A 2.56-mm^2 718GOPS Configurable Spiking Convolutional Sparse Coding Accelerator in 40-nm CMOS

Chester Liu, *Student Member, IEEE*, Sung-Gun Cho, *Student Member, IEEE*, and Zhengya Zhang, *Senior Member, IEEE*

***Abstract*— A configurable neuroinspired inference accelerator is designed as an array of neurons, each operating in an independent clock domain. The accelerator implements a recurrent network using a novel sparse convolution for feedforward operations and sparse spike-driven reconstruction for feedback operations. The proposed sparse convolution efficiently skips zero-patches, and can be made to support practically any image and kernel size. A globally asynchronous locally synchronous architecture enables scalable design and load balancing to achieve 22% reduction in power. Fabricated in 40-nm CMOS, the 2.56-mm^2 inference accelerator integrates 48 neurons, a hub, and an OpenRISC processor. The chip achieves 718GOPS at 380 MHz, and demonstrates applications in feature extraction from images and depth extraction from stereo images.**

***Index Terms*— Configurable convolution, globally asynchronous locally synchronous (GALS) architecture, recurrent neural network (RNN), sparse coding, sparsity optimization.**

I. Introduction

NEUROINSPIRED sparse coding algorithms have been applied to various types of sensory inputs, including image [1]–[3], audio [4], [5], and video [6]–[8], for feature extraction in a wide range of applications such as denoising [9], [10], super-resolution [11], [12], object recognition [13], [14], and face recognition [15].

The classic sparse coding is mapped to a fully connected recurrent neural network (RNN) [16], and application-specific integrated circuits (ASICs) have been designed to achieve impressive performance and efficiency [14], [17]. However, the ASIC designs have been limited to relatively small feature sizes, e.g., 4 × 4, and small input image patch sizes, e.g., up to 16 × 16. The designs are not scalable to efficiently support applications that require larger feature sizes or larger input patch sizes. Convolutional sparse coding was therefore introduced to improve sparse coding's scalability by exploiting the shift-invariant characteristic commonly found in sensory inputs [18], [19].

Although convolutional sparse coding is scalable in theory, a number of important challenges still remain. First, sparse coding is meant to be a universal encoding algorithm that is input agnostic, but until now, it is unclear whether a universal, or programmable, hardware can be made to extend the applicability of sparse coding to more than one application. Second, convolutional sparse coding is implemented in a convolutional RNN that requires iterations of feedforward and feedback convolution operations. Compared with popular feedforward convolutional neural networks (CNNs), a convolutional RNN requires possibly an order of magnitude or more operations than a comparable CNN.

To address the programmability and complexity challenges, we note two design opportunities. First, if a configurable convolution engine can be made to support various kernel sizes, a convolutional sparse coding hardware can be made programmable to support different applications. Second, convolutional sparse coding and sparse coding, in general, provide inherent data sparsity that can be leveraged to reduce the computational complexity to improve both performance and efficiency.

Convolution engines supporting the configurable kernel size have been reported recently [20]–[24]. However, in these designs, the hardware, i.e., multipliers and adders, cannot be fully utilized in supporting all kernel sizes, and these designs are unable to further improve their performance by exploiting input sparsity effectively. A sparse convolution engine was presented to increase the throughput by skipping zeros in the input [25]; however, it only supports a fixed kernel size, so its application is limited. A scalable hardware architecture consisting of asynchronous modules has been designed for spiking neural networks [26], [27], but the development of such asynchronous designs requires specialized CAD tools, which are not readily accessible.

In this paper, we present a configurable spiking convolutional sparse coding accelerator that incorporates three new features to advance the state of the art:

1) a new programmable convolution architecture, named maze convolution, that supports configurable kernel sizes and the full utilization of the hardware for all supported kernel sizes;

Manuscript received February 12, 2018; revised May 16, 2018 and July 26, 2018; accepted July 26, 2018. Date of publication September 1, 2018; date of current version September 21, 2018. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Cooperative Agreement HR0011-13-2-0015, in part by Systems on Nanoscale Information fabriCs (SONIC), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and in part by Intel Corporation. This paper was approved by Guest Editor Chia-Hsiang Yang. *(Corresponding author: Chester Liu.)*

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA (e-mail: cwhliu@umich.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2018.2865457

2) a new zero-patch skipping technique that effectively exploits the sparsity in the input to increase the performance and efficiency of sparse convolutions;
3) a scalable globally asynchronous locally synchronous (GALS) hardware architecture that does not require specialized CAD tools and can be implemented in the standard digital design flow.

A prototype 40-nm test chip was designed to demonstrate the performance of 718GOPS at 380 MHz, consuming 257 mW. The prototype chip can be programmed for a variety of applications for learning and extracting features, and performing classifications.

II. Sparse Neural Coding Algorithm and Mapping

Sparse coding is a neuroinspired coding algorithm that attempts to find efficient, sparse representation of input stimulus. Through learning, sparse coding can be used to develop a dictionary of basis functions, or features, that are the representative of the underlying structure in the data.

For sparse coding, the learning is unsupervised, so it can be deployed in the field and learned directly from unlabeled data. The resulting dictionary is overcomplete, meaning that the size of the dictionary is larger than the input dimension. The dictionary is developed in a way so as to maximize the sparsity of the representation, which is a key feature of sparse coding. After learning converges, sparse coding can be used for inference to encode an input stimulus using the learned dictionary.

Mathematically, sparse coding can be described by $\mathbf{x} = \mathbf{a}\Phi$, where $\mathbf{x}$ is a given input, Φ is the dictionary, and $\mathbf{a}$ is the coefficient vector that is inferred by sparse coding. The objective of sparse coding is to find $\mathbf{a}$ that minimizes the encoding error and maximizes the sparsity of $\mathbf{a}$, i.e.,

$$\arg\min_{\mathbf{a}}(\|\mathbf{x} - \mathbf{a}\Phi\|^2 + \lambda T(\mathbf{a})) \tag{1}$$

where T is a threshold function and λ controls the weighting between the encoding error term and the sparsity term.

