SC-DCNN: Highly-Scalable Deep Convolutional Neural Network using Stochastic Computing

Ao Ren[1], Ji Li[2], Zhe Li[1], Caiwen Ding[1], Xuehai Qian[2], Qinru Qiu[1], Bo Yuan[3] and Yanzhi Wang[1]

[1]Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY, USA
[2]Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA
[3]Department of Electrical Engineering, City University of New York, City College, NY, USA
aren@syr.edu, jli724@usc.edu, {zli89,cading}@syr.edu, xuehai.qian@usc.edu, qiqiu@syr.edu, byuan@ccny.cuny.edu, ywang393@syr.edu

Abstract

With recent advancing of wearable devices and Internet of Things (IoTs), it becomes very attractive to implement the deep convolutional neural networks (DCNNs) onto embedded and portable systems. Presently, executing the software-based DCNNs requires high-performance server clusters in practice, restricting their widespread deployment on the personal and mobile devices. In order to overcome this issue, considerable research efforts have been conducted in the context of developing highly-parallel and specific DCNN hardware, utilizing GPGPUs, FPGAs, and ASICs.

Stochastic Computing (SC), which uses a bit-stream to represent a number within [-1, 1] by counting the number of ones in the bit-stream, has high potential for implementing DCNNs with high scalability and ultra-low hardware footprint. Since multiplications and additions can be calculated using AND gates and multiplexers in SC, significant reductions in power (energy) and hardware footprint can be achieved compared to the conventional binary arithmetic implementations. The tremendous savings in power (energy) and hardware resources bring about immense design space for enhancing scalability and robustness for hardware DCNNs.

This paper presents the first comprehensive design and optimization framework of SC-based DCNNs (SC-DCNNs), using a bottom-up approach. We first present the optimal designs of function blocks that perform the basic operations, i.e., inner product, pooling, and activation function, in DCNN. Then we propose the optimal design of four types of combinations of basic function blocks, named feature extraction blocks, which are in charge of extracting features from input feature maps. Besides, weight storage methods are proposed and investigated to reduce the area and power (energy) consumption for storing weights. Finally, the whole SC-DCNN implementation is optimized, with feature extraction blocks carefully selected, to minimize area and power (energy) consumption while maintaining a high network accuracy level. Experimental results indicate that the proposed SC-DCNN implementing LeNet5 consumes only 17 mm^2 area and 1.53 W power, and achieves throughput of 781250 images/s, area efficiency of 45946 images/s/mm^2, and energy efficiency of 510734 images/J.

1. Introduction

In the recent decade, deep learning, or deep structured learning, has emerged as a new area of machine learning research, which enables a system to automatically learn complex information and extract representations at multiple levels of abstraction [1]. *Deep Convolutional Neural Network (DCNN)*, one of the most promising types of artificial neural networks taking advantage of deep learning, has been recognized as the dominant approach for almost all recognition and detection tasks [2]. Specifically, DCNN has achieved significant success in a wide range of machine learning applications, such as image classification [3], natural language processing [4], speech recognition [5], and video classification [6].

High-performance server clusters are usually required for executing software-based DCNNs since software-based DCNN implementations involve a large amount of computations so as to achieve outstanding performance. However, the use of server clusters implies high power (energy) consumptions and large hardware volumes, and is therefore inappropriate for low-power applications in personal and mobile devices, which are playing an increasingly important role in our everyday life and exhibit a notable trend of being "smart". To overcome the limitation of low-power and low-hardware footprint implementations of DCNNs, utilizing highly-parallel or dedicated hardware has attracted much academic and industrial attention in recent years, including the works utilizing *General-Purpose Graphics Processing Units (GPGPUs)*, *Field-Programmable Gate Array (FPGAs)*, and *Application-Specific Integrated Circuit (ASICs)* to implement DCNNs [7–18]. Despites the performance and power (energy) efficiency gains, a large margin of improvement still exists due to the inherent inefficiency in implementing DCNNs using conventional computing methods or using general-purpose computing devices [19, 20].

Novel computing paradigms need to be investigated in order to provide the ultra-low hardware footprint and therefore the highest possible energy efficiency and scalability. *Stochastic Computing (SC)*, which represents a probability number using a bit-stream [21], has the potential to implement DCNNs with significantly reduced hardware resources and achieve high power (energy) efficiency, and therefore can potentially trigger a revolutionary reshaping of hardware design of large-scale deep learning systems. To be more specific, in SC, key arithmetic calculations such as multiplications and additions can be implemented as simple as AND gates and multiplexers (MUX), respectively [22]. Considering the large number of multiplications and additions in DCNN, the efficient implementations using stochastic computing save a large design space for further improvements on the parallelism degree.

Inspired by the promising characteristics, in this paper, we propose the first comprehensive design and optimization framework of SC-based DCNNs (SC-DCNNs), using a bottom-up approach. The proposed SC-DCNN fully utilizes the advantages of SC technology, and could achieve ultra-low hardware footprint, low power and energy consumption, while maintaining high network accuracy level. Besides the SC-DCNN architecture itself, key contributions in the proposed design and optimization framework are listed as follows:

- **Basic function blocks and hardware-oriented max pooling.** We first design and investigate the *function blocks* that perform the basic operations, i.e., inner product, pooling, and activation functions, in DCNN. More specifically, we present a novel hardware-oriented max pooling design for effectively implementing (approximate) max pooling in SC domain. We thoroughly investigate the pros and cons of different types of function block implementations.
- **Joint optimizations for feature extraction blocks.** We propose the optimal designs of four types of combinations of basic function blocks, named *feature extraction blocks*, which are in charge of extracting features from input feature maps. The function blocks inside the feature extraction block are jointly optimized through both analysis and experiments with respect to input bit-stream length, function block structure, and function block compatibilities.
- **Weight storage schemes.** We present effective designs and optimizations on weight storage to reduce the corresponding area and power (energy) consumptions, including efficient filter-aware SRAM sharing, effective weight storage methods, and layer-wise weight storage optimizations.
- **Overall SC-DCNN optimization.** We conduct thorough optimizations on the overall SC-DCNN, with feature extraction blocks carefully selected, to minimize area and power (energy) consumption while maintaining a high network accuracy level. The optimization procedure leverages the important observation that hardware inaccuracies in different layers in DCNN have different effects on the overall network accuracy, therefore different designs may be exploited to minimize area and power (energy) consumptions.
- **Ultra-low hardware footprint and low power (energy) consumptions.** Overall, the proposed SC-DCNN achieves the lowest hardware cost and energy consumption in implementing LeNet5 compared with reference works.

2. Related Works

Authors in [7, 8, 23, 24] leveraged the parallel computing and storage resources in GPUs for efficient DCNN implementations. FPGA-based accelerators, benefited from the advantages of being programmable, high degree of parallelism and short develop round, is another promising path towards the hardware implementation of DCNNs [9, 10]. However, these GPU and FPGA-based implementations still exhibit a large margin of performance enhancement and power reduction. This is because (i) GPUs and FPGAs are general-purpose computing devices not specifically optimized for executing DCNNs, and (ii) the relatively limited signal routing resources in such general platforms will restrict the performance of DCNNs which exhibit high inter-neuron communication requirements.

ASIC-based implementations of DCNNs have been recently exploited to overcome the limitations of general-purpose computing devices. Two representable recent works on ASIC-based implementations are DaDianNao [17] and EIE [18]. The former proposes an ASIC "node" which could be connected in parallel to implement a large-scale DCNN, whereas the latter focuses specifically on the fully-connected layers of DCNN and achieves high throughput and energy efficiency.

