[x] \Leftarrow [W_S[0], 0]$; // Pare the currently maximum weight with a zero.
14: Remove $W_S[0]$ from W_S, and move remaining elements to the left to squeeze one bubble;
15: $T_C \Leftarrow T_C - 1$;
16: **end if**
17: $x \Leftarrow x + 1$; flag $\Leftarrow$ False; // Reset the flag for the next pairing operation.
18: **end while**

weight is viewed as its integer form (i.e., $2^Q \times$ weight). The main principle is to pair the maximum weight of the whole candidate set with the maximum weight in the remaining weights that meets the magnitude constraint. Otherwise, we pair it with a zero. Then, we remove the paired weights/weight from the candidate set. Finally, this process is repeated until all weights are paired.

5 Hybrid zero-skipping scheme

This section focuses on accelerating both CLs and FCLs by exploring the data sparsity, including both the compile-time sparsity (zero-valued weights) and the run-time sparsity (zero-valued IAs). First, we analyze the reasons for using different types of sparsity in different types of layers. Second, different data compression formats used respectively in the CL and FCL for weights and IAs are introduced. Third, a customized dataflow with high reconfigurability and data reuse is demonstrated.

5.1 Exploring sparsity: weights and IAs

The basic operation in the typical dense CL is formulated by

$$\mathrm{OA}_{(e,f,m)} = \sigma\left(\sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{c=0}^{C-1} \mathrm{IA}_{(h+i,w+j,c)} \times \mathrm{Weight}_{(i,j,c,m)}\right), \tag{7}$$

where σ is the activation function. For simplicity, the bias is ignored, and the padding size and stride are assumed to be zero and one, respectively. In this paper, we will obey the following indexing format: a weight index of (r, s, c, m) denotes (row, column, channel, kernel), an IA index of (h, w, c) corresponds to (row, column, channel), and an output activation (OA) index of (e, f, m) denotes (row, column, kernel).

In the FCL, the calculation of an OA is given by

$$\mathrm{OA}_{(n)} = \sigma\left(\sum_{k=0}^{K-1} \mathrm{IA}_{(k)} \times \mathrm{Weight}_{(k,n)}\right), \tag{8}$$

where K is the number of input neurons, n is the index of the output neuron.

Generally, although using sparsity of two multiplicative operands may get a larger speedup, it also requires more complex mechanisms to find valid weight-IA pairs as in SNAP [25] or more expensive arithmetical circuits to compute the address of every intermediate psum as in SCNN [24]. Therefore, there is a performance-area trade-off we need to consider carefully for every crafted sparse accelerator. We propose a unique zero-skipping technique, HZSS, to accelerate both CLs and FCLs in a hybrid strategy. Different from most current sparse accelerators [22–25, 30], SPA adopts different zero-skipping strategies to conquer different types of layers. Thus we refer to our method as **hybrid**. To be specific, we skip the computation of zero-valued weights in CLs and zero-valued IAs in FCLs but do not exploit the sparsity

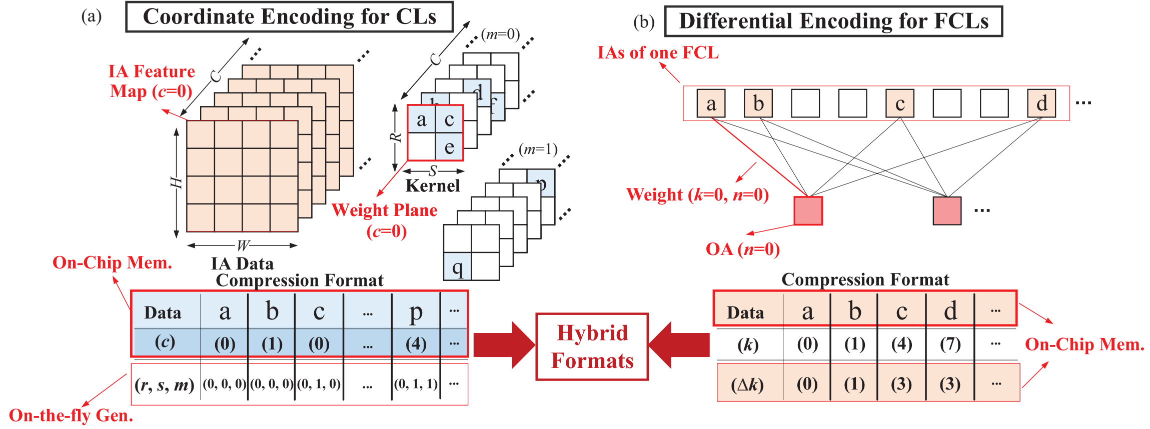


Figure 5 (Color online) Data compression in SPA. (a) The diagram of IA data and a compressed weight kernel, in the CL; (b) sparse input neurons in the FCL and its compressed format.

of both the weights and IAs in the same layer. The reasons are two folds. For CLs, we only need to find two valid weights rather than making four operands zero-valued, giving more flexibility to the process of paring weights. Moreover, the proposed SCPP dataflow adopts the channel-first fashion, making two psums generated by SPSC-TVMs additive. For FCLs, it is inefficient for SPA to compute the FCL with the same hardware, where one weight is broadcast to many IAs, because all weights of FCLs cannot be shared except the batch size is larger than one, which is much less common in accelerators towards the inference of CNNs. Therefore, we assign the processing of sparse FCLs to an independent and tiny engine, which will be introduced in Subsection 6.4.

5.2 Hybrid data compression format

In this subsection, we show and explain different data compression formats of weights in CLs and IAs in FCLs, respectively. Regarding data compression, our goal is maintaining a smooth processing flow while minimizing the size of required on-chip memories, part of which are wasted by storing indices of data. To enable different zero-skipping strategies in different types of layers, we use the coordinate encoding and differential encoding for weights in CLs and IAs in FCLs, respectively.

In CLs, weights and IAs are stored in the compressed form and dense form, respectively. In coordinate encoding, four indices are required to determine the absolute position of the weight, dispatch it to associated computing elements and calculate the addresses of results. As shown in Figure 5(a), the weight storage is composed of a data array and a channel index array, which are generated during compile time. The other three indices, including the row index array, column index array and kernel index array are generated on-the-fly by the on-chip controller. This on-chip storage configuration has two benefits. First, the memory cost on indices is significantly reduced. Second, since fetching weights along the channel dimension is irregular (recall MAWP, which re-orders the weights) and SPA adopts a serial channel processing flow (i.e., SCPP, which will be introduced in the following section), the channel index and data of each weight enable efficient out-of-order processing.

In FCLs, only IAs are compressed and stored in on-chip memories. SPA does not contain any on-chip memories for weights of FCLs because here weights can not be reused, and SPA employs a serial manner to process FCLs. Directly using absolute indices of input neurons significantly increases the on-chip buffer size when there are a larger number of input neurons. Therefore, as depicted in Figure 5(b), SPA uses differential encoding to compress the IA data in FCLs. In practice, the differential index is represented by five bits. If the distance between two non-zero input neurons exceeds 31, a zero will be inserted.