It is advantageous to adopt sparse coding in designing practical image, video, and audio processing systems, as it learns a good dictionary that is the representative of the data, and it allows the compression of large, dense inputs to sparse coefficient vectors, akin to compressive sampling.

From the hardware design point of view, sparse coding produces sparse coefficient vectors, which simplify downstream processing to improve the throughput and the energy efficiency. Unlike conventional image and video codecs whose dictionary of basis functions conveys a little information about the input, the dictionary learned by sparse coding consists of representative features. As such, the encoding contains meaningful information as to which features are existent and important in the input. After sparse coding, downstream processing can be carried out directly in the encoded, i.e., compressed, domain.

A. Locally Competitive Algorithm

One of the earliest sparse coding algorithms Sparsenet solves the optimization of (1) using a conjugate gradient descent method [1]. A hardware implementation of conjugate gradient descent is however inefficient. Rozell *et al.* [16] introduced a sparse coding algorithm named locally competitive algorithm (LCA) that makes use of an RNN to solve the optimization of (1). The RNN can be efficiently parallelized and mapped to hardware, making it more appealing for practical applications.

In the LCA formulation, each neuron retains a feature and a potential that is charged with input stimuli through feedforward connections, and discharged with lateral inhibition through feedback connections. Inference using LCA is carried out over iterations. An iteration consists of a feedforward step for the input stimuli to excite the neurons and a feedback step for the neurons to inhibit or "compete" with each other to represent the input. A typical inference converges in a few tens of iterations.

In performing the feedforward and feedback steps, a few simple rules are followed: 1) the closer a neuron's feature resembles the input, the faster the neuron's potential is charged; 2) a pair of neurons' inhibition is the strongest if the pair shares similar features; and 3) a neuron of high potential leaks faster than a neuron of low potential. Combining these rules, LCA's inference is described mathematically by the following equations:

$$\begin{aligned}\Delta\mathbf{u} &= \Phi^T\mathbf{x} - (\Phi^T\Phi - I)T(\mathbf{u}) - \mathbf{u}\\ \mathbf{u}' &= \mathbf{u} + \alpha\cdot\Delta\mathbf{u}\end{aligned} \tag{2}$$

where $\mathbf{u}$ is an N-dimensional vector that stores neuron potentials, $\mathbf{x}$ is an M-dimensional vector that stores input stimuli, Φ is an $M \times N$ matrix that stores the dictionary, I is an identity matrix that is used to remove self-inhibition, T is the threshold function, α is the step size, and $\mathbf{u}'$ is the updated neuron potential vector. In (2), $\Phi^T\mathbf{x}$ describes the feedforward excitation; $-(\Phi^T\Phi - I)T(\mathbf{u})$ describes the feedback inhibition; and $-\mathbf{u}$ describes the leakage term.

Thanks to the RNN formulation, LCA is suitable for hardware implementation, and it has been successfully demonstrated in hardware for image classification [14]. However, LCA relies on a fully connected network with both feedforward and feedback connections, i.e., each neuron is connected to all the inputs and all the other neurons. The fully connected network does not scale with the input dimension.

B. Convolutional Formulation of LCA (CLCA)

Features in images tend to be shift-invariant, meaning that a feature may appear at different locations in an image. Scanning (i.e., convolving) the image with small-sized features known as kernels is much more efficient than processing the image using features of the same size as the image itself.

Taking advantage of this insight, Schultz *et al.* [19] introduced convolution to the original LCA algorithm to improve scalability by using small-sized kernels. In convolutional LCA, the operational steps are the same as in LCA, but each neuron is equipped with a potential map, as opposed to a single potential, to keep track of the potential updated by the

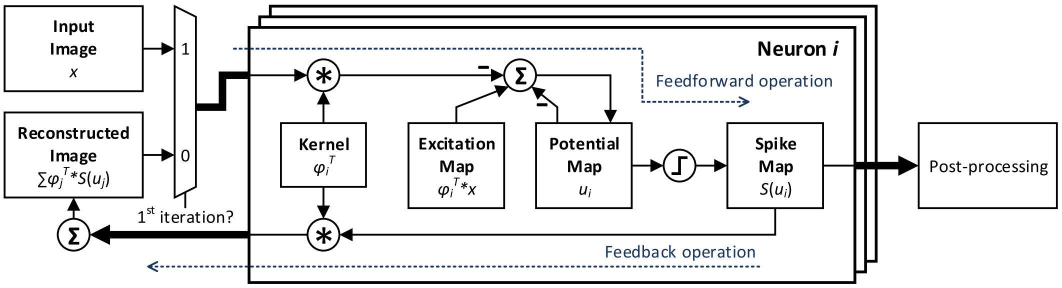


Fig. 1. Hardware mapping of the spiking convolutional sparse coding algorithm. In the first iteration, the input image is selected and the convolution result is stored in the excitation map. In the subsequent iterations, the image reconstructed by the previous iteration's feedback operation is selected. The neuron generates spikes when the potential exceeds a threshold. The collection of spikes forms the sparse representation of the input image.

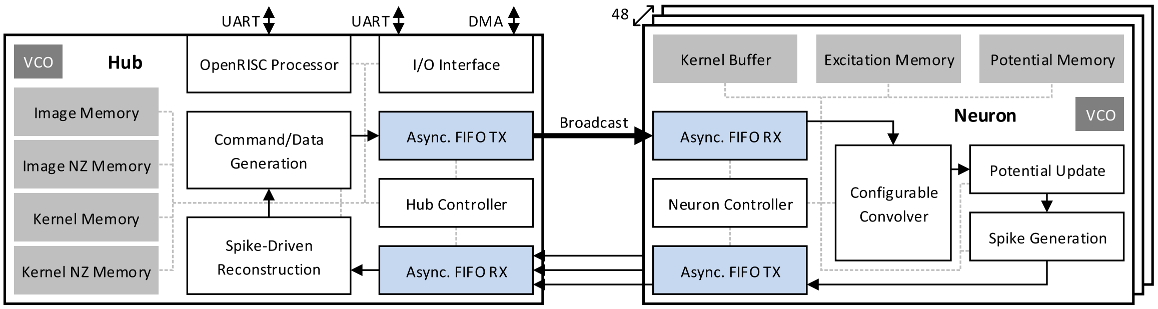


Fig. 2. Block diagram showing the system containing a hub and 48 neurons, all running in different clock domains.