Novel computing paradigms need to be investigated in order to provide the ultra-low hardware footprint and the highest possible energy efficiency and scalability. Stochastic computing-based design of neural networks is an attractive candidate to meet the above goals and facilitate the widespread of DCNNs in personal, embedded, and mobile IoT devices. Although not focusing on deep learning systems, predecessors in [25] proposed the design of a neurochip using stochastic logic. Reference [19] utilized stochastic logic to implement a radial basis function-based neural network, and the neuron design with SC for deep belief network was presented in [20]. However, there is no existing work that investigates comprehensive designs and optimizations of SC-based hardware DCNNs including both computation blocks and weight storing methods.

3. Overview of DCNN Architecture and Stochastic Computing

3.1. *DCNN Architecture Overview*

Deep convolutional neural networks are biologically inspired variants of multilayer perceptrons (MLPs) by mimicking the animal visual mechanism [26]. An animal visual cortex contains two types of cells and they are only sensitive to a small region (receptive field) of the visual field. Thus a neuron in a DCNN is only connected to a small receptive field of its previous layer, rather than connected to all neurons of previous layer like traditional fully connected neural networks.

As shown in Figure 1, each layer of DCNN is a 3D volume that has neurons arranged in three dimensions: *height* $\times$ *width* $\times$ *depth*. Height and width refer to the size of one feature map, while depth represents the number of feature maps. A whole feature map is covered by tiling receptive fields [26].

A DCNN is in the simplest case a stack of three types of layers: *Convolutional Layer*, *Pooling Layer*, and *Fully Connected Layer*. The Convolutional layer is the core building block of DCNN, and the main operation is the convolution that calculates the dot-product of receptive fields and a set of learnable filters (or kernels) [27]. Figure 2 illustrates the process of convolution operations. Suppose that the size of the input feature map is 7×7, and the size of a filter is 3×3, thus the feature map is divided into nine receptive fields if the stride is two. The first and ninth elements of the output feature map are computed by respectively convolving the first and ninth receptive fields with the filter.

After the convolution operations, the nonlinear down-samplings are conducted in the pooling layers for reducing the dimension of data. The most common pooling strategies are *max pooling* and *average pooling*. Max pooling is to pick up the maximum value from the candidates, and average pooling is to calculate the average value of the candidates. Then the extracted feature maps after down-sampling operations are sent to *activation functions* that conduct non-linear transformations such as Rectified Linear Unit (ReLU) $f(x) = max(0, x)$, Sigmoid function $f(x) = (1 + e^{-x}) - 1$ and hyperbolic tangent (tanh) function $f(x) = \frac{2}{1+e^{-2x}} - 1$. Finally, the high-level reasoning is completed via the fully connected layer. Neurons in this layer are connected to all activation results in the previous layer. Finally, the loss layer is normally the last layer of DCNN and it specifies how the deviation between the predicted and true labels is penalized in the network training process. Various loss functions such as softmax loss, sigmoid cross-entropy loss and so on may be used for different tasks.

The main operations in DCNNs are inner product, pooling, and activation function operations, as shown in Figure 3 (a), (b), and (c), respectively. In convolutional layers, the inner product operation is performed by a convolutional neuron to calculate the dot-product of a receptive field (x_i's in Figure 3 (a)) and a filter (w_i's in Figure 3 (a)). Generally, the inner products are then subsampled through pooling operations performed by pooling neurons, and Figure 3 (b) shows the average pooling and max pooling which are studied in this work. The subsampled outputs are transformed by an activation function shown in Figure 3 (c) to ensure the inputs of the next layer are within the [-1, 1] range. In the fully connected layer, x_i is the i-th activation output from the previous layer and w_i is a weight of the corresponding link, and they are also inputs of neurons in the fully connected layer.

The concept of "neuron" is widely used in the software/algorithm domain. In the context of DCNNs, a neuron

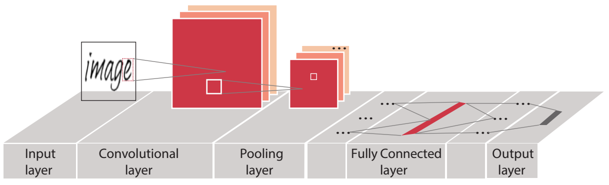

Figure 1: The general DCNN architecture.

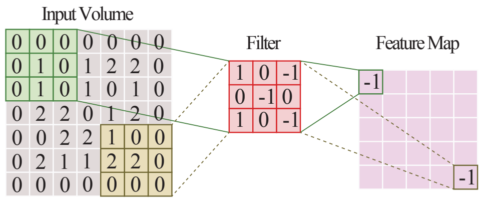

Figure 2: Illustration of the convolution process.

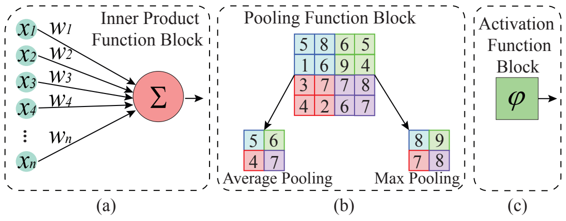

Figure 3: Three types of basic operations (function blocks) in DCNN. (a) Inner Product, (b) pooling, and (c) activation.

may consist of one or multiple basic operations. For example, neurons in convolutional layers implement inner product operations only; those in pooling layers implement pooling and activation operations; and those in fully connected layers implement inner product and activation operations. Since this paper focuses on hardware designs and optimizations, we focus on the basic operations, i.e., inner product, pooling, and activation, and the corresponding SC-based designs of these fundamental operations are termed *function blocks*. Furthermore, different function blocks (main operations) need to be jointly optimized with respect to the bit-stream length and structure compatibilities (e.g., an APC-based inner product block needs to connect to a Btanh-based activation function block). The composition of an inner product block, a pooling block, and an activation function block is referred to as the *feature extraction block*, which takes charge of extracting features from feature maps. The design and optimizations of the basic function blocks and feature extraction blocks will be discussed in Section 4.

3.2. Stochastic Computing (SC)

Stochastic computing is a technology that represents a probabilistic number by counting the number of ones in a bit-stream. For instance, the bit-stream 0100110100 contains four ones in a ten-bit stream, thus it represents $P(X = 1) = 4/10 = 0.4$. In addition to this unipolar encoding format, SC can also represent numbers in the range of [-1, 1] using the bipolar encoding format. In the scenario of bipolar encoding scheme, a real number x is processed by $P(X = 1) = (x + 1)/2$, thus 0.4 can be represented by 1011011101. To represent a number beyond the range [0, 1] using unipolar format or beyond [-1, 1] using bipolar format, a pre-scaling operation [28] can be used.

The major advantage of stochastic computing is its much lower hardware cost on a large category of arithmetic calculations, when compared to conventional binary computing. The abundant area budget offers immense design space in optimizing hardware performance via efficient tradeoffs between the area and other metrics, such as power, latency, and parallelism degree, and thus becomes a promising technology for implementing large-scale DCNNs.

Multiplication. Figure 4 shows the basic multiplication components in SC domain. A unipolar multiplication can be performed by an AND gate since $P(A \cdot B = 1) = P(A = 1)P(B = 1)$ (assuming independence of two random variables), and a bipolar multiplication is performed by means of a XNOR gate since $c = 2P(C = 1) - 1 = 2(P(A = 1)P(B = 1) + P(A = 0)P(B = 0)) - 1 = (2P(A = 1) - 1)(2P(B = 1) - 1) = ab$.