5.3 SCPP dataflow for CLs

In this subsection, we present the dataflow of CLs. Chen et al. [22] has demonstrated that the choice of dataflow plays a great effect on the efficiency of an architecture. In essence, the dataflow of CLs can be viewed as a specific permutation of the nested loop with six loop variables: m, c, h, w, r, s. While crafting an optimal dataflow is beyond the scope of this work, we propose a customized dataflow called SCPP, which is compatible with the fundamental SPSC-TVM and enables data reuse. Figure 6 shows

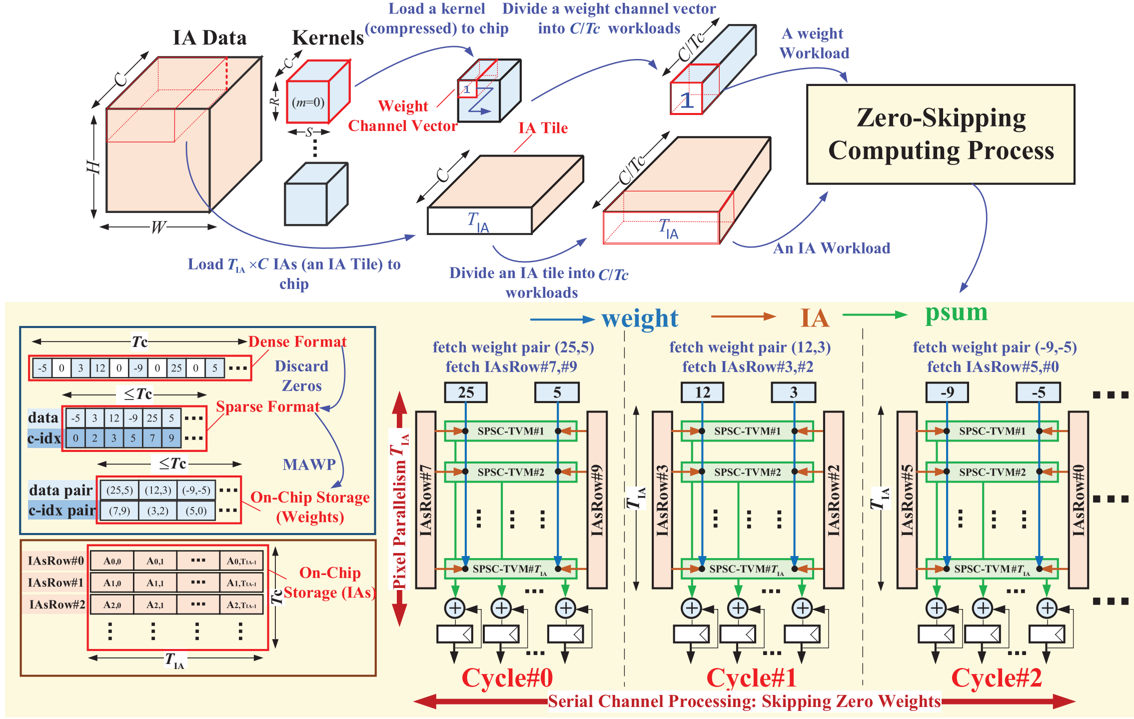


Figure 6 (Color online) SCPP dataflow presented in the picture work flow style to give a simple illustration.

the picture version of the SCPP dataflow without detailed mathematical functions to give a simple and visual illustration.

SPA processes CLs in a layer-by-layer style. In a layer, we choose the kernel dimension m as the outermost loop. To this end, SPA has a weight buffer that can accommodate a whole kernel. Then, the weight buffer needs to be refilled by the next new kernel once an OA feature map is obtained. This configuration has two good properties: (a) reducing frequent accesses to the energy-expensive off-chip dynamic random access memory (DRAM) and (b) exploiting weight sparsity to reduce the size of the on-chip weight buffer. For IA data, it is usually impossible to store all of them in the on-chip memory as IA data are not compressed here and the size of IA data is usually much larger than a weight kernel. Therefore, we break IA data with size $H \times W \times C$ into $H \times W/T_{\rm IA}$ small IA tiles with size $(T_{\rm IA} \times C)$ along the pixel dimension of the IA plane. One IA tile is loaded onto the IA buffer at a time.

Now, we introduce the detailed processing flow after a kernel and an IA tile are loaded onto the chip. The computational form of SPSC-TVM requires two psums (i.e., $a_1 \times w_1$ and $a_2 \times w_2$) to be additive. To satisfy this characteristic, we find that according to (7), $R \times S \times C$ multiplications' outputs contributing to an OA are additive. Moreover, SPA needs to be highly efficient in all possible kernel sizes, such as 3×3 and 1×1. We therefore break down the kernel with size $R \times S \times C$ into $R \times S$ weight channel vectors with size $1 \times 1 \times C$, which are processed serially. Moreover, each weight vector is further broken into C/T_C sub-vectors defined as weight workloads, which are serially fed to the computing resources. Here, the design parameter T_C is the tiling size along the channel dimension. For the IA data, each IA tile is further split into C/T_C IA workloads with the size $T_{\rm IA} \times T_C$.

The processing of one weight workload and one IA workload is the most distinctive feature that differs the proposed SCPP dataflow from others. All zero weights in the current workload are discarded. The non-zero weights in the weight workload are paired via the proposed MAWP and stored in the data-index style. All the valid weight pairs are fetched and sent to the computing resources serially. Therefore, the multiply-accumulates (MACs) corresponding to these zero weights are efficiently skipped. However, this serial processing may lead to low performance. Therefore, we improve the parallelism in the pixel dimension of IA planes. IAs in the IA workload are stored in the dense format, and $T_{\rm IA}$ IAs reside in a row of the IA buffer, indicating the degree of the parallelism is $T_{\rm IA}$. Consecutive rows store IA sub-planes with consecutive channel numbers to enable simple indexing for zero-skipping processing. The two channel indices of the weight pair indicating the channel numbers together with a bias given by the on-chip controller are used to locate the two rows of IAs. Then the data fields of the weight pair and two

```
1 Memory      [C/T_C][T_IA][T_C]            IA_BUF;   // buffer storing an IA tile
2 Memory      [R][S][C/T_C][T_C/2][2]        W_BUF;   // buffer storing a whole kernel
3 Memory      [T_IA][E][F]                  Psum_BUF; // buffer storing intermediate psums
4 Memory      [E][F]                        OA_BUF; //  buffer storing OAs applied to the activation function
5 Memory      [T_IA]                        Acc_BUF; //  accumulator buffer
6 Parameter Pad_Up, Pad_Left, Strd_H, Strd_W; // padding sizes of the upper and left edge of the IA plane,  strides along the row and column dimension
7  for m = 0 to M-1 { // loop1  over kernels
8 |   load the m-th kernel to W_BUF;
9 |   for i = 0 to H × W/T_IA -1 { // loop2  over all  IA  tiles
10|   |   load the i-th IA tile to IA_BUF;
11|   |   get h_Coord[0:T_IA -1] and w_Coord[0:T_IA -1]; // get the coordinates of the T_IA  channel vectors in this IA tile
12|   |   for r = 0 to R-1 {  // loop4  over the row dimension of the kernel
13|   |   |   for s = 0 to S-1 { // loop5  over the column dimension of the kernel
14|   |   |   |   e_Coord[:] = OutCoord_Gen(h_Coord[:], Pad_Up, r, Strd_H); f_Coord[:] = OutCoord_Gen(w_Coord[:], Pad_Left, s, Strd_W); // get the coordinates of psums
15|   |   |   |   for c′ = 0 to C/T_C -1 { // loop6  over a channel vector containing C/T_C  workloads
16|   |   |   |   |   for t_C = 0 to T_C/2-1 { // loop7  over weight pairs in a workload
17|   |   |   |   |   |   parallel_for t_IA = 0 to T_IA -1 { // T_IA  SPSC-TVMs  operating synchronously
18|   |   |   |   |   |   |   Acc_BUF[t_IA] += SPSC-TVM(IA_BUF[c′][t_IA][2×t_c], W_BUF[r][s][c′][t_c][0], IA_BUF[c′][t_IA][2×t_c+1], W_BUF[r][s][c′][t_c][1]); } } }
19|   |   |   |   parallel_for t_IA = 0 to T_IA -1 {Psum_BUF[t_IA][e_Coord[t_IA]][f_Coord[t_IA]] += Acc_BUF[t_IA]; clear Acc_BUF; } } } }
20|   for (e = 0 to E-1; f = 0 to F-1) {OA_BUF[e][f] = Point_Wise_Func(Sum(Psum_BUF[:][e][f])); clear Psum_BUF;Pooling; } } // post processing
```