following equation:

$$\Delta u_i = \left(\phi_i^T * x\right) - \phi_i^T * \left(\sum_j^N \phi_j * S(u_j)\right) - u_i. \qquad (3)$$

Equation (3) largely resembles (2) except that matrix multiplications are all replaced by 2-D convolutions, and a binary threshold function S is applied, i.e., $S(x) = 1$ if x is above a threshold, otherwise $S(x) = 0$. Equation (3) can be implemented in a convolutional RNN that consists of: 1) $\phi_i^T * x$ as the feedforward operation in the first iteration; 2) $\sum_j \phi_j * S(u_j)$ as the feedback operation; 3) $\phi_i^T * \sum_j \phi_j * S(u_j)$ as the feedforward operation in subsequent iterations; and 4) $-u_i$ as the leakage term in every iteration. Mapping of (3) to hardware is shown in Fig. 1.

III. System Architecture

To implement a convolutional RNN for sparse coding, a modular hardware architecture consisting of a single hub and a multitude of neurons is designed as shown in Fig. 2, where the feedforward operations are distributed to the neurons, and the neuron output, in the form of binary spikes, is sent to the central hub for feedback operations. The sparse neuron spikes allow us to share one hub for feedback operations.

In a feedforward operation, the hub broadcasts a dense input image (in the first iteration) or a sparse reconstructed image (in subsequent iterations) to neurons. Upon receiving an image, each neuron convolves it with its kernel and accumulates the result to the potential map. Spikes will be generated for locations in the map that exceed a threshold. The convolution is performed by the proposed convolution engine that is optimized for both dense and sparse input with a configurable kernel size. The design of the convolution engine is described in Section IV.

In a feedback operation, neuron spikes are convolved with their kernels to reconstruct the input image. A direct implementation of this convolution is computationally expensive. We take advantage of the binary spikes to replace all multiplications in the convolution by additions, and further make use of the high sparsity of the spikes (typically >80% sparse) to design a sparsely activated spike-driven reconstruction that saves computation, power, and chip area. The design of the spike-driven reconstruction is described in Section VI.

Each neuron is equipped with one 8-Kb (32 b×32 b×8 b) excitation memory, which buffers the excitation computed in the first iteration, one 8-Kb (32 b×32 b×8 b) potential memory, and a 1.8-Kb (15 b×15 b×8 b) latch-based kernel buffer. The hub contains 96-Kb (48 b×16 b×16 b×8 b) kernel memory, 19-Kb kernel non-zero (NZ) memory, 32 Kb (32 b×32 b×16 b×2) image memory, and 4-Kb image NZ memory. NZ memory provides fast NZ entry lookup and is described in Section V. Image memory and image NZ memory are double-buffered, enabling seamless data transfer from one feedback operation to the next iteration's feedforward operation. With these memory configurations, the system can process up to 15 × 15 kernel size and 32 × 32 image size. An input image exceeding this size needs to be divided into sub-images, with overlaps if necessary to minimize edge artifacts. Note that the kernel and the image size are not limited by the architecture, but by the available on-chip memory.

The modular structure is made possible by deploying a voltage-controlled oscillator (VCO) in every neuron and

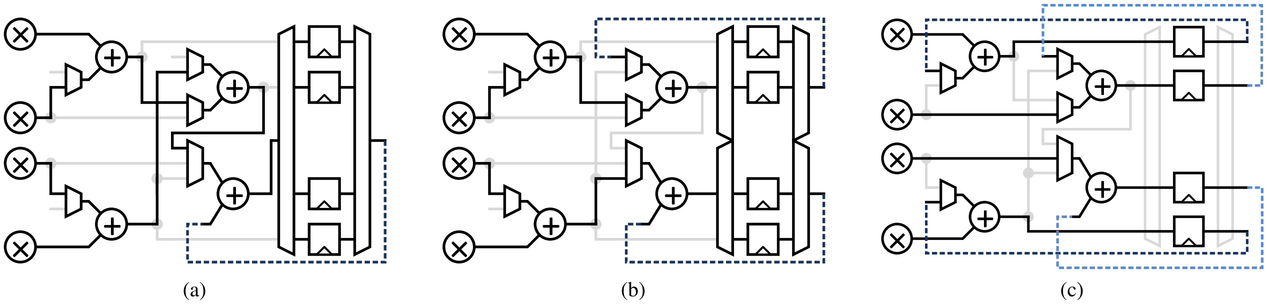


Fig. 3. Three modes of a 2 × 2 configurable convolver. (a) Mode 1 adds four products and accumulates the sum to one of the four buffer entries. (b) Mode 2 operates in 2-way parallel, each adding two products and accumulating the sum to one of the two buffer entries. (c) Mode 3 operates in 4-way parallel, each accumulating one product to one buffer entry.

the hub. The hub broadcasts commands and data to all neurons through a 128-bit (time-multiplexed 128-bit command and 16 8-bit data) asynchronous FIFO, and each neuron sends neuron spikes back to the hub through a 10-bit (5-bit x- and y-coordinates) asynchronous FIFO. The design of asynchronous FIFOs is described in Section VII.

The accelerator is initialized by populating the kernel and kernel NZ memories with data loaded through a 16-bit bidirectional direct memory access (DMA) interface in the hub, which is also used to off-load neuron spikes for verification. A UART interface in the hub is used for controlling operations and setting configurations such as the kernel size and the number of iterations. As a demonstration of an integrated system, the accelerator is integrated with an OpenRISC processor, which can be tasked with learning and other postprocessing operations.

IV. Configurable Convolver

A 2-D convolution operation can be viewed graphically by sliding the kernel on the top of the input image and computing one inner product of the kernel with the covered image portion per step of the slide. Each inner product involves an elementwise product of the kernel and the covered image portion followed by the summation of the elements. If the kernel size is fixed, a parallel compute array can be designed to support the inner product computation. In the following, we will refer to the inner product compute array as a convolver. If the size of the kernel varies, a convolver of a fixed size cannot be efficiently utilized. How to design a configurable convolver to support different kernel sizes while achieving the highest utilization is a challenge.