Addition. In this paper, four popular stochastic addition methods are investigated, optimized, and carefully selected for SC-DCNNs. An OR gate in Figure 5 (a) is the simplest method that consumes the least hardware footprint to perform an addition, but this method will introduce much accuracy loss because the computation "logic 1 OR logic 1" only generates a single logic 1 and results in inaccuracy. The second component in Figure 5 is a multiplexer, which is the most popular method to perform additions in either the unipolar or the bipolar format [22]. For example, a bipolar addition is performed as $c = 2P(C = 1) - 1 = 2(1/2(P(A = 1) + 1/2P(B = 1)) - 1 = 1/2(2P(A = 1) - 1) + (2P(B = 1) - 1)) = 1/2(a + b)$. *Approximate parallel counter (APC)* depicted by Figure 5 (c) is proposed in [29], and it calculates the summation of inputs by accumulating the number of ones. It consumes fewer logic gates when compared with the conventional accumulative parallel counter [29, 30]. The fourth implementation of stochastic addition uses two-line representation of a stochastic number, which is proposed in [31]. The two-line representation consists of a magnitude stream $M(X)$ and a sign stream $S(X)$, in which 1 represents a negative bit and 0 represents a positive bit. The value of the represented stochastic number is calculated by: $x = \frac{1}{L}\sum_{i=0}^{L-1}(1 - 2S(X_i))M(X_i)$, where L is the length of the bit-stream. As an example, -0.5 can be represented by $M(-0.5)$: 10110001 and $S(-0.5)$: 11111111.

Hyperbolic Tangent (tanh). The tanh function is highly suitable for stochastic computing-based implementations, for the reasons that (i) it can be easily implemented with a K-state *finite state machine (FSM)* in the SC domain [22] and causes less hardware cost when compared to the piecewise linear approximation (PLAN)-based implementation [32] in conventional computing domain, and (ii) replacing ReLU or sigmoid function by tanh function does not cause accuracy loss in DCNN [23]. Therefore we choose tanh as the activation function in SC-DCNNs in this work. The diagram of the FSM is shown in Figure 6. It will output a zero when the current state is on the left half of the diagram, otherwise output a one. The value calculated by the FSM satisfies $Stanh(K, x) \cong tanh(\frac{K}{2}x)$, where *Stanh* denotes stochastic tanh.

3.3. Network Accuracy vs. (Hardware) Accuracy

The overall network accuracy (e.g., the overall recognition or classification rates) is one of the key optimization goals of the SC-based hardware DCNN. On the other hand, the SC-based function blocks and feature extraction blocks exhibit certain degree of inaccuracy due to the inherent stochastic nature. The network accuracy and hardware accuracies are different but correlated, i.e., high accuracies in each function block will likely lead to a high overall network accuracy. Hence, the hardware accuracies will be optimized in the design of SC-based function blocks and feature extraction blocks.

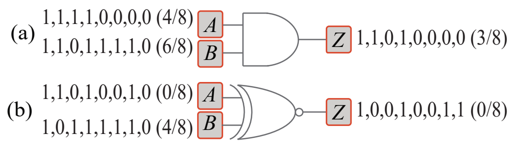

Figure 4: Stochastic multiplication. (a) Unipolar multiplication and (b) bipolar multiplication.

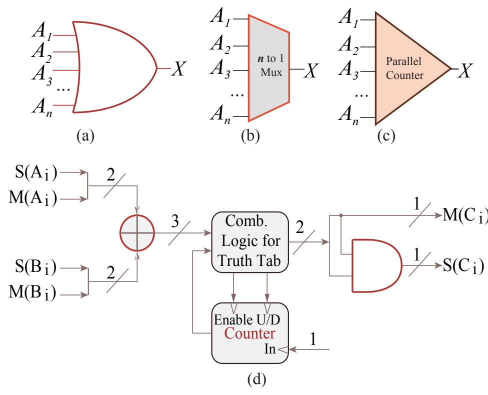

Figure 5: Stochastic addition. (a) OR gate, (b) MUX, (c) APC, and (d) two-line representation-based adder.

4. Design and Optimization for Function Blocks and Feature Extraction Blocks in SC-DCNN

In this section, we first perform comprehensive designs and optimizations in order to derive the most efficient SC-based implementations for function blocks, including inner product/convolution, pooling, and activation function, in terms of power, energy, and hardware resource, meanwhile maintaining a high accuracy level. Based on the detailed analysis of pros and cons of each type of basic function block design, we propose the optimal designs of feature extraction blocks for SC-DCNNs through both analysis and experiments.

4.1. Inner Product/Convolution Block Design

As shown in Figure 3 (a), an inner product/convolution block in DCNNs is composed of multiplication and addition operations. Since in SC-DCNNs, inputs are distributed in the range of [-1, 1], we adopt the bipolar multiplication implementation (the XNOR gate) for the inner product block design. The summation of all products is performed by the adder(s). In the SC domain, the addition operation has various possible implementations, such as OR gate-based, multiplexer (MUX)-based, APC-based, and two-line representation-based adders. Therefore, we present and investigate four types of inner product block structures by replacing the summation unit in Figure

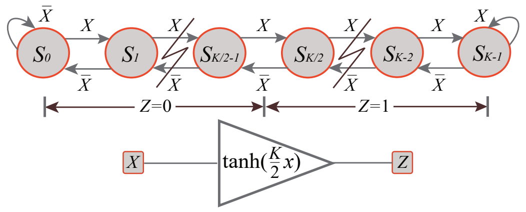

Figure 6: Stochastic hyperbolic tangent.

Table 1: Inaccuracies of OR Gate-Based Inner Product Block

Input Size	8	16	24	32
Unipolar inputs	0.16	0.33	0.47	0.66
Bipolar inputs	1.04	1.39	1.54	1.70

3 (a) with various adder implementations shown in Figure 5. Their pros and cons are carefully analyzed for the subsequent step of developing feature extraction blocks in SC-DCNNs.

OR Gate-Based Inner Product Block Design. The idea of utilizing OR gate to perform addition is straightforward. For instance, $\frac{3}{8}+\frac{4}{8}$ can be performed by "00100101 OR 11001010", which generates "11101111" ($\frac{7}{8}$). However, the first input bit-stream can also be "10011000", which makes the output of OR gate as "11011010" ($\frac{5}{8}$) and results in inaccuracy. The accuracy loss comes from the fact that "logic 1 OR logic 1" only generates a single logic 1 without a carry bit. To reduce the accuracy loss, the input streams should be pre-scaled to ensure that there are only very few 1's in the bit-streams. For the unipolar format bit-streams, the scaling can be easily performed by dividing the original number by a scaling factor. Nevertheless, in the scenario of bipolar encoding format, there are about 50% 1's in the bit-stream when the original value is close to 0, which renders the scaling ineffective in reducing the number of 1's in the bit-stream.

Table 1 displays the average inaccuracies of OR gate-based inner product block with different input sizes, in which the bit-stream length is fixed at 1024 and all average inaccuracy values are obtained with the most suitable pre-scaling. The experimental results suggest that the accuracy of unipolar calculations may be acceptable, but the accuracy is too low for bipolar calculations and becomes even worse with the increase of input size. Since it is almost impossible to have only positive input values and weights, the OR gate-based inner product block is not appropriate in SC-DCNNs.

MUX-Based Inner Product Block Design. According to [22], an n-to-1 MUX can sum all inputs together and generate an output with a scaling down factor $\frac{1}{n}$. Since only one bit is selected among all inputs to that MUX at one time, the probability of each input to be selected is $\frac{1}{n}$. The selection signal is controlled by a randomly generated natural number between 1 and n. Taking Figure 3 (a) as an example, the output of the summation unit (MUX) is $\frac{1}{n}(x_0w_0+...+x_{n-1}w_{n-1})$.

As displayed in Table 2, the average inaccuracies of the MUX-based inner product block are measured with different input sizes and bit-stream lengths. The accuracy loss of MUX-based block mainly comes from the fact that only one input is selected at one time, and all the other inputs are not used.