Figure 7 (Color online) SCPP dataflow in the nested loop style gives a clear and detailed illustration. Here, the weight buffer (W_BUF) is assumed to be a dense format for illustration, which is larger than the practical implementation. The Point_Wise_Func performs $\sigma(a \times x + b)$, which is able to add biases, compute the batch normalization, complete the scaling operation, and calculate the activation function.

corresponding IA rows are sent to the computing resources built upon SPSC-TVMs. Here, one weight is broadcast to $T_{\rm IA}$ IAs. Then, the next weight pair is fetched and processed in the same way. To sum up, the proposed SCPP dataflow serially processes data along the channel dimension and improves the parallelism in the pixel dimension.

Besides the computing of data, SPA also needs extra logic to calculate coordinates of the run-time generated psums to take benefits of sparsity. The pixel dimension coordinates of the results of $T_{\rm IA}$ parallel SPSC-TVMs are calculated by a coordinate generator according to the following equations:

$$e = \left\lfloor \frac{h + {\rm Pad}_{\rm Up} - r}{{\rm Strd}_H} \right\rfloor, \quad f = \left\lfloor \frac{w + {\rm Pad}_{\rm Left} - s}{{\rm Strd}_W} \right\rfloor, \tag{9}$$

where ${\rm Pad}_{\rm Up}$ and ${\rm Pad}_{\rm Left}$ are the padding sizes of the upper and left side of the IA plane, ${\rm Strd}_H$ and ${\rm Strd}_W$ are the stride sizes along the height and width dimension.

When all C/T_C weight workloads and IA workloads are processed, these output coordinates are used to fetch the corresponding psums from the Psum buffer. And these psums are added together with the intermediate data stored in the accumulators, then the final results are written into the Psum buffer and accumulators are reset to zero. Then, the next IA tile is loaded to the chip. Once finishing the current kernel (i.e., all IA tiles are processed), the post processing, consisting of calculating the activation function, the batch normalization (optional), adding biases, re-scaling, and the pooling operation (optional) starts to work. Now, the processing of the current kernel is done.

For more details, please refer to Figure 7, which gives a more concrete and precise representation of the proposed SCPP dataflow in a nested loop fashion.

6 SPA architecture

In this section, we provide the overall architecture of SPA and the detailed function modules. Then the three-level time interleave technique is proposed to increase the processing speed.

6.1 Overall architecture

Figure 8 depicts the overview of SPA architecture. The main controller completes the control of instruction decoding, routing of data, interaction with the data loader & dispatcher, sub-controllers, and the overall pipeline scheduling. The data loader transfers off-chip data, including weights and IAs, to the on-chip IA buffers and the weight buffers. The data dispatcher provides the computing resources with a certain volume of IAs and weights according to the dataflow. Besides, the CL engine, which processes CLs, is composed of IA buffers, weight buffers, the multi-banked psum buffer, the computing engine and the post processing unit (the green part). Furthermore, SPA has an independent FCL engine specialized for the processing of FCLs.

In CLs, weights and IAs are represented by 6 bits (1 bit for sign and 5 bit for magnitude) and 15 bits, respectively. We divide the 15-bit IA into three 5-bit parts and compute them separately with three

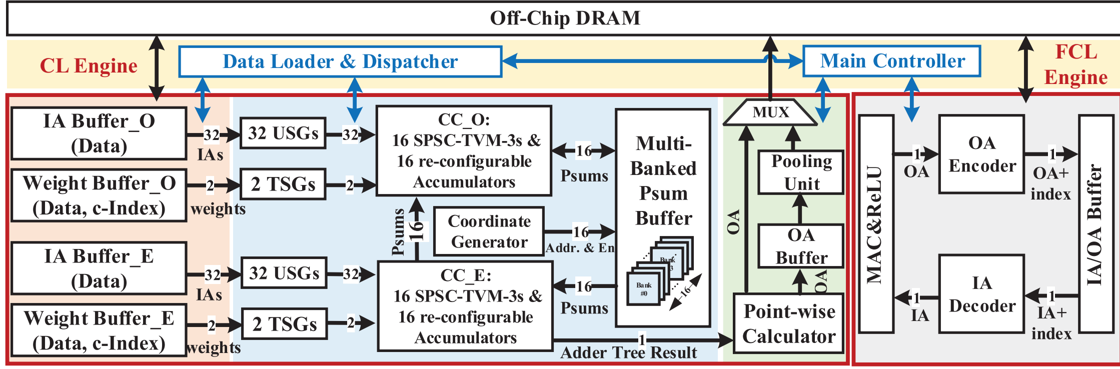


Figure 8 (Color online) Overall architecture of SPA.

SPSC-TVMs, then the results are shifted and summed to obtain the final output. We call this design SPSC-TVM-3. We use TSGs to convert weights and USGs to convert IAs. Because all weights are known during compile time after training, we can pair weights to satisfy the magnitude constraint.

6.2 SPSC-based multi-function computing engine

The main goal of SPA is to exploit the low-cost properties of the proposed SPSC and provide smooth processing of reconfigurable and sparse CNNs. Therefore, the design of an efficient and flexible computing engine is important.

In this design, we set $T_{\rm IA}$ to 16 and T_C to 32, and a computing core (CC) has 16 SPSC-TVM-3s. Moreover, 16 accumulators (Acc_BUF in line 18, Figure 7) are provided for 16 SPSC-TVM-3s to perform the accumulation of psums. To amortize the overhead of TSGs over 16 SPSC-TVM-3s, only one weight pair is sent to CC at one time, and then encoded by two TSGs and broadcast to 16 SPSC-TVM-3s. In this way, each SPSC-TVM-3 only cost 1/16 logic resources of two TSGs. Because the USG cost zero logic resources, 32 IAs are freely encoded by 32 USGs and fed to CC in parallel.

To increase the parallelism, we implement two CCs named CC_E and CC_O, respectively. Moreover, both CC_E and CC_O can perform some specialized operations. In Subsection 5.3, we have broken the calculation of the CL into many workloads, each of which has T_C weights to be processed. For a weight channel vector and its corresponding IAs, C/T_C workloads are split into the even part and the odd part, which are assigned to CC_E and CC_O, respectively. And each CC obeys the proposed SCPP dataflow. When both the CC_E and CC_O complete their respective workloads, 16 psums stored in the accumulators of CC_E need to be added to 16 psums stored in the accumulators of CC_O. This operation is performed by the reconfigurable accumulators of CC_O. In addition, CC_O also completes the read-and-write operation of the multi-banked Psum buffer.

The coordinate generator consisting of four pipelined fixed-point dividers, some counters, and comparators generates addresses for psums and write-enable signals for the multi-banked Psum buffer, according to (9).

6.3 Post processing unit

The post processing unit composed of the point-wise calculator, the OA buffer, and the pooling unit serves the following functions: (a) summing psums stored in the multi-banked Psum buffer to get OAs not applied to ReLU; (b) calculating point-wise functions including adding biases, activation functions, re-scaling, and batch normalization; (c) performing the pooling operation if required. For (a), the 16 accumulators of CC_E can be reconfigured as a 16-input adder tree, receiving and summing 16 psums from the multi-banked Psum buffer. For (b), the point-wise calculator with a multiplier, an adder, and a comparator is deployed. For (c), if the pooling layer is defined after the CL, OAs are first written to the OA buffer, then processed by the pooling unit. Otherwise, OAs are sent to the off-chip DRAM for the following layers.