A. Fractional Partition and Combine

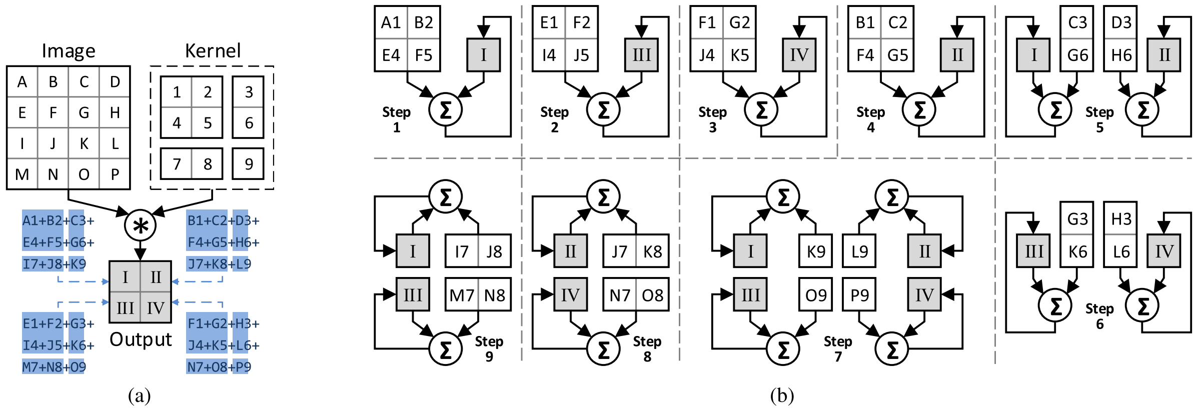
To address this challenge, our high-level idea is to design a fixed $C \times C$ convolver, and partition a $K \times K$ ($K > C$) kernel into $C \times C$ sub-kernels that can be directly mapped to the $C \times C$ convolver. Convolutions by sub-kernels produce partial sums, and the complete convolution results are formed by adding the partial sums. If a kernel cannot be partitioned into an integer number of $C \times C$ sub-kernels, a naïve design would result in the underutilization of the convolver. To overcome this inefficiency, we apply a new fractional partition and combine scheme to ensure the full utilization of the hardware.

For simplicity, we will use a $C = 2$, 2 × 2 convolver for illustration. A 2 × 2 convolver is made of four multipliers and four adders. We add configurable input connections to the adders and a multi-port buffer memory with four entries to make a configurable convolver that supports three basic modes.

In mode 1, as shown in Fig. 3(a), the convolver computes four products and a 4:1 summation, i.e., sums the four products and accumulates the sum in one of the four buffer entries. In mode 2, as shown in Fig. 3(b), the convolver functions as two independent sub-convolvers work in parallel. Each sub-convolver computes two products, sums them, and accumulates in one of the buffer entries. In mode 3, as shown in Fig. 3(c), the convolver functions as four independent sub-convolvers. Each sub-convolver performs one product followed by the accumulation in one buffer entry.

With the configurable convolver, we illustrate our fractional partition and combine scheme in Fig. 4 for an example of a 2 × 2 configurable convolver computing the convolution of a 3 × 3 kernel with a 4 × 4 image. The convolution will create four output values. In this case, $K = 3$ is not divisible by $C = 2$, so fractional partition is done as follows: the 3 × 3 kernel is partitioned into four sub-kernels: a 2 × 2 square, a 2 × 1 column, a 1 × 2 row, and a 1 × 1 element. We show the four sub-kernels in Fig. 4(a) and highlight how each is used to compute partial sums. Note that except for the 2 × 2 square, the other three partitions are fractional.

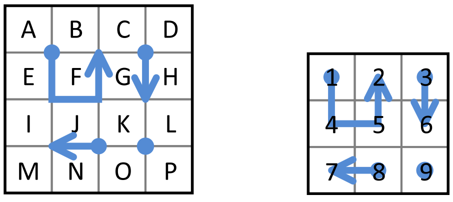
The convolution proceeds as illustrated in Fig. 4(b) by convolving sub-kernels with the input image using the 2 × 2 configurable convolver. In step 1, mode 1 is set to convolve the 2 × 2 sub-kernel with the top-left 2 × 2 input image patch. In step 2, the 2 × 2 sub-kernel slides one row down the input image to compute another 2 × 2 convolution. Similarly, in steps 3 and 4, the 2 × 2 sub-kernel slides one column right and one row up, respectively, to compute the 2 × 2 convolutions. In step 5, mode 2 is set to convolve the 2 × 1 sub-kernel with two 2 × 1 columns in the input image. The two 2 × 1 fractional partitions are combined to fully utilize the convolver. A similar operation is done in step 6. In step 7, mode 3 is set to convolve the 1 × 1 sub-kernel with four 1 × 1 elements in the input image. The four 1 × 1 fractional partitions are combined to ensure the full utilization of the hardware. In steps 8 and 9, mode 2 is set to convolve the 1 × 2 sub-kernel



(a)

(b)

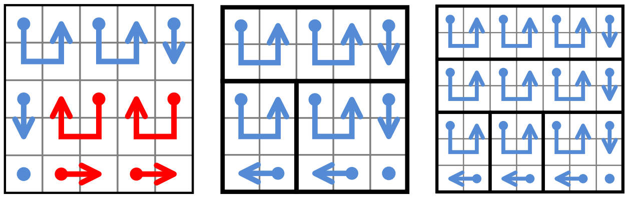
Fig. 4. Convolution of a 3 × 3 kernel with a 4 × 4 image. (a) Computation of the four output values. (b) Compute the convolution in nine steps using the 2 × 2 convolver. The convolver is set to mode 1 for steps 1–4, mode 2 for steps 5 and 6, mode 3 for step 7, and mode 2 for steps 8 and 9.