Table 2: Inaccuracies of MUX-Based Inner Product Block

Input size	Bit stream length			
	512	1024	1536	2048
8	0.27	0.19	0.16	0.14
16	0.54	0.39	0.31	0.28
32	1.18	0.77	0.64	0.56

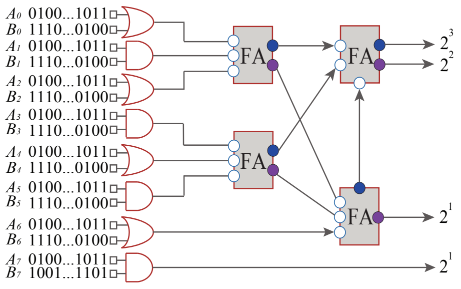

Figure 7: 16-bit Approximate Parallel Counter.

The increasing input size causes accuracy reduction because more bits are dropped, but good enough accuracy can still be obtained by increasing the bit-stream length.

Two-Line Representation-Based Inner Product Block. As mentioned above, the MUX-based adder is a down-scaled adder and the down-scaling is a main source of accuracy loss. Accordingly, reference [31] proposed a two-line representation-based SC scheme that can be used to construct a non-scaled adder. Figure 5 (d) illustrates the structure of a two-line representation-based adder. Since A_i, B_i, and C_i are bounded as the element of $\{-1,0,1\}$, a carry bit may be missed. Therefore, a three-state counter is used here to store the positive or negative carry bit.

However, there are two limitations in utilizing the two-line representation-based inner product block in hardware DCNNs: (i) because an inner product block generally has more than two inputs, the overflow problem often occurs in the two-line representation-based inner product calculation due to its non-scaling characteristics, which incurs significant accuracy loss, and (ii) the area overhead is too high compared with other inner product implementation methods.

APC-Based Inner Product Block. The structure of an 16-bit APC is shown in Figure 7. A_0-A_7 and B_0-B_7 are the outputs of XNOR gates, i.e., the products of inputs x_i's and weights w_i's. Suppose the number of inputs is n and the length of a bit-stream is m, then the products of x_i's and w_i's can be represented by a bit-matrix of size $n\times m$. The function of the APC is to count the number of ones in one column and represent the result in the binary format, thereby the number of outputs is $\log_2 n$. Taking a 16-bit APC as an example, the output should be 4-bit to represent a number between 0 - 16. However, please notice that the weight of the least significant bit is 2^1 rather than 2^0 to represent 16. Therefore, the output of the APC is a bit-matrix with size of $\log_2 n\times m$.

From Table 3, we know that the APC (approximate parallel counter) only results in less than 1% accuracy degradation when compared with the conventional accumulative parallel counter, but it can achieve about 40% reduction of gate count

Table 3: Inaccuracies of the APC-Based Compared with the Conventional Parallel Counter-Based Inner Product Blocks

Input size	Bit stream length			
	128	256	384	512
16	1.01%	0.87%	0.88%	0.84%
32	0.70%	0.61%	0.58%	0.57%
64	0.49%	0.44%	0.44%	0.42%

[29]. This observation illustrates the significant advantage for the goal of implementing an efficient inner product block in terms of power, energy, and hardware resource.

4.2. Pooling Block Designs

Pooling (or down-sampling) operations are performed by pooling function blocks in DCNNs to significantly reduce (i) inter-layer connections and (ii) the number of parameters and computations in the network, meanwhile maintaining the translation invariance of the extracted features [27]. Average pooling and max pooling are two widely used pooling strategies. Average pooling is straightforward to implement in the SC scheme, while max pooling, which exhibits higher performance in general, requires high hardware resources. In order to overcome this limitation, we propose and investigate a novel hardware-oriented max pooling with high performance and high compatibility to SC scheme. Details are discussed next.

Average Pooling Block Design. Figure 3 (b) shows how the feature map is average pooled with 2×2 filters. Since average pooling is used to calculate the mean value of entries in a small matrix, the inherent down-scaling property of the MUX can be utilized. Therefore, the average pooling can be performed by the structure as shown in Figure 5-(b) with a very small hardware footprint.

Hardware-Oriented Max Pooling Block Design. The max pooling operation has recently shown higher performance in practice when compared with the average pooling operation [27]. However, in the stochastic domain, we can find out the bit-stream with the maximum value among four candidates only after counting the total number of 1's through the whole bit-streams, which will inevitably result in a long latency and notable energy consumption.

To overcome this limitation, we propose a novel SC-based hardware-oriented max pooling scheme. The idea behind this design is that once a set of bit-streams are sliced into segments, the globally largest bit-stream (among the four candidates) has the highest probability to be the locally largest one in each set of bit-stream segments. This is because all 1's are randomly distributed in the stochastic bit-streams. Accordingly, all input bit-streams of the hardware-oriented max pooling block are sliced into segments with a fixed length c, e.g., 16 bits, and one segment is selected from each set (one set has four segments) to be sent to the output. To determine the selected segment in a set, all segments in a set are counted on the number of 1's in parallel, and the maximum counted result is utilized to determine the next c-bit segment that to be sent to the output of the pooling block. In other words, the currently selected c-bit segment is determined by the counting results of the previous set. Please notice that the c-bit segment from the first set of bit-stream segments is randomly chosen to reduce the latency. This strategy will incur zero extra latency and will only cause a negligible accuracy loss when c is a sufficiently small value compared with the bit-stream length.

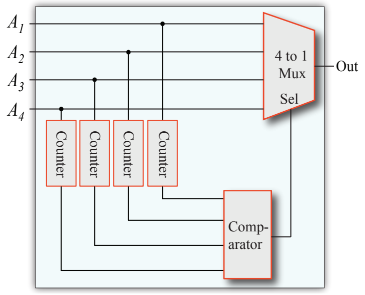

Figure 8: The Proposed Hardware-Oriented Max Pooling.

Table 4: Relative Result Deviation of Hardware-Oriented Max Pooling Block Compared with Software-Based Max Pooling

Input size	Bit-stream length			
	128	256	384	512
4	0.127	0.081	0.066	0.059
9	0.147	0.099	0.086	0.074
16	0.166	0.108	0.097	0.086

Figure 8 illustrates the structure of the hardware-oriented max pooling block, where the output from max_output approximately equals to the largest bit-stream. The four input bit-streams sent to the multiplexer are also connected to four counters, and the outputs of the counters are connected to a comparator to determine the largest segment. Then the output of the comparator is used to control the selection of the four-to-one MUX. Suppose in the previous set of segments, the second line is the largest, then MUX will output the second bit-stream for the current c-bit segment.

Table 4 shows the result deviations of the hardware-oriented max pooling design compared with the software-based max pooling implementation. The length of a bit-stream segment is 16. In general, the proposed pooling block can provide a sufficiently accurate result even with a large input size.

4.3. Activation Function Block Designs

Stanh. Reference [22] proposed a K-state FSM-based design (i.e., Stanh) in the SC domain for implementing the tanh function, and also describes the relationship between Stanh and tanh as $Stanh(K,x) \cong tanh(\frac{K}{2}x)$. When the input stream x is distributed in the range [-1, 1], i.e., $\frac{K}{2}x$ is distributed in the range $[-\frac{K}{2}, \frac{K}{2}]$, this equation works well, and higher accuracy can be achieved with the increased state number K.