6.4 Support for FCLs

In SPA, we assign the processing of FCLs to an independent module named the FCL engine. There are three reasons for this special design: (a) directly using CCs to perform the matrix multiplication is inefficient because weights can not be shared in FCLs; (b) the independent FCL engine enables pipelined processing of CLs and FCLs, improving the speed of inference; (c) the arithmetic circuit of the FCL engine is implemented by the traditional fixed-point 16-bit multiplier and 32-bit accumulator, and thus mitigating the precision degradation caused by the computing error of the proposed SPSC.

To get a better inference performance and energy efficiency, the FCL engine is designed to satisfy the following rules: (a) low area cost and power consumption compared with the CL engine; (b) high processing speed so as not to decrease overall performance. To this end, the hardware of the FCL engine obeys the following design characteristics. First, as shown in Figure 8, there is only one MAC unit in the computing circuit to reduce computing logic. Second, activation sparsity is explored efficiently to improve speed via the IA decoder. Third, indices of OAs are compressed by differential encoding before being stored in the on-chip IA/OA buffer to reduce memory size by OA encoder.

The dataflow of processing FCLs is similar to the output stationary dataflow. We compute the result of each output neuron one by one. Within the workflow of a single output neuron, a non-zero IA is first fetched from the on-chip IA/OA buffer and the IA decoder outputs the absolute index of the current IA, which is used to find the corresponding weight. Then, the MAC operation is preformed. When all non-zero IAs in the current layer are processed, the ReLU activation clamps the negative results to zero. Finally, only non-zero OAs are encoded by the OA encoder and stored in the on-chip IA/OA buffer to save the memory cost.

6.5 Three-level time interleaving technique

To further increase the processing speed, we proposed the three-level time interleaving technique. SPA explores time interleaving opportunities from the micro scope, the medium scope, to the macro scope to make the time overhead of some tasks hidden into others. In detail, SPA uses the simple handshake protocol to schedule tasks so that two or more tasks can execute at the same time while maintaining semantic correctness.

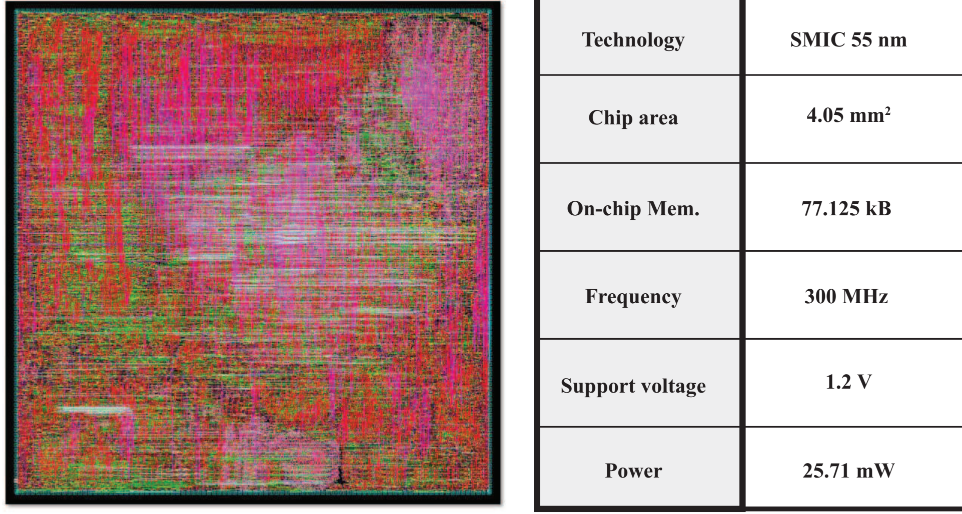
For the micro level, the coordinate generator and CCs can work simultaneously. In most cases, the generation of addresses of psums requires fewer clock cycles than the computing and accumulating of psums, so that we can cover the time of the coordinate generator with that of CCs. For the medium level, the data loader&dispatcher and on-chip IA buffers and Weight buffers can work in parallel with the post processing unit. That is, in most cases, the time overhead of executing post processing steps can be hidden in that of the data preparation of the processing of the next kernel. For the macro level, the CL engine and the FCL engine can work in parallel. Note that compared with CLs, FCLs involve less computation in CNNs. For instance, FCLs account for less than 10% of the total operations in most popular CNNs such as GoogleNet and ResNet [31]. Therefore, the time of processing FCLs can be hidden into the convolution of the next input image.

Here, we clarify the benefits that HZSS can bring to the hardware architecture. The proposed HZSS significantly simplifies the process of finding valid weight-IA pairs. Since we only exploit weight/IA sparsity in CLs/FCLs, only one type of indices is needed to index the corresponding data. For instance, in the processing of CLs, we just use a 5-bit weight index to find its associated 16 IAs, which is highly efficient. Moreover, HZSS enables us to handle and optimize the computation of CLs and FCLs separately. For instance, in the CLs engine, customized computing engines are implemented to amortize the overhead of TSGs. In the FCL engine, only one traditional fixed-point MAC unit is employed to save area and energy while maintaining enough processing speed by skipping IAs. Furthermore, the CL engine and FCL engine can work in a pipeline style to improve the overall throughput.

7 Experiments

7.1 Experimental setup

To measure the area, power, and possible maximum clock frequency, SPA was implemented in Verilog HDL and the output results were verified with the assistance of a Python-based simulator. Then, SPA



Technology	SMIC 55 nm
Chip area	4.05 mm^2
On-chip Mem.	77.125 kB
Frequency	300 MHz
Support voltage	1.2 V
Power	25.71 mW

Figure 9 (Color online) Place and route result of SPA (excluding latch-based memories).

was synthesized by the Synopsys design compiler based on SMIC 55 nm low leakage process HD RVT standard cell library under 25°C. Figure 9 shows the place and route result from Synopsys IC compiler. The on-chip buffer sizes of IA buffer, Psum buffer, OA buffer, Weight buffer in the CL engine and IA/OA buffer in the FCL engine are 7.5, 48, 1.875, 11.5, and 8.25 KB, respectively. To reduce the power consumed by memories, all the on-chip buffers are implemented by customized latch-based standard cell memorys (SCMs). Although SCMs have larger area cost than static random access memorys (SRAMs), they provide lower power consumption, resulting in the possible better power efficiency [32]. The power, area, and latency were reported by Synopsys design compiler. Synthesis results show that SPA can work at a maximum clock frequency of 300 MHz with an area of 4.08 mm^2. The design consumes 25.71 mW under a supply voltage of 1.2 V. The proposed architecture can achieve a maximum energy efficiency of 2477.6 Gops/W.

7.2 Hardware performance and accuracy analysis of the proposed SPSC-TVM

We compare the proposed SPSC-TVM with its fixed-point counterpart, FIXED-TVM (i.e., the fixed-point implementation of $a_1 \times w_1 + a_2 \times w_2$), and the traditional serial SC-based implementation of $a_1 \times w_1 + a_2 \times w_2$ (SC-TVM). SC-TVM is implemented by two AND gates, an MUX-based adder, and a counter-based probability estimator. Five LFSRs and comparators are used to generate stochastic bitstreams for two weights, two IAs, and a selection signal. We present hardware metrics including the area, power, energy, and accuracy measured by MAE (%), as shown in Figure 10. To exhibit the advantage of sharing weights, which in turn amortizes the TSG overhead over many SPSC-TVMs, we show two cases: SPSC-TVMs with/without shared TSGs. For a fair comparison, we also provide the results when SC-TVM with shared forward converters (i.e., LFSR&Comparator).