(a)

(b)

Fig. 5. Maze path for a 3 × 3 kernel. (a) When overlaid on image, the path represents how it is traversed by the 2 × 2 convolver. (b) When overlaid on kernel, the sub-paths cover the sub-kernels used in the corresponding steps.



(a)

(b)

(c)

Fig. 6. Construction of maze path for larger kernels. (a) To maximize data reuse for the 5 × 5 kernel, two new sub-path types are needed. (b) Maze path constructed by adding one 2 × 5 segment and one 3 × 2 segment to the 3 × 3 maze path, all using the sub-paths already defined for 3 × 3 kernel. (c) Maze path for 7 × 7 constructed by adding segments to the 3 × 3 maze path.

with two 1 × 2 rows in the input image. Again, the two 1 × 2 fractional partitions are combined.

By partitioning a kernel to sub-kernels and mapping fractional sub-kernel convolutions from consecutive steps on the convolver, we ensure the full utilization of the hardware and reduce the number of processing steps from 16 to 9 in this small example, achieving the highest efficiency and the minimum latency.

B. Maze Convolution

Notice from the small example that the steps are designed to maximize data reuse following two principles: 1) steps that use the same convolver mode are grouped together to maximize the reuse of the same sub-kernel and 2) only allow a single row or column slide in the input image between steps to maximize the reuse of the input. These principles lead to a carefully designed maze walking path that traverses the input image, as illustrated in Fig. 5(a). The maze walking path consists of a set of sub-paths that start with a dot and end with an arrow, and each sub-path corresponds to a convolver mode, as illustrated in Fig. 5(b). We name the convolution following the maze walking path "maze convolution."

Having a dedicated maze path for each kernel size allows us to maximize data reuse, but it also increases the storage requirement and control complexity. Fig. 6(a) illustrates a maze path for a 5 × 5 kernel that maximizes data reuse. The maze path uses two new types of sub-paths, as highlighted in Fig. 6(a), adding extra storage requirement and control complexity. Instead, we can design the maze path for a large kernel incrementally based on the sub-paths that are defined for a small kernel. As illustrated in Fig. 6(b), the maze path for the 5 × 5 kernel can be composed of the sub-paths already defined for the maze path for the 3 × 3 kernel shown in Fig. 5(b). High modularity is achieved by constructing the maze path incrementally, as exemplified by the maze path for the 7 × 7 kernel in Fig. 6(c), and maze paths for larger kernels, e.g., 9 × 9, 11 × 11, and so on, can be constructed based on the same principle.

C. Convolver Design Specification

To generalize, a $C \times C$ configurable convolver is made of C^2 multipliers, C^2 adders, and C^2 buffer entries. A $C \times C$ convolver supports any convolution kernel size and image size. Full utilization of the convolver can be achieved by mapping the convolution of a $K \times K$ $(K > C)$ kernel with a $(C + K - 1) \times (C + K - 1)$ image, producing a $C \times C$ output. Such a convolution requires K^2C^2 multiplications and accumulations, and only K^2 steps to complete on the $C \times C$ convolver. A smaller image size can also be mapped, but it would result in underutilization of the hardware. A larger image can also be mapped, but for full utilization, the image needs to be of the size $((C+K-1)+NK)\times((C+K-1)+NK)$ $(N \in \mathbb{Z}, N \geq 0)$ or padded to the size.

In our design, padding is performed implicitly by the controller to maintain a compact and constant memory size for

TABLE I

COMPARISON OF SPARSITY UTILIZATION TECHNIQUES

Technique	Throughput	Overhead
Zero-entry masking [20]	low	low
Zero-line skipping [25]	mid	mid
Zero-entry skipping [28], [29]	highest	highest
Proposed zero-patch skipping	high	mid

the convolver, which is especially important for a configurable convolver, because the size of padding varies for different kernel sizes. After padding, a larger image is processed by the configurable convolver one patch at a time. Each patch is $(C + K - 1) \times (C + K - 1)$, and adjacent patches are offset by K columns or K rows. The total number of patches is $(N + 1)^2$, and thus, the total number of steps to complete the convolution is $(K(N + 1))^2$.

V. SPARSITY OPTIMIZATION

If the input to a convolver is sparse, i.e., the input contains many zero entries, there is a potential opportunity to improve the design to increase the processing throughput and efficiency. In particular, three methods have been demonstrated in prior work. The first method, which we name zero-entry masking [20], disables the multiply accumulate circuit when encountering a zero in the input. Zero-entry masking reduces the dynamic power consumption, but the throughput remains constant, because it does not skip over a zero entry. The second method, which we name zero-line skipping [25], skips over an input line containing all zeros. Zero-line skipping increases the throughput as the input sparsity increases, and it can be combined with zero-entry masking to reduce the power consumption. However, we notice that, in practice, it is more likely to have an $n \times n$ ($n \in \mathbb{Z}, n > 0$) square patch of zeros than a $1 \times n^2$ line of zeros. The third method, which we name zero-entry skipping [28], [29], aims to skip over all zeros in the input. Zero-entry skipping, albeit theoretically optimal, requires complex control and incurs large hardware overhead.

In this paper, we design a new sparsity optimization method called zero-patch skipping to skip over square patches in the input that contain all zeros. As shown in Table I, compared with the three existing methods, zero-patch skipping offers several advantages: 1) higher throughput than zero-entry masking; 2) more effective than zero-line skipping due to the higher likelihood of encountering a square patch of zeros than a line of zeros; and 3) lower hardware overhead than zero-entry skipping thanks to the simpler per-patch-based control than the entry-by-entry-based control.

To enable zero-patch skipping, we introduce a new data structure called NZ map to associate with each input. Conceptually, an NZ map is constructed by scanning an $I \times I$ input image with a $C \times C$ NZ detector. The NZ detector outputs 1 if at least one entry in a $C \times C$ patch is NZ, otherwise it outputs 0. After scanning the $I \times I$ input, an $(I - C + 1) \times (I - C + 1)$ NZ map is constructed. An NZ map example is shown in Fig. 7(a) for the given input image.