However, $Stanh$ cannot be applied directly in our framework for three reasons. First, as shown in Figure 9 and Table 5 (with bit-stream length fixed at 8192), when the input variable of Stanh (i.e., $\frac{K}{2}x$) is distributed in the range of [-1, 1], the inaccuracy is quite notable and is not suppressed with the increasing of K. Second, the equation works well when x is precisely

Table 5: The Relationship Between State Number and Relative Inaccuracy of Stanh

State Number	8	10	12	14	16	18	20
Relative Inaccuracy (%)	10.06	8.27	7.43	7.36	7.51	8.07	8.55

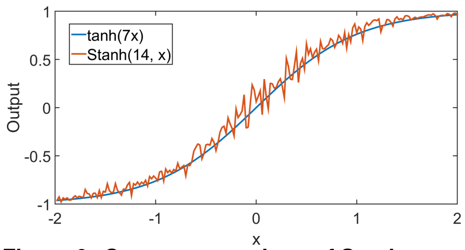

Figure 9: Output comparison of Stanh vs tanh.

represented. However, when the bit-stream is not impractically long (less than 2^{16} according to our experiments), the equation should be adjusted with a consideration of bit-stream length. Third, in the practice of implementing SC-DCNNs, we usually need to proactively down-scale the inputs since a bipolar stochastic number cannot reach beyond the range [-1, 1]. Besides, the stochastic number may be sometimes passively down-scaled by certain components, like a MUX-based adder or an average pooling block. A *scaling-back* process is thus imperative to obtain an accurate result. Based on the above reasons, the design of Stanh needs to be optimized together with other function blocks to achieve high accuracy for different bit-stream lengths and meanwhile provide a scaling-back function, with more details in Section 4.4.

Btanh. Btanh is specifically designed for the APC-based adder to perform a scaled hyperbolic tangent function. Instead of using FSM, a saturated up/down counter is used here to convert the binary outputs of the APC-based adder back to a bit-stream. The implementation details and the determination of state number can be found in reference [20].

4.4. Design & Optimization for Feature Extraction Blocks

In this section, we propose and investigate the optimal designs of feature extraction blocks, which are in charge of extracting features from input feature maps. Based on the above analysis results, the MUX-based and APC-based inner product/convolution blocks, average pooling and hardware-oriented max pooling blocks, Stanh and Btanh blocks are selected as candidates for constructing feature extraction blocks, which are in charge of extracting features from input feature maps in SC-DCNNs (as shown in Figure 10). Instead of simply composing the basic function blocks, a series of joint optimizations are performed on each type of feature extraction block to achieve the optimal performance. In the SC domain, factors like input size, bit-stream length, and the inaccuracy introduced by the previous connected block can make a significant difference on the overall performance of a feature extraction block. Therefore, separate optimizations on each individual basic function block cannot guarantee to achieve the best performance for the entire feature extraction block. For example, the most important advantage of the APC-based

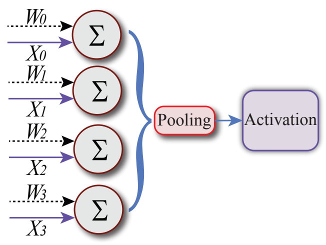

Figure 10: The structure of a feature extraction block.

inner product block is high accuracy and thus the bit-stream length can be reduced; and the most important advantage of MUX-based inner product block is low hardware footprint and the accuracy can be improved by increasing the bit-stream length. Accordingly, in this work, we design feature extraction blocks with a consideration of fully making use of the advantages of each of the building blocks.

For the convenience of following discussions, we define that MUX/APC represents the MUX-based or APC-based inner product/convolution blocks; Avg/Max represents the average or hardware-oriented max pooling blocks; Stanh/Btanh represents the corresponding activation function blocks. For instance, MUX-Avg-Stanh means that four MUX-based inner product blocks, one average pooling block, and one Stanh activation function block are cascade-connected.

MUX-Avg-Stanh. As mentioned in Section 4.3, when Stanh is utilized, the state number needs to be carefully selected with a comprehensive consideration of the scaling factor, bit-stream length, and accuracy requirement. Below is the empirical equation that is extracted from our comprehensive experiments to obtain the approximately optimal state number K to achieve a high accuracy:

$$K = f(L,N) \approx 2 \times \log_2 N + \frac{\log_2 L \times N}{\alpha \times \log_2 N}, \qquad (1)$$

where the nearest even number of the result calculated by the above equation is assigned to K, N is the input size, L is the bit-stream length, and empirical parameter $\alpha = 33.27$.

MUX-Max-Stanh. The hardware-oriented max pooling block shown in Figure 8 in most cases generates an output that is slightly less than the maximum value. In this design of feature extraction block, the inner products are all scaled down by a factor of n (n is the input size), and the subsequent scaling back function of Stanh will enlarge the inaccuracy, especially when the positive/negative sign of the selected maximum inner product value is changed. For instance, 505/1000 is a positive number, and 1% under-counting will lead the output of the hardware-oriented max pooling unit to be 495/1000, which is a negative number. Thereafter, the obtained output of Stanh may be -0.5, but the expected result should be 0.5. Therefore, the bit-stream has to be long enough to diminish the impact of under-counting, and the Stanh needs to be re-designed to fit the correct (expected) results. As shown in Figure 11, the redesigned FSM for Stanh will output zero when the current state is at the left 1/5 of the diagram, otherwise output a one. The optimal state number K is calculated through the following

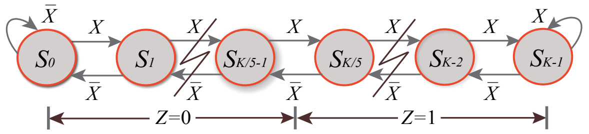

Figure 11: Structure of optimized Stanh for MUX-Max-Stanh.

empirical equation derived from experiments:

$$K = f(L,N) \approx 2 \times (\log_2 N + \log_2 L) - \frac{\alpha}{\log_2 N} - \frac{\beta}{\log_5 L}, \qquad (2)$$

where the nearest even number of the result calculated by the above equation is assigned to K, N is the input size, L is the bit-stream length, $\alpha = 37$, and empirical parameter $\beta = 16.5$.

APC-Avg-Btanh. When the APC is used to construct the inner product block, conventional arithmetic calculation components, such as full adders and dividers, can be utilized to perform the averaging calculation, because the output of APC-based inner product block is a binary number. Since the design of Btanh initially aims at directly connecting to the output of APC, and an average pooling block is now inserted between APC and Btanh, the original formula proposed in [20] for calculating the optimal state number of Btanh needs to be reformulated as:

$$K = f(N) \approx \frac{N}{2}, \qquad (3)$$

from our experiments. In this equation N is the input size, and the nearest even number to $\frac{N}{2}$ is assigned to K.

APC-Max-Btanh. Although the output of APC-based inner product block is a binary number, the conventional binary comparator cannot be directly utilized to perform max pooling. This is because the output sequence of APC-based inner product block is still a stochastic bit-stream. If the maximum binary number is selected at each time, the pooling output is always greater than the actual maximum inner product result. Instead, the proposed hardware-oriented max pooling design should be utilized here, and the counters should be replaced by accumulators for accumulating the binary numbers. Benefited from the high accuracy provided by accumulators in selecting the maximum inner product result, the original Btanh design presented in [20] can be directly utilized without adjustment.

5. Weight Storage Scheme and Optimization

As introduced in Section 4, the main computing task of an inner product block is to calculate the inner products of x_i's and w_i's. x_i's are input by customers, but w_i's are weights obtained by training using software and should be stored in the hardware-based DCNNs. *Static random access memory (SRAM)* is the most appropriate circuit structure for weight storage due to its high reliability, high speed, and small area. And specifically optimized SRAM placement schemes and weight storage methods are imperative for further reductions of area and power (energy) consumptions. In this section, we present optimization techniques including efficient filter-aware SRAM sharing, weight storage method, and layer-wise weight storage optimizations.

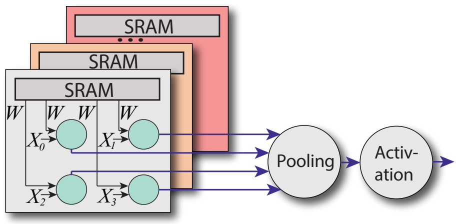

Figure 12: Filter-Aware SRAM Sharing Scheme.