The results indicate that the proposed SPSC-TVM with shared TSGs has both less area and power consumption compared to FIXED-TVMs when the bitwidth is less than or equal to 5 bits. Specifically, SPSC-TVMs require 18.91% less area and consume 21.67% less power than FIXED-TVMs when the bitwidth is 5 bits. However, the area of SPSC-TVM without shared TSGs exceeds that of FIXED-TVM at 5-bit data width. The energy is calculated by the product of the power and the critical path latency. Note that the critical path latency of SPSC-TVMs is irrelevant to sharing TSGs as the same TSGs logic exists at critical paths of all SPSC-TVMs. For the computing accuracy, it is clearly shown that the MAE (%) drops with an exponential decay as the bitwidth increases. The MAE is 4.35% at 5-bit data width, which is the case of real implementation of SPA.

SC-TVM consumes the lowest area because of its serial computing essence when the bitwidth is greater than 4 bits. And it can be observed that the area of SC-TVM grows more slowly than FIXED-TVM and SPSC-TVM. However, the power consumption is much higher than the proposed SPSC-TVM because the LFSRs, which contain some D flip-flops, consume a significant amount of power. Although SC-TVM and the other two designs are on the same order of magnitude in terms of energy, the actual energy

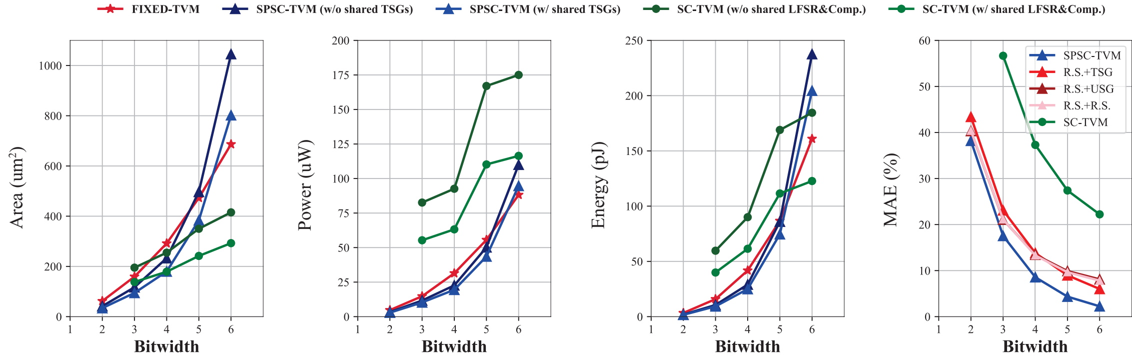
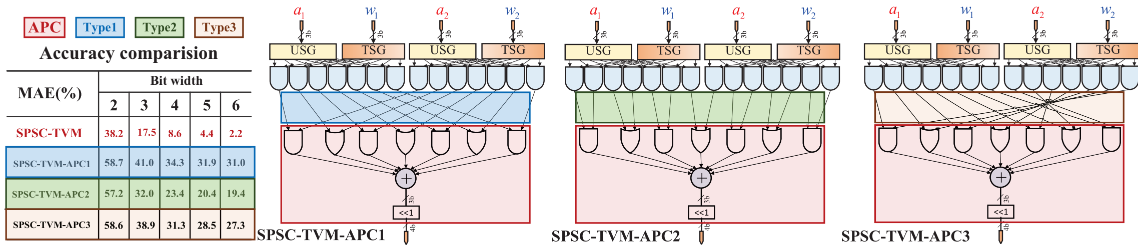


Figure 10 (Color online) Hardware and accuracy results of SPSC-TVM and FIXED-TVM under increasing bitwidths. The MAE (%) is defined as $(\sum_{l=1}^{L}|y_l-\hat{y}_l|/\sum_{l=1}^{L}y_l)\times 100$ and obtained from 100000 random tests for each bitwidth. The data of area and power of SPSC-TVMs with shared TSGs only include 1/24 these data of one TSG because one TSG is shared by 24 SPSC-TVMs in our practical design. For a fair comparison, we also give the results of SC-TVM with shared LFSRs, which only includes 1/24 area and power of the two LFSRs&Comp. to encode two weights. And the area and power of the other three LFSRs&Comp. to encode two IAs and the selection signal keep unchanged. Note that the actual computing energy of SC-TVM should be 2^Q-1 times higher because of the serial bitstream with a length of 2^Q-1 bits.



Accuracy comparision

MAE(%)	Bit width				
	2	3	4	5	6
SPSC-TVM	38.2	17.5	8.6	4.4	2.2
SPSC-TVM-APC1	58.7	41.0	34.3	31.9	31.0
SPSC-TVM-APC2	57.2	32.0	23.4	20.4	19.4
SPSC-TVM-APC3	58.6	38.9	31.3	28.5	27.3

Figure 11 (Color online) Three alternatives based on APC for the proposed OR-first-stage adder tree: SPSC-TVM-APC1, SPSC-TVM-APC2, and SPSC-TVM-APC3.

consumption of SC-TVM is much higher because it requires an exponentially increasing number of clock cycles to perform a complete operation. For example, when the bitwidth is 5 bits, SC-TVM (w/ shared LFSR&Comp.) consumes 46× energy compared to SPSC-TVM (w/ shared TSGs). For the computing accuracy, SC-TVM behaves worse than SPSC-TVM in the metric of MAE (%) because the MUX-based adder introduces large computing errors compared to the proposed BSO-optimized adder tree.

To better understand the improvement of the computing accuracy caused by the deterministic encoding schemes (i.e., TSG and USG), Figure 10 also provides the MAE (%) with half (R.S.+TSG, R.S.+USG) and full (R.S.+R.S.) random shuffle schemes. We can observe that the combined effect of TSG+USG leads to the highest computing accuracy. Moreover, USG implemented by wires only, is more hardware-friendly than a random shuffle method, which typically requires LFSRs and multiplexers.

Finally, we conducted another numerical experiment to test the advantage of the high accuracy resulting from the proposed BSO, where OR gates serve as the first stage of the adder tree. Kim et al. [33] proposed the approximate parallel counter (APC) to sum 1-bit signals in parallel. As shown in Figure 11, in APC, AND, and OR gates appear alternatively to serve as the first stage of the adder tree. Therefore, APC reduces the gate count by about 40% compared to the conventional adder tree. Here, the APC can also be used to sum the 1-bit outputs from AND gates in SPSC-TVM while keeping the same gate count as our OR gate-based adder tree. However, it cannot be compatible with the proposed SPSC-TVM well to achieve high computing accuracy. Note that our SPSC-TVM uses two deterministic encoders TSG and USG, so these 1-bit outputs from AND gates are not totally random or uniform but rather obey the pattern that zeros concentrate in the middle. Therefore, the connection type between these 1-bit outputs and the inputs of APC can significantly affect the computing error. Here, we show three connection types that beat others in our numerical experiments. On the left table, the computing accuracy of SPSC-TVM outperforms APC-based schemes in terms of MAE (%).