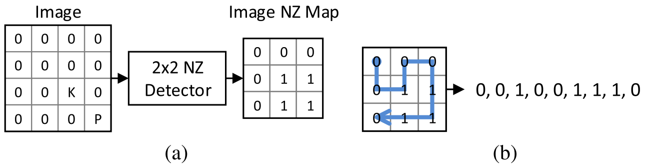


Fig. 7. Example of zero-patch skipping. (a) NZ map obtained from scanning the image with an NZ detector. (b) Maze path guided by an NZ map.

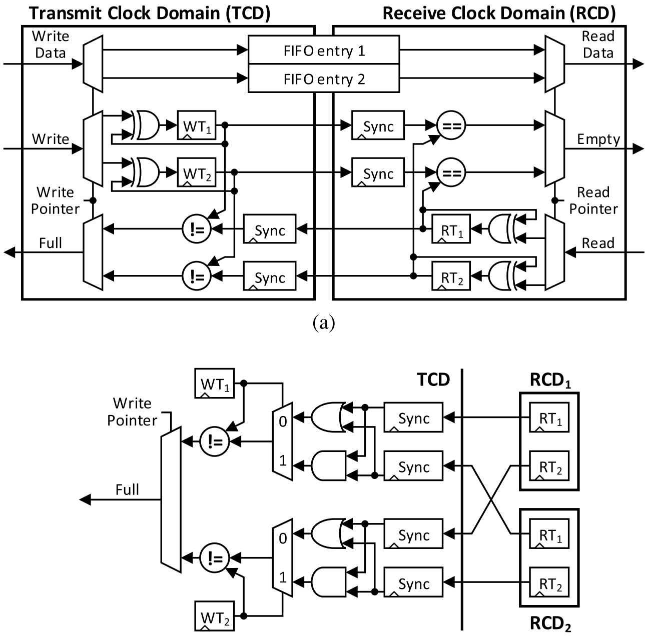
In practice, we do not use an NZ detector, because it would be too costly in terms of latency. Instead, the NZ map is built incrementally using very simple operations. In the first iteration, the NZ map is initialized with all 1s, because the input image is assumed to be dense. The NZ map is updated in every subsequent iteration in the reconstruction process based on the neurons, or kernels, that are activated. Since sparse coding enforces sparse neuron spike, the NZ map will be sparsely populated with 1s. The construction of the NZ map will be discussed in detail in Section VI.

Maze convolution natively supports zero-patch skipping. Guided by the NZ bit sequence obtained from traversing the NZ map using the maze path, maze convolution skips steps where the NZ bit is 0 to realize the sparsity-proportional throughput increase. We show in Fig. 7(b) the NZ bit sequence for the NZ map constructed in Fig. 7(a), and that five steps are skipped in performing the maze convolution, effectively doubling the throughput. Compared with [25], maze convolution with zero-patch skipping increases the throughput by up to 40% at 90% input sparsity. The proposed maze convolution with zero-patch skipping is equally applicable to CNNs [30].

VI. SPIKE-DRIVEN RECONSTRUCTION

Image reconstruction is performed at the end of each iteration to compute the input for the next iteration. Reconstruction latency therefore needs to be minimized so as not to halt the next iteration. Reconstructing an image according to (3) is costly in terms of both computation and latency as it involves convolution of every neuron's output with the neuron's kernel followed by the summation of all convolution results. However, in computing convolution, we are able to replace the expensive multiplication with simple addition by exploiting the neuron's spiking output to reduce the computation cost significantly. Furthermore, the sparse spikes allow us to hide the latency by performing image reconstruction incrementally.

Triggered by a neuron's spike, the hub performs image reconstruction by retrieving the neuron's kernel from the kernel memory and accumulating the kernel in the image memory, with the kernel's center aligned to the spike location. In parallel, the hub incrementally constructs the NZ map of the reconstructed image by ORing the kernel's NZ map retrieved from the kernel NZ memory with the image NZ memory. Constructing an NZ map incrementally eliminates the need to scan the reconstructed image, saving both computation and latency. Kernel NZ maps are pre-computed and loaded to the kernel NZ memory during initialization. The spike-driven reconstruction eliminates the need to store the spike



(a)

(b)

Fig. 8. Asynchronous interface design. (a) Token-based asynchronous FIFO. (b) Modified logic for the asynchronous FIFO to support broadcast.

map within each neuron, and a 16-entry FIFO in the hub is sufficient for buffering spikes, cutting the storage by 2.5×.

VII. Globally Asynchronous Interfaces

We use a token-based asynchronous interface [31] to enable the exchange of messages between neurons and the hub that operate in different clock domains. The asynchronous interface consists of a transmit clock domain (TCD) located in the sender, a receive clock domain (RCD) located in the receiver, and an N-entry memory storage bridging the two clock domains as shown in Fig. 8(a).

TCD consists of a write pointer that selects the next entry, N write token (WT) bits, and N synchronized read token (RT) bits from the RCD. To send a message from the TCD to the RCD, the message data are written to the entry pointed by the write pointer, and the corresponding WT bit is toggled before the write pointer is incremented. If the WT bit and the synchronized RT bit pointed by the write pointer have the same value, the entry has been read and is vacant for write; on the other hand, if the WT bit and the synchronized RT bit pointed by the write pointer in TCD have different values, the data stored in the entry have not been read, indicating that the FIFO is full. The RCD design follows the same principle.

We modify the FIFO full condition logic to allow one TCD to broadcast data to M RCDs. When the WT bit pointed by the write pointer is 0, the FIFO is full if any of the M synchronized RT bits pointed by the write pointer is 1, which can be detected by ORing these M synchronized RT bits; on the other hand, when the WT bit is 1, the FIFO is full if any of the M synchronized RT bits is 0, which can be detected by ANDing these M synchronized RT bits. An example with $N = 2$ and $M = 2$ is shown in Fig. 8(b). Compared with a conventional handshake-based asynchronous FIFO that requires two-way handshake, the token-based asynchronous FIFO has lower transmission latency; however, it consumes more routing resources, because, instead of just the transmitted data, all entries are routed from TCD to RCD.