5.1. Efficient Filter-Aware SRAM Sharing Scheme

Since all receptive fields of a feature map share one filter (a matrix of weights), all weights functionally can be separated into filter-based blocks and each block of weights are shared by all inner product/convolution blocks using the corresponding filter. Inspired by this fact, we propose an efficient filter-aware SRAM sharing scheme, with structure illustrated in Figure 12. The scheme divides the whole SRAM into small blocks to mimic filters. Besides, all inner product blocks can also be separated into feature map-based groups, and each group of inner product blocks takes charge of extracting one feature map. Therefore, a local SRAM block is shared by all the inner product blocks of the corresponding group, and then the weights of the corresponding filter are stored into the local SRAM block of this group. This scheme can significantly reduce the routing overhead and wire delay.

5.2. Weight Storage Method

Except for the reduction on routing overhead, the size of SRAM blocks can also be reduced by trading off accuracy and hardware resources. The trading off is implemented by eliminating certain least significant bits of a weight value to reduce the SRAM size. Accordingly, we present a weight storage method for significantly reducing the SRAM size with little accuracy loss.

Baseline: High Precision Weight Storage. In general, DCNN will be trained with single floating point precision. Thus on hardware, up to 64-bit SRAM is needed for storing one weight value in the fixed point format to maintain its original high precision. This scheme can provide high accuracy as there is almost no information loss of weights. However, it also brings about high hardware consumptions in that the size of SRAM and its related read/write circuits is increasing with the increasing of precision of the stored weight values.

Low Precision Weight Storage Method. According to our software-level experiments, many least significant bits far from the decimal point only have a very limited impact on the overall network accuracy, thus the number of bits for weight representation in the SRAM block can be significantly reduced. We propose a mapping equation that converts a weight in the real number format to the binary number stored in SRAM to eliminate the proper numbers of least significant bits. Suppose the weight value is x, and the number of bits to store a weight value in SRAM is w (which is defined as the *precision* of the represented weight value in this paper), then the binary

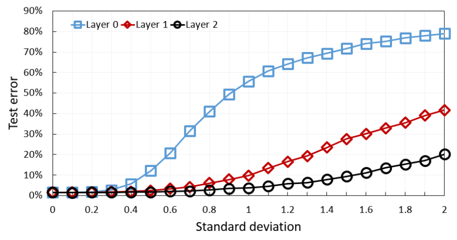

Figure 13: The impact of inaccuracies at each layer on the overall SC-DCNN network accuracy.

number to be stored for representing x is:

$$y = \frac{Int(\frac{x+1}{2} \times 2^w)}{2^w}, \tag{4}$$

where $Int()$ means only keeping the integer part. Figure 14 illustrates the network error rates when the reductions of weights' precision are performed at a single layer or all layers. The precision loss of weights at Layer0 (consisting of a convolutional layer and pooling layer) has the least impact, while the precision loss of weights at Layer2 (a fully connected layer) has the most significant impact. The reason is that Layer2 is the fully connected layer, which has the largest number of weights. On the other hand, when w is set equal to or greater than seven, the network error rates are low enough and almost not decreasing with the further increasing of precision. Therefore, our proposed weight storage method can significantly reduce the size of SRAMs and their read/write circuits through decreasing the precision. The area savings achieved by this method based on estimations from CACTI 5.3 [33] is 10.3×.

5.3. Layer-wise Weight Storage Optimization

As shown in Figure 13, the inaccuracies at different layers have different impacts on the overall accuracy of the network. Layer0 is the most sensitive to inaccuracies, and Layer2 is the least sensitive to inaccuracies. Interestingly, this is the opposite to the sensitivity to precision as shown in Figure 14. Combining the observations from Figure 13 and Figure 14, we propose a layer-wise weight storage scheme, which sets different weight precisions at different layers. More specifically, we set the weights at Layer2 to be a relatively low precision but higher than four, while setting the weights of the previous layers with a relatively high precision to compensate the accuracy loss, so as to maintain the overall high network accuracy. This method is effective to obtain savings in SRAM area and power (energy) consumptions because Layer2 has the most number of weights compared with the previous layers. For instance, when we set weights as 7-7-6 at the three layers of LeNet5, the network error rate is 1.65%, which has only 0.12% accuracy degradation compared with the error rate obtained on software. However, 12× improvements on area and 11.9× improvements on power consumptions are achieved for the weight representations (from CACTI 5.3 estimations), comparing with the baseline without any reduction in weight representation bits.

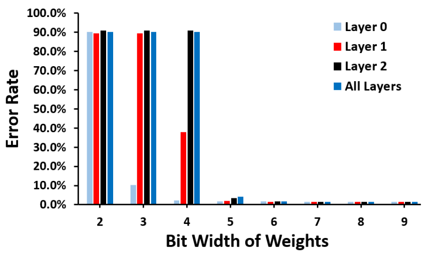

Figure 14: The impact of precision of weights at different layers on the overall SC-DCNN network accuracy.

6. Overall SC-DCNN Optimizations and Results

In this section, we first present optimizations of feature extraction blocks along with comparison results with respect to accuracy, area/hardware footprint, power (energy) consumption, etc. Based on the results, we perform thorough optimizations on the overall SC-DCNN to construct LeNet5 structure, which is one of the most well-known large-scale deep DCNN structure, to minimize area and power (energy) consumption while maintaining a high network accuracy level. Comprehensive comparison results are provided among SC-DCNN designs (with different target network accuracy levels) and with existing hardware platforms. The hardware performance of the SC-DCNNs regarding area, path delay, power and energy consumptions are obtained by: (i) synthesizing with the 45nm Nangate Open Cell Library [34] using Synopsys Design Compiler, (ii) estimating using CACTI 5.3 [33] for the SRAM blocks. Key peripheral circuitry in the SC domain, e.g., the random number generators, are developed using the design in [35] and synthesized using Synopsys Design Compiler.

6.1. Optimization Results on Feature Extraction Blocks

We present optimization results of feature extraction blocks under different structures, input sizes, and bit-stream lengths on accuracy, area/hardware footprint, power (energy) consumption, etc. Figure 15 illustrates the accuracy performance of four types of feature extraction blocks: MUX-Avg-Stanh, MUX-Max-Stanh, APC-Avg-Btanh, and APC-Max-Btanh. The horizontal axis represents the input size that increases logarithmically from 16 (2^4) to 256 (2^8). The vertical axis represents the hardware inaccuracies of feature extraction blocks. Three bit-stream lengths are tested and their impacts are shown in the figure. Figure 16 illustrates the comparisons among four feature extraction blocks with respect to area, path delay, power, and energy consumptions, and the horizontal axis represents the input size that increases logarithmically from 16 (2^4) to 256 (2^8). The bit-stream length is fixed at 1024.

MUX-Avg-Stanh. From Figure 15-(a), we know that it has the worst accuracy performance among the four structures. Because MUX-based adder as mentioned in Section 4 is a down-scaling adder and incurs inaccuracy because of information loss. Besides, average pooling is performed with MUXes, thus the inner products are further down-scaled and more inaccuracies are incurred. As a result, this structure of feature

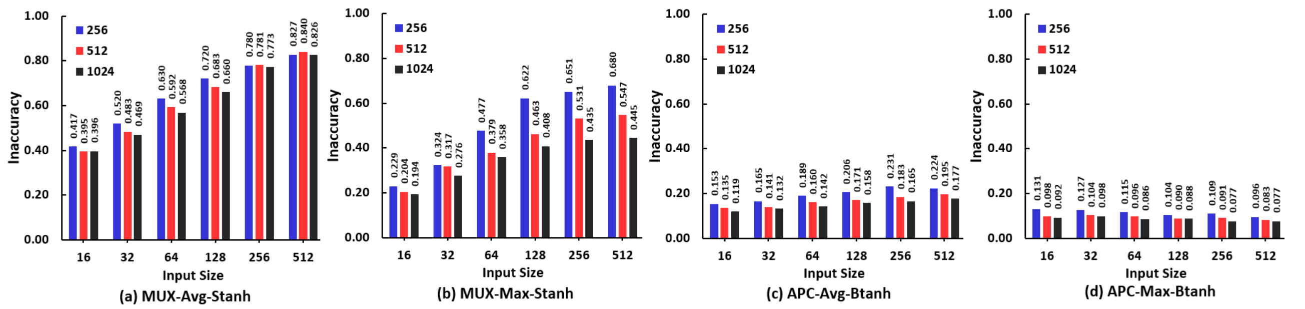

Figure 15: Input size versus absolute inaccuracy for (a) MUX-Avg-Stanh, (b) MUX-Max-Stanh, (c) APC-Avg-Btanh, and (d) APC-Max-Btanh with different bit stream lengths.