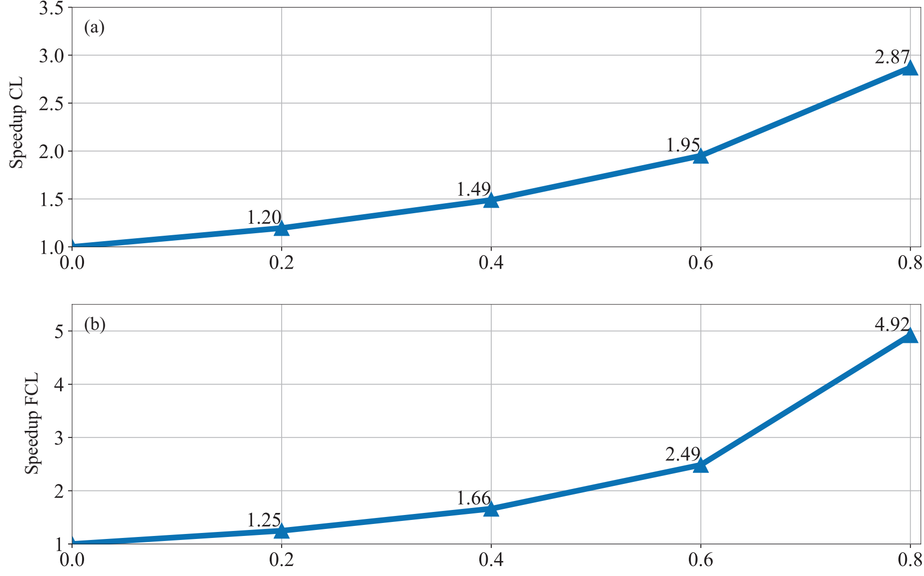


Figure 12 (Color online) Sensitivity analysis of the processing of CLs and FCLs under difference sparsity levels. Here, in the analysis of CLs, to provide an accurate sparsity level that ignores the influence of zero paddings introduced by paring weights, we directly put [(1 − weight sparsity) × (# of total weights)] weights into the weight buffer. (a) Weight sparsity; (b) activation sparsity.

7.3 Sensitivity analysis

To explore the potential of SPA, we analyze the sensitivity of performance under different sparsity levels of weights in CLs, and IAs in FCLs, respectively, since SPA has the independent CL engine and FCL engine.

For CLs, in this experiment, the kernel size is set to the common configuration 3×3 and the number of input channels is 200. The average computation time of one kernel was recorded for the analysis as the proposed SCPP dataflow chooses the kernel dimension as the outermost loop. Figure 12(a) shows the speedup normalized to the dense case when the level of weight sparsity varies from 0.0 to 0.8 with a granularity of 0.2. We find that SPA can benefit from weight sparsity efficiently. For instance, when the weight sparsity level is 80%, SPA achieves 2.87× speedup. The reason why SPA cannot obtain the ideal 5× speedup is that the time of loading data, control and offloading results still takes some proportion.

Figure 12(b) shows the effect of different levels of activation sparsity in FCLs. It can be observed that SPA can take advantage of the sparsity of IAs in FCLs greatly. For example, when the sparsity level is 0.8 (i.e., 80% IAs are zero), the FCL engine obtains 4.92× speedup, which is close to the theoretical maximum 5× speedup. This is because the FCL engine has only one MAC unit, making a relatively high utilization, which means the reduced amount of IAs is nearly directly translated into the improvement of performance.

7.4 Overall evaluation

To evaluate the classification performance of the proposed SPSC when applied to CNNs, we adopted Zhang's work [12] to train the sparse CNN models with floating-point precision. Then, we retrained these sparse CNN atop the original training. Zero weights in CLs are fixed in the retraining phase to maintain the sparsity. Both weights and biases are quantized to fixed-point numbers and SPSC-based multiplication is used in the forward pass. We use one CNN model designed for MNIST and FashionMNIST datasets and compare two cases: (a) floating-point precision; (b) the proposed SPSC. The size of the input image is padded to 32×32.

For MNIST CNN, the weight sparsity of CLs (biases are excluded) before and after the weight pairing is 94.86% and 93.36%, respectively. Therefore, MAWP only sacrifices 1.5% sparsity. The accuracy before and after retraining is respectively 99.17% and 99.42%. The accuracy after retraining even outperforms the floating-point accuracy of 99.39%. This promising result mainly comes from the fact that MNIST is fairly easy. For the more challenging FashionMNIST dataset, we set the sparsity to 90.83%, which is less than that of MNIST CNN, to obtain more retrainable weights to compensate for the accuracy loss. After applying the proposed MAWP, the weight sparsity only decreases by 1.81%. Without retraining,

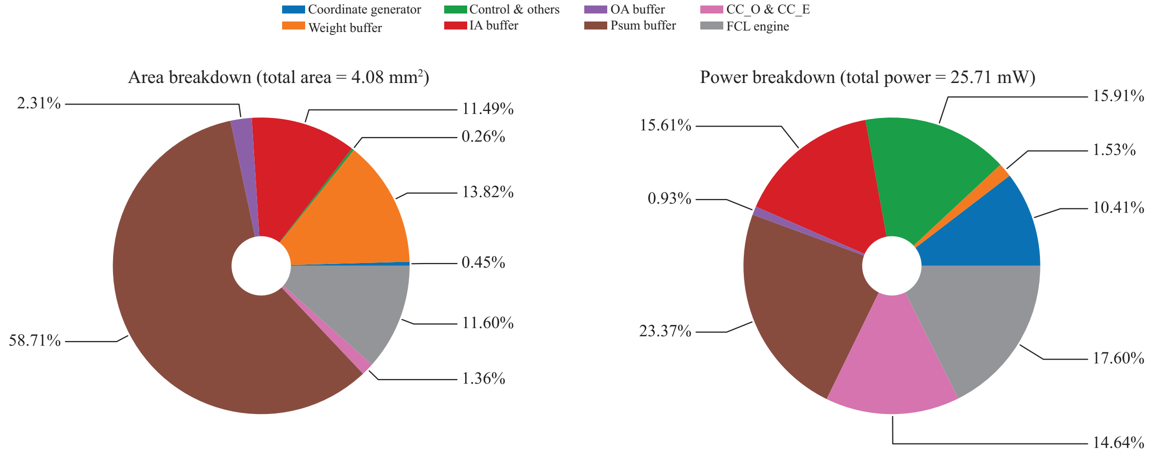


Figure 13 (Color online) Power and area breakdown of SPA.

the proposed SPSC achieves 91.56% accuracy on FashionMNIST CNN. The accuracy after retraining is 92.53%, which is comparable with the floating-point result, 92.85%.

Figure 13 shows the area and power breakdown of SPA. All on-chip buffers, including IA buffer, Weight buffer, OA buffer, and Psum buffer account for 97.66% area of the entire CL engine. However, this percentage decreases to 50.30% in the power report owing to the latch-based buffers trade resources for better power performance. Note that IA buffer exhibits an inverse trend as opposed to the other three buffers. In detail, the proportion of IA buffer in the whole chip increases from 11.49% in the area breakdown to 15.61% in the power breakdown as it has two wide read ports to support the indexing from two weights (i.e., a weight pair) simultaneously. Two CCs take up only 1.36% area of SPA, thanks to the proposed SPSC's low-cost merit. The independent FCL engine takes only 11.60% area and 17.60% power consumption, which obeys the design principle of SPA.

Table 1 [20, 30, 34–37] provides the overall comparison with existing CNN accelerators. We compare SPA with three types of accelerators, including the sparse CNN accelerator ZASCA [30], the SC-based implementation of CNNs [20, 34], and the binary weight CNN and DNN accelerator [35, 36].