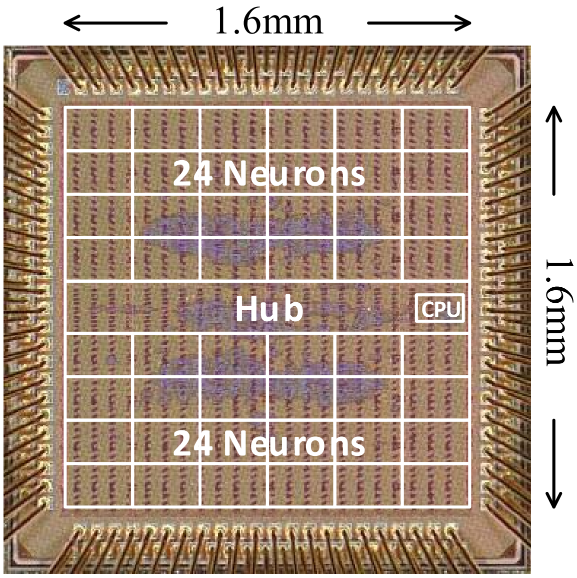


Fig. 9. Chip microphotograph.

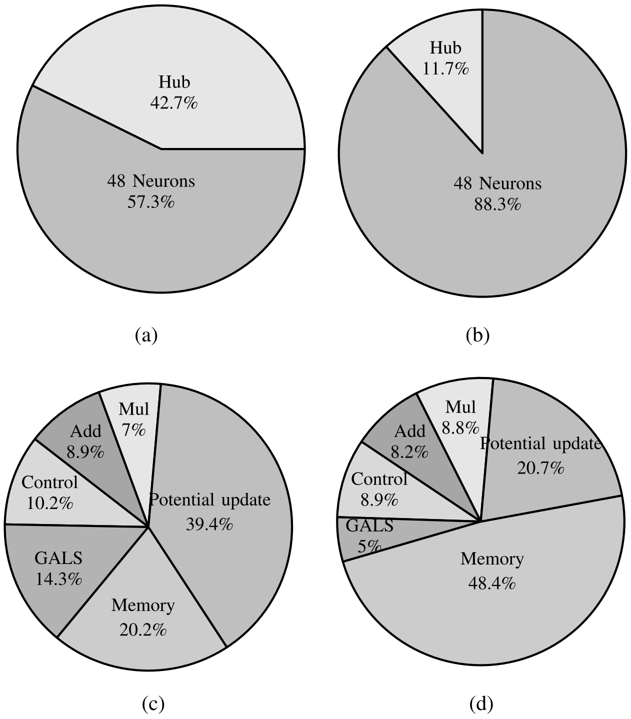


Fig. 10. Chip power and area breakdown. (a) Power breakdown of the entire chip. (b) Area breakdown of the entire chip. (c) Power breakdown of a single neuron. (d) Area breakdown of a single neuron.

VIII. Implementation Results

A 4.1-mm^2 test chip is implemented in a 40-nm CMOS process, and the core design occupies 2.56 mm^2. The chip

TABLE II
COMPARISON WITH PRIOR WORK

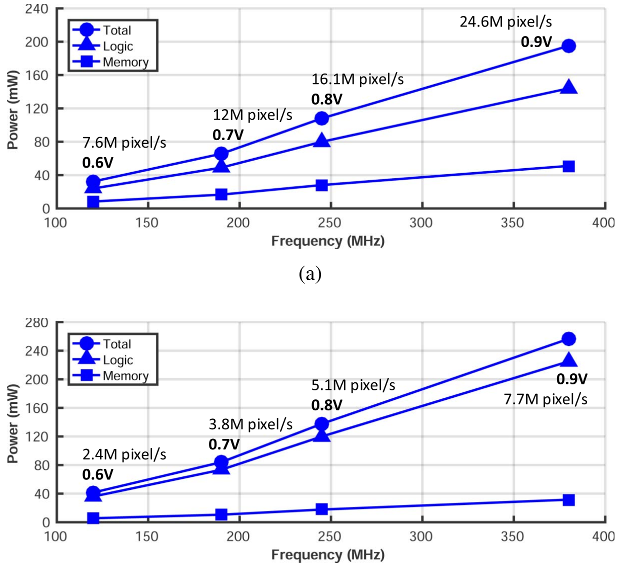
	This Work	VLSI 2015 [14]	VLSI 2016 [25]	JSSC 2016 [20]	ISSCC 2017 [21]
Architecture	Recurrent	Recurrent	Feedforward	Feedforward	Feedforward
Algorithm	Convolutional sparse coding	Sparse coding	Convolutional neural network	Convolutional neural network	Convolutional neural network
Technology	40nm GP CMOS	65nm GP CMOS	40nm GP CMOS	65nm LP CMOS	28nm UTBB FD-SOI
Configurable	Yes	No	No	Yes	Yes
Exploit Sparsity	Yes	No	Yes	No	No
Core Area	2.56mm^2	1.82mm^2	1.4mm^2	12.25mm^2	1.87mm^2
Kernel Size	5×5 to 15×15 (odd only)	16×16	8×8	Width: 1 to 32 Height: 1 to 12	programmable
Image Size	32×32	16×16	100×100	configurable	programmable
SRAM Size	120KB	37.6KB	93KB	181.5KB	144KB
Weight Width	8-bit	4-bit	8-bit	16-bit	16-bit
Voltage	0.9V	1V	0.9V	1V	1V
Frequency	380MHz	635MHz	240MHz	200MHz	200MHz
Performance	718GOPS	1138GOPS	898.2GOPS	33.6GOPS	204GOPS
Throughput†	24.6M pixel/s	10.16G pixel/s	96.4M pixel/s	1.8M pixel/s	2.42M pixel/s
Power	257mW	268.2mW	140.9mW	278mW	44mW
Power Efficiency	2.79 TOPS/W	4.24 TOPS/W	6.37 TOPS/W	0.12 TOPS/W	4.64TOPS/W
Area Efficiency‡	280 GOPS/mm^2	1.65 TOPS/mm^2	641.6 GOPS/mm^2	7.24 GOPS/mm^2	53.4 GOPS/mm^2

†Based on different benchmark due to the lack of a common benchmark. [14] reported input throughput while others reported output throughput.
‡Normalized to 40nm by multiplying the square of the ratio between node sizes.

microphotograph overlaid with the floorplan is shown in Fig. 9. We use a mixture of 80.5% high-V^{T} and 19.5% standard-V^{T} cells to reduce the logic leakage power by 33% (8.3 mW). Dynamic clock gating is applied to reduce the dynamic power by 24% (52 mW). A total of 49 VCOs, 48 for neurons and one for the hub, are instantiated, with each VCO occupying only 250 μm^2. A 4×4 configurable convolver and 4×4 NZ detection are implemented, supporting the odd kernel size from 5×5 to 15×15; the maximum supported kernel size is limited by the on-chip memory resource, not by the proposed maze convolution. The chip achieves a maximum 718GOPS running at 380 MHz with a nominal 0.9-V supply at room temperature; here, we define an operation as an 8-bit multiply or a 16-bit add.