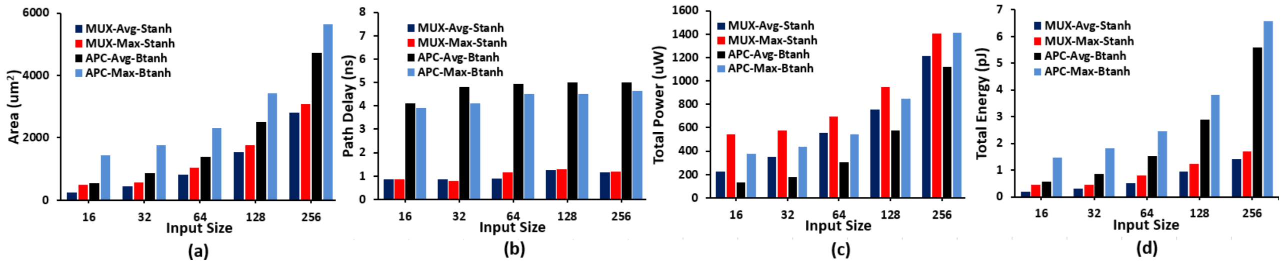

Figure 16: Input size versus (a) area, (b) path delay, (c) total power, and (d) total energy for four different designs of feature extraction blocks.

extraction block is only appropriate for dealing with receptive fields with a small size. On the other hand, it also possesses advantages in that it is the most area and energy efficient design with the smallest path delay. Hence, it is appropriate for scenarios with tight limitations on area and delay.

MUX-Max-Stanh. Figure 15-(b) shows that it has a better performance in terms of accuracy when compared with the MUX-Avg-Stanh. The reason is that the mean of four numbers is generally closer to zero than the maximum value of the four numbers. As mentioned in Section 4, minor inaccuracies on the stochastic numbers near zero can cause significant inaccuracies on the outputs of feature extraction blocks. Thus the structures with hardware-oriented pooling are more resilient than the structures with average pooling. In addition, the accuracy can be significantly improved by increasing the bit-stream length, thus this structure can be applied for dealing with the receptive fields with both small and large sizes. With respect to area, path delay, and energy, its performance is just second to the MUX-Avg-Stanh and close enough. Despite its relatively high power consumption, the power can be remarkably reduced by trading-off with the path delay.

APC-Avg-Btanh. Figure 15-(c) and 15-(d) illustrate the hardware inaccuracies of APC-based feature extraction blocks. The results imply that they significantly outperform the MUX-based feature extraction blocks in terms of accuracy, since the APC-based inner product blocks maintain most information of inner products and thus generate results with high accuracy, which is the drawback of the MUX-based inner product blocks. On the other hand, APC-based feature extraction blocks consume more hardware resources and result in much longer path delays as well as energy consumptions. The long path delay is also the reason that their power consumptions are lower than MUX-based designs. Therefore, the APC-Avg-Btanh is appropriate for DCNN implementations that have a tight specification on the accuracy performance and have a relative loose hardware resource constraint.

APC-Max-Btanh. Figure 15-(d) indicates that this feature extraction block design has the best accuracy performance since: First, it is an APC-based design. Second, the average pooling in the APC-Avg-Btanh causes more information loss than the proposed hardware-oriented max pooling. To be more specific, the fractional part of the number after average pooling is dropped, e.g., the mean of (2, 3, 4, 5) is 3.5, but it will be represented as 3 in binary format, thus some information is lost during the average pooling. Generally, the increase of input size will incur significant inaccuracies except for APC-Max-Btanh. The reason that APC-Max-Btanh performs better with more inputs is: more inputs will make the four inner products sent to the pooling function block more distinct from one another, i.e., more inputs result in higher accuracy in selecting the maximum value. The drawbacks of APC-Max-Btanh are also distinct. It has the highest area and energy consumptions, and its path delay is just second to and very close to the APC-Avg-Btanh. Besides, its power consumption is just second to and close to the MUX-Max-Stanh. Accordingly, this design is appropriate for the applications that have a very tight requirement on the accuracy performance.

6.2. Overall Optimizations and Results on SC-DCNNs

Based on the results on feature extraction blocks, we perform thorough optimizations on the overall SC-DCNN to construct the LeNet 5 DCNN structure, to minimize area and power (energy) consupmtion while maintaining a high network accuracy. The four types of feature extraction blocks, the basic function

Table 6: Comparison among Various SC-DCNN Designs Implementing LeNet 5

No.	Pooling	Bit Stream	Configuration Layer 0	Layer 1	Layer 2	Performance Inaccuracy (%)	Area (mm^2)	Power (W)	Delay (ns)	Energy (μJ)
1	Max	1024	MUX	MUX	APC	2.64	19.1	1.74	5120	8.9
2			MUX	APC	APC	2.23	22.9	2.13	5120	10.9
3		512	APC	MUX	APC	1.91	32.7	3.14	2560	8.0
4			APC	APC	APC	1.68	36.4	3.53	2560	9.0
5		256	APC	MUX	APC	2.13	32.7	3.14	1280	4.0
6			APC	APC	APC	1.74	36.4	3.53	1280	4.5
7	Average	1024	MUX	APC	APC	3.06	17.0	1.53	5120	7.8
8			APC	APC	APC	2.58	22.1	2.14	5120	11.0
9		512	MUX	APC	APC	3.16	17.0	1.53	2560	3.9
10			APC	APC	APC	2.65	22.1	2.14	2560	5.5
11		256	MUX	APC	APC	3.36	17.0	1.53	1280	2.0
12			APC	APC	APC	2.76	22.1	2.14	1280	2.7

Table 7: Comparison with Existing Hardware Platforms

Platform	Dataset	Network Type	Year	Platform Type	Area (mm^2)	Power (W)	Accuracy (%)	Throughput (Images/s)	Area Efficiency (Images/s/mm^2)	Energy Efficiency (Images/J)
SC-DCNN (No.6)	MNIST	CNN	2016	ASIC	36.4	3.53	98.26	781250	21439	221287
SC-DCNN (No.11)			2016	ASIC	17.0	1.53	96.64	781250	45946	510734
2×Intel Xeon W5580			2009	CPU	263	156	98.46	656	2.5	4.2
Nvidia Tesla C2075			2011	GPU	520	202.5	98.46	2333	4.5	3.2
Minitaur [36]		ANN[1]	2014	FPGA	N/A	≤1.5	92.00	4880	N/A	≥3253
SpiNNaker [37]		DBN[2]	2015	ARM	N/A	0.3	95.00	50	N/A	166.7
TrueNorth [38, 39]		SNN[3]	2015	ASIC	430	0.18	99.42	1000	2.3	9259
DaDianNao [17]	ImageNet	CNN	2014	ASIC	67.7	15.97	N/A	147938	2185	9263
EIE-64PE [18]		CNN layer	2016	ASIC	40.8	0.59	N/A	81967	2009	138927

blocks, and the weight storage schemes are carefully compared and selected in the procedure. The (max pooling-based or average pooling-based) LeNet 5 is a widely-used DCNN structure [40] with a configuration of 784-11520-2880-3200-800-500-10. The SC-DCNNs are evaluated with the MNIST handwritten digit image dataset [41], which consists of 60,000 training data and 10,000 testing data.