The SC-based accelerator [20] achieves the best energy efficiency because of its fully parallel implementation, which comes at the cost of much limited reconfigurability, leading to little compatibility with current versatile neural networks and fast evolving algorithms. Another state-of-the-art SC-based CNN accelerator, ACOUSTIC [34], is a programmable, full-system SC implementation. The results of ACOUSTIC are reported from Table-II, ACOUSTIC ULP-128 in [38] (the same team as [34]). SPA outperforms ACOUSTIC by 7.7× in terms of energy efficiency when all data are scaled to 65 nm with a supply voltage of 1.0 V. Compared with the digital implementation of binary CNN [35], SPA achieves 4.9× better energy efficiency. We also surpass [36] on energy efficiency, which integrates six processing-in-memory modules capable of hosting a maximum 13-layer, 4.2k-neuron/0.8M-synapse binary/ternary DNN in 3.9 mm^2. Ref. [30] only skips zero IAs, which has less energy efficiency compared with our design. In a word, SPA achieves a relatively high energy efficiency while maintaining the sufficient reconfigurability and exploiting data sparsity.

To further analyze and demonstrate the high accuracy of the proposed SPSC and the low zero-padding ratio of the proposed MAWP, we show the sparsity and classification accuracy on different network types and datasets. The results are summarized in Table 2. When the overall weight sparsity in CLs is as high as 88.7%/85.8%, MAWP leads to only 3.6%/3.7% sparsity loss in the large CNN VGG13-Like and challenging datasets CIFAR10/CIFAR100. The small loss comes from both the efficient MAWP and the fact that, after being scaled and quantized, the magnitude of 71.1% (CIFAR10)/74.2% (CIFAR100) non-zero weights appears in the lower half range. Therefore, most weights can be paired successfully. However, for the small CNN LeNet5, the weight sparsity drops from 91.3% tp 83.5% after pairing. The reason is that, in LeNet5, the maximum channel count is 16 which is less than the weight workload size 32 (Subsection 5.3), leading to an inadequate searching space for weight pairing. Please note that future deep CNNs tend to be increasingly deeper and larger. For the analysis of classification accuracy, we use the accuracy after pruning as the baseline. In the two simpler tasks, MNIST and FashionMNIST, even with the weight sparsity of 91.3% and the small LeNet5, our SPSC introduces only 0.09% and

Table 1 Comparison with existing neural network accelerators

	TC'2018 [30]	SiPS'2019 [20]	ISCAS'2018 [35]	JSSC'2018 [36]	DATE'2020 [34]	This work
Tech. (nm)	65	65	65	65	28(65)$^{f)}$	55(65)$^{f)}$
Status	Layout	Synthesis	Synthesis	Tape-out	Layout	Synthesis
Core area (mm^2)	6	0.03	1.2	3.9	0.57(3.07)$^{f)}$	4.08(5.70)$^{f)}$
On-chip buff. (kB)	36.9	–	18.4	306	–	77.125
Weight/IA (bit)	16/16	3/3	1/12	1–2/1–2	–	6/15$^{a)}$
Layer type support	CL+FCL	CL+FCL	CL+FCL	FCL	CL+FCL	CL+FCL
Reconfig. support$^{b)}$	All	None	Not All$^{c)}$	–	All	All
Sparsity support	IA	No	No	No	No	Weight+IA
Network type	AlexNet	LeNet5	VGG-like	13 FCLs	LeNet5	CNN$^{d)}$
Dataset	ImageNet	MNIST	CIFAR10	MNIST	MNIST	MNIST/F.MNIST
Supply voltage (V)	1.0	–	1.0	1.0	0.9(1.0)$^{f)}$	1.2(1.0)$^{f)}$
Frequency (MHz)	200	–	200	400	400(172)$^{f)}$	300(254)$^{f)}$
Power (mW)	270	0.35	128	580	72(206.35)$^{f)}$	25.71(21.10)$^{f)}$
Performance (Gops)	93.6	–	67.4	1264.4	160 (68.9)$^{f)}$	63.7/55.9 (53.9/47.3)$^{e),f)}$
Energy efficiency (Gops/W)	346.7	3620.3	526.6	2172.42	2220 (333.9)$^{f)}$	2477.6/2174.3 (2554.5/2241.7)$^{e),f)}$
Accuracy (%)	81.1$^{g)}$	98.92	–	90.1	99.3	99.42/92.53

a) This denotes data width used in CLs. For FCLs, both weights and IAs are represented by 16 bits.
b) All reconfigurability means variable sizes of kernels, input feature maps, input/output channels, strides and zero paddings.
c) The kernel size of [35] is fixed to 3 × 3.
d) The configuration of the CNN model for both MNIST dataset and FashionMNSIT dataset is: CL1: (1, 16, 3, 3, 1), ReLU, CL2: (16, 32, 3, 3, 1), ReLU, MaxPooling1, CL3: (32, 64, 3, 3, 1), ReLU, CL4: (64, 240, 3, 3, 1), ReLU, MaxPooling2, CL5: (240, 240, 7, 7, 1), ReLU, CL6: (240, 240, 7, 7, 1), ReLU, MaxPooling3, FCL1: (3840, 512), ReLU, FCL2: (512, 256), ReLU, FCL3: (256, 128), ReLU, FCL4: (120, 10). Here, the five-tuple CLx denotes (C, M, R, S, Strd_H&Strd_W), the two-tuple FCLx denotes (K, N). All MaxPooling layers: kernel size (2,2) and stride 2.
e) Results of MNIST CNN/FashionMNIST CNN.
f) For a fair comparison, these data are normalized to 65 nm and 1.0 V CMOS process according to the scaling method provided in [37]: Frequency $\sim s$, Area $\sim 1/s^2$, Power $\sim (1/s)(1/V'_{\rm dd})^2$, where s = Technology/65 nm.
g) Top-5 accuracy reported from [2]. Ref. [30] maintains the same accuracy as the dense model because it only exploits IA sparsity.

Table 2 Comprehensive analysis on sparsity and classification accuracy

Dataset	Network type	Weight sparsity (%)		Accuracy (%)		
		w/o MAWP	w/ MAWP	w/o pruning w/o SPSC	w/ pruning w/o SPSC	w/ pruning w/ SPSC
MNIST	LeNet5	91.3	83.5	99.05	98.99	98.90
FashionMNIST	LeNet5	91.3	83.5	90.23	90.20	90.05
CIFAR10	VGG13-Like$^{a)}$	88.7	85.1	89.42	89.28	89.03
CIFAR100	VGG13-Like	85.8	81.9	64.90(86.83)$^{b)}$	64.74(87.34)	64.25(87.53)

a) Details of VGG13-Like: CL1+BN: (3, 32, 3, 3, 1), ReLU, CL2+BN: (32, 32, 3, 3, 1), ReLU, MaxPooling1, CL3+BN: (32, 64, 3, 3, 1), ReLU, CL4+BN: (64, 254, 3, 3, 1), ReLU, MaxPooling2, [CL+BN: (254, 254, 3, 3, 1), ReLU]*3, MaxPooling3, [CL+BN: (254, 254, 5, 5, 1), ReLU]*3, MaxPooling4, FCL1: (1016, 512), ReLU, FCL2: (512, 512), ReLU, FCL3: (512, 256), ReLU, FCL4: (256, 10/100). All MaxPooling layers: kernel size (2,2) and stride 2.
b) top-1 and top-5 accuracy.

0.15% accuracy loss, respectively. Regarding the more complex CIFAR10 and CIFAR100 with larger CNN VGG13-Like, SPSC achieves 0.25% and 0.49% (top-1) accuracy loss, respectively. These promising results are a corollary of the error analysis of the SPSC-based multiplier in Subsection 4.1, where we have verified that the computing error can be modeled as the sum of independent uniform distributed round-off errors. Therefore, the SPSC-based multiplication is an unbiased estimation of the exact multiplication. Moreover, in CLs, an OA typically requires a large amount of SPSC multiplications, thus the relative computing error is further reduced. Please note that we also use retraining to mitigate the accuracy loss.