Fig. 10 shows the power and area breakdown of the test chip. As shown in Fig. 10(a), the hub typically has more workload than individual neurons, and our experiments show that a balanced clock setting, in which the neuron clock frequency is lowered to 70% of the hub clock, can further reduce the power consumption by a maximum of 22% with less than 5% drop in the throughput. Maze convolution and zero-patch skipping require a more complex controller, and yet, this overhead is within 15% of the power and area budget for a neuron We use two sample applications to demonstrate the proposed accelerator: extracting sparse representation of images and extracting depth information from stereo images.

To extract sparse representation, we ran the accelerator with 7×7 kernels for 10 iterations per input image. The inference result, i.e., the collected neuron spikes, represents the sparse representation of the image. Kernels are trained using unsupervised learning with the natural scene image data set provided in [1]. When running at 380 MHz with a 0.9-V supply, the accelerator consumes 195 mW while providing 24.6 Mpixel/s throughput. Running at 120 MHz with a 0.6 V supply, the power consumption is lowered to 35 mW, as shown in Fig. 11(a), providing 7.6 Mpixel/s throughput.



(a)

(b)

Fig. 11. Measured power and throughput of (a) feature extraction application and (b) depth extraction application.

To estimate depth from stereo images [32], we use one accelerator per channel (left or right) to extract features. Both the accelerators use 15×15 kernels and run for 10 iterations per input image. Kernels are learned by training on the natural scene image data set. A simple matching algorithm can be programmed on the on-chip OpenRISC processor to estimate depth by comparing the distance between the locations where the same feature appears in the two channels. When running at 380 MHz with a 0.9-V supply, each accelerator consumes 257 mW while providing 7.68 Mpixel/s throughput. Running at 120 MHz with a 0.6-V supply, the power consumption is lowered to 45 mW, as shown in Fig. 11(b), providing 2.4 Mpixel/s throughput. Compared with the optimal baseline designs without exploiting sparsity, the throughput of the two tasks is improved by 7.7× and 9.7×, respectively.

IX. Conclusion

In this paper, we present a configurable convolver, maze convolution with sparsity optimization, and modular architecture and load balancing enabled by GALS. Using these techniques, we designed a convolutional RNN that implements convolutional sparse coding for a range of applications. The techniques are equally applicable to CNNs.

Compared with other state-of-the-art inference accelerators in Table II, the proposed accelerator implements a convolutional RNN that is configurable to support different kernel sizes, and exploits sparsity to increase throughput and efficiency. Compared with an ASIC implementation of sparse coding based on fully connected RNN [14], this design is scalable, configurable, and applicable to a wide range of applications. Compared with popular feedforward CNNs [20], [21], [25], the convolutional RNN implementing sparse coding can be more versatile as it supports unsupervised learning. The design sacrifices power efficiency and area efficiently by a small factor compared with [25] to gain configurability. Note that the supported image and the kernel size in this paper are limited by the available on-chip memory, not by the architecture, which in theory supports any image and kernel size. By exploiting sparisity, the power efficiency and area efficiency of this design easily overtake those of dense CNN accelerators [20], [21].

Acknowledgment

The authors would like to thank Prof. B. Olshausen and Dr. G. Kenyon for their advice.

References

[1] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, pp. 607–609, Jun. 1996.

[2] H. Lee, A. Battle, R. Raina, and A. Y. Ng, "Efficient sparse coding algorithms," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Dec. 2006, pp. 801–808.

[3] J. K. Kim, P. Knag, T. Chen, and Z. Zhang, "Efficient hardware architecture for sparse coding," *IEEE Trans. Signal Process.*, vol. 62, no. 16, pp. 4173–4186, Aug. 2014.

[4] R. Grosse, R. Raina, H. Kwong, and A. Y. Ng, "Shift-invariant sparse coding for audio classification," in *Proc. 23rd Conf. Uncertainty Artif. Intell. (UAI)*, Jul. 2007, pp. 149–158.

[5] M. D. Plumbley, T. Blumensath, L. Daudet, R. Gribonval, and M. E. Davies, "Sparse representations in audio and music: From coding to source separation," *Proc. IEEE*, vol. 98, no. 6, pp. 995–1005, Jun. 2010.

[6] B. A. Olshausen, "Sparse coding of time-varying natural images," in *Proc. Int. Conf. Independ. Compon. Anal. Blind Source Separat.*, 2000, pp. 603–608.

[7] C.-E. Lee, T. Chen, and Z. Zhang, "A 127 mW 1.63 TOPS sparse spatio-temporal cognitive SoC for action classification and motion tracking in videos," in *Proc. IEEE Symp. VLSI Circuits*, Aug. 2017, pp. C226–C227.

[8] S. Y. Lundquist, D. M. Paiton, P. F. Schultz, and G. T. Kenyon, "Sparse encoding of binocular images for depth inference," in *Proc. IEEE Southwest Symp. Image Anal. Interpretation (SSIAI)*, Apr. 2016, pp. 121–124.

[9] M. Elad and M. Aharon, "Image denoising via sparse and redundant representations over learned dictionaries," *IEEE Trans. Image Process.*, vol. 15, no. 12, pp. 3736–3745, Dec. 2006.

[10] M. Protter and