The baseline error rates of the max pooling-based and average pooling-based LeNet5 DCNNs using software implementations are 1.53% and 2.24%, respectively. In the optimization procedure, we set a threshold on the error rate difference as 1.5%, i.e., the network accuracy degradation of the SC-DCNNs cannot exceed 1.5% compared with the error rates when tested using software. We set the maximum bit-stream length as 1024 to avoid over-long delays. In the optimization procedure, for the configurations that achieve the target network accuracy, the bit-stream length is reduced by half in order to reduce energy consumptions. Configurations are removed if they fail to meet the network accuracy goal. The process is iterated until no configuration is left.

Table 6 displays some selected typical configurations and their comparison results (including the consumptions of SRAMs and random number generators). Configurations No.1-6 are max pooling-based SC-DCNNs, and No.7-12 are average pooling-based SC-DCNNs. It can be observed that the configurations involving more MUX-based feature extraction blocks achieve less hardware footprint, and those involving more APC-based feature extraction blocks achieve higher network accuracy. For the max pooling-based configurations, No.1 is the most area efficient as well as power efficient configuration, and No.5 is the most energy efficient configuration. With regard to the average pooling-based configurations, No.7, 9, 11 are the most area efficient and power efficient configurations, and No.11 is the most energy efficient configuration.

Table 7 displays the comparison of our proposed SC-DCNNs with software implementations using Intel Xeon Dual-Core W5580 or Nvidia Tesla C2075 GPU as well as existing hardware platforms. For example, EIE [18]'s performance was evaluated on a fully connected layer of AlexNet [23]; the state-of-the-art platform DaDianNao [17] proposed an ASIC "node" that could be connected in parallel to implement a large-scale DCNN; and other hardware platforms implement different types of hardware neural networks such as spiking neural network or deep-belief network. Configurations No.6 and No.11 are selected for comparison, since No.6 is the most accurate max pooling-based configuration and No.11 is the most energy efficient average pooling-based configuration.

When comparing with software implementation on CPU server or GPU, the proposed SC-DCNNs are much more area efficient, with improvements up to 30.6× by comparing SC-DCNN (No.11) with Nvidia Tesla C2075. Besides, our proposed SC-DCNNs also have outstanding performance in terms of throughput, area efficiency, and energy efficiency. The proposed SC-DCNN (No.11) achieves 15625× throughput improvements and 159604× energy efficiency improvements, comparing with Nvidia Tesla C2075. Regarding the reference hardware platforms, although not directly comparable to some extent due to the difference in neural network types and structures, the proposed SC-DCNN achieves the lowest hardware footprint, the highest throughput, and the highest energy efficiency. Despite the fact that the power performance of SC-DCNNs is not the best, it is still comparable with other

[1]ANN: Artificial Neural Network; [2]DBN: Deep Belief Network; [3]SNN: Spiking Neural Network

ASIC platforms like DaDianDao, EIE and Minitaur.

7. Conclusion

In this paper, a comprehensive SC-DCNN architecture is explored to achieve high power (energy) efficiency and low hardware footprint. First, various function blocks involving inner product calculations, pooling operations, and activation functions are investigated. Then four types of feature extraction blocks, which are constructed with the carefully selected function blocks, are proposed and jointly optimized to achieve the optimal accuracy. And three weight storage optimization schemes are investigated for reducing the area and power (energy) consumptions of SRAM. Experimental results demonstrate that our proposed SC-DCNN achieves ultra-low hardware footprint and low energy consumptions. It achieves the throughput of 781250 images/s, area efficiency of 45946 images/s/mm^2, and energy efficiency of 510734 images/J.

References

[1] L. Deng and D. Yu, "Deep learning," *Signal Processing*, vol. 7, pp. 3–4, 2014.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 160–167.

[5] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, "Deep convolutional neural networks for lvcsr," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 8614–8618.

[6] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[7] E. László, P. Szolgay, and Z. Nagy, "Analysis of a gpu based cnn implementation," in *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*. IEEE, 2012, pp. 1–5.

[8] G. V. STOICA, R. DOGARU, and C. Stoica, "High performance cuda based cnn image processor," 2015.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[10] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[11] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," *arXiv preprint arXiv:1606.05487*, 2016.

[12] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A cgra-based approach for accelerating convolutional neural networks," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 2015, pp. 73–80.

[13] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[14] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[15] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 10, pp. 1864–1878, 2014.

[16] L. Xia, B. Li, T. Tang, P. Gu, X. Yin, W. Huangfu, P.-Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "Mnsim: Simulation platform for memristor-based neuromorphic computing system," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 469–474.

[17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[19] Y. Ji, F. Ran, C. Ma, and D. J. Lilja, "A hardware implementation of a radial basis function neural network using stochastic logic," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 880–883.

[20] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 124.

[21] B. R. Gaines, "Stochastic computing," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 149–156.

[22] B. D. Brown and H. C. Card, "Stochastic neural computation. i. computational elements," *IEEE Transactions on computers*, vol. 50, no. 9, pp. 891–905, 2001.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[25] S. Sato, K. Nemoto, S. Akimoto, M. Kinjo, and K. Nakajima, "Implementation of a new neurochip using stochastic logic," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 1122–1127, 2003.

[26] "Convolutional neural networks (lenet)," 2016. [Online]. Available: http://deeplearning.net/tutorial/lenet.html#motivation

[27] "Stanford cs class, cs231n: Convolutional neural networks for visual recognition," 2016. [Online]. Available: http://cs231n.github.io/convolutional-networks/

[28] B. Yuan, C. Zhang, and Z. Wang, "Design space exploration for hardware-efficient stochastic computing: A case study on discrete cosine transformation," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016, pp. 6555–6559.

[29] K. Kim, J. Lee, and K. Choi, "Approximate de-randomizer for stochastic circuits," *Proc. ISOCC*, 2015.

[30] B. Parhami and C.-H. Yeh, "Accumulative parallel counters," in *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, vol. 2. IEEE, 1995, pp. 966–970.

[31] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 1. IEEE, 2000, pp. 599–602.

[32] D. Larkin, A. Kinane, V. Muresan, and N. O'Connor, "An efficient hardware architecture for a neural network activation function generator," in *International Symposium on Neural Networks*. Springer, 2006, pp. 1319–1327.

[33] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.3," *HP Laboratories, Palo Alto, CA*, 2008.

[34] Nangate 45nm Open Library, Nangate Inc., 2009. [Online]. Available: http://www.nangate.com/

[35] K. Kim, J. Lee, and K. Choi, "An energy-efficient random number generator for stochastic circuits," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 256–261.

[36] D. Neil and S.-C. Liu, "Minitaur, an event-driven fpga-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, 2014.

[37] E. Stromatias, D. Neil, F. Galluppi, M. Pfeiffer, S.-C. Liu, and S. Furber, "Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on spinnaker," in *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2015, pp. 1–8.

[38] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Advances in Neural Information Processing Systems*, 2015, pp. 1117–1125.

[39] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L. McKinstry, T. Melano, D. R. Barch, C. di Nolfo, P. Datta, A. Amir, B. Taba, M. D. Flickner, and D. S. Modha, "Convolutional networks for fast, energy-efficient neuromorphic computing," *CoRR*, vol. abs/1603.08270, 2016. [Online]. Available: http://arxiv.org/abs/1603.08270

[40] Y. LeCun, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[41] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.