8 Conclusion

In this work, the SPA accelerator toward energy-efficient processing of CNNs was demonstrated. SPA features the innovations and optimizations spanning from the circuit level to the architecture level. First,

following our previous work NSPC, we proposed and proved the weight-adjustable SPSC, which generates deterministic bitstreams and performs multiplications within only one clock cycle, overcoming the long computational latency of traditional SC. Second, the proposed SPSC-TVM with BSO exploring the special bits distribution of the thermometer sequence further reduces the logical overhead. With these circuit-level optimizations, SPSC-TVM gains 45.2%–18.9% less area and 38.7%–21.7% less power consumption compared to the fixed-point counterpart. Third, HZSS was proposed in accordance with SPSC-based arithmetic circuits to apply different zero-skipping schemes to different types of layers. Fourth, the specialized CL and FCL engines with the three-level time interleaving technique were designed to efficiently improve the processing speed. SPA achieves an energy efficiency of up to 2477.6 Gops/W with only 4.08 mm^2 while providing high reconfigurability to solve versatile sparse CNN models, delivering high performance in power/area and energy/resource-constrained intelligent edge device envelope.

Acknowledgements This work was supported by National Key Research and Development Program (Grant No. 2020YFB2-205500).

References

1 LeCun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444
2 Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, Lake Tahoe, 2012. 1097–1105
3 He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, 2016. 770–778
4 Ren S Q, He K M, Girshick R, et al. Faster R-CNN: towards real-time object detection with region proposal networks. In: Proceedings of Advances in Neural Information Processing Systems, Montreal, 2015. 91–99
5 Sze V, Chen Y H, Yang T J, et al. Efficient processing of deep neural networks: a tutorial and survey. Proc IEEE, 2017, 105: 2295–2329
6 Zhou A J, Yao A B, Guo Y W, et al. Incremental network quantization: towards lossless CNNs with low-precision weights. In: Proceedings of the 5th International Conference on Learning Representations, Toulon, 2017. 1–14
7 Courbariaux M, Bengio Y, David J P, et al. Binaryconnect: training deep neural networks with binary weights during propagations. In: Proceedings of Advances in Neural Information Processing Systems, Montral, 2015. 3123–3131
8 Moons B, de Brabandere B, van Gool L, et al. Energy-efficient convnets through approximate computing. In: Proceedings of IEEE Winter Conference on Applications of Computer Vision, Lake Placid, 2016. 1–8
9 Chen C Y, Choi J, Gopalakrishnan K, et al. Exploiting approximate computing for deep learning acceleration. In: Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition, Dresden, 2018. 821–826
10 Cheng C D, Tiw P J, Cai Y M, et al. In-memory computing with emerging nonvolatile memory devices. Sci China Inf Sci, 2021, 64: 221402
11 Qian X H. Graph processing and machine learning architectures with emerging memory technologies: a survey. Sci China Inf Sci, 2021, 64: 160401
12 Zhang T Y, Ye S K, Zhang K Q, et al. A systematic DNN weight pruning framework using alternating direction method of multipliers. In: Proceedings of the European Conference on Computer Vision, Munich, 2018. 184–199
13 Guo L H, Chen D W, Jia K. Knowledge transferred adaptive filter pruning for CNN compression and acceleration. Sci China Inf Sci, 2022, 65: 229101
14 Brown B D, Card H C. Stochastic neural computation. I. Computational elements. IEEE Trans Comput, 2001, 50: 891–905
15 Naderi A, Mannor S, Sawan M, et al. Delayed stochastic decoding of LDPC codes. IEEE Trans Signal Process, 2011, 59: 5617–5626
16 Zhang C, Parhi K K. Latency analysis and architecture design of simplified SC polar decoders. IEEE Trans Circ Syst II, 2013, 61: 115–119
17 Kim K, Kim J, Yu J, et al. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In: Proceedings of the 53nd ACM/EDAC/IEEE Design Automation Conference, Austin, 2016. 1–6
18 Li Z, Li J, Ren A, et al. HEIF: highly efficient stochastic computing-based inference framework for deep neural networks. IEEE Trans Comput-Aided Des Integr Circ Syst, 2018, 38: 1543–1556
19 Xie Y, Liao S Y, Yuan B, et al. Fully-parallel area-efficient deep neural network design using stochastic computing. IEEE Trans Circ Syst II, 2017, 64: 1382–1386
20 Zhang Y W, Zhang X Y, Song J H, et al. Parallel convolutional neural network (CNN) accelerators based on stochastic computing. In: Proceedings of IEEE International Workshop on Signal Processing Systems, Nanjing, 2019. 19–24
21 Sim H, Lee J. A new stochastic computing multiplier with application to deep convolutional neural networks. In: Proceedings of the 54th Annual Design Automation Conference, Austin, 2017. 1–6
22 Chen Y H, Krishna T, Emer J S, et al. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J Solid-State Circ, 2016, 52: 127–138
23 Han S, Liu X Y, Mao H Z, et al. EIE: efficient inference engine on compressed deep neural network. In: Proceedings of the 43rd Annual International Symposium on Computer Architecture, Seoul, 2016. 243–254
24 Parashar A, Rhu M, Mukkara A, et al. SCNN: an accelerator for compressed-sparse convolutional neural networks. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, 2017. 27–40
25 Zhang J F, Lee C E, Liu C, et al. SNAP: an efficient sparse neural acceleration processor for unstructured sparse deep neural network inference. IEEE J Solid-State Circ, 2021, 56: 636–647
26 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. In: Proceedings of International Conference on Learning Representations, San Diego, 2015. 1–14
27 Xia Z H, Chen J N, Huang Q, et al. Neural synaptic plasticity-inspired computing: a high computing efficient deep convolutional neural network accelerator. IEEE Trans Circ Syst I, 2020, 68: 728–740
28 Liu S T, Han J. Energy efficient stochastic computing with sobol sequences. In: Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition, Lausanne, 2017. 650–653

29 Anderson J H, Hara-Azumi Y, Yamashita S. Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy. In: Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition, Dresden, 2016. 1550–1555

30 Ardakani A, Condo C, Gross W J. Fast and efficient convolutional accelerator for edge computing. IEEE Trans Comput, 2019, 69: 138–152

31 Zhang C, Li P, Sun G Y, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, 2015. 161–170

32 Teman A, Rossi D, Meinerzhagen P, et al. Power, area, and performance optimization of standard cell memory arrays through controlled placement. ACM Trans Des Autom Electron Syst, 2016, 21: 1–25

33 Kim K, Lee J, Choi K. Approximate de-randomizer for stochastic circuits. In: Proceedings of IEEE International SoC Design Conference (ISOCC), Gyeongju, 2015. 123–124

34 Romaszkan W, Li T, Melton T, et al. ACOUSTIC: accelerating convolutional neural networks through or-unipolar skipped stochastic computing. In: Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, 2020. 768–773

35 Ardakani A, Condo C, Gross W J. A convolutional accelerator for neural networks with binary weights. In: Proceedings of IEEE International Symposium on Circuits and Systems, Florence, 2018. 1–5

36 Ando K, Ueyoshi K, Orimo K, et al. BRein memory: a single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 TOPS at 0.6 W. IEEE J Solid-State Circ, 2018, 53: 983–994

37 Wang H Z, Xu W H, Zhang Z C, et al. An efficient stochastic convolution architecture based on fast FIR algorithm. IEEE Trans Circ Syst II, 2022, 69: 984–988

38 Li T, Romaszkan W, Pamarti S, et al. GEO: generation and execution optimized stochastic computing accelerator for neural networks. In: Proceedings of IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, 2021. 689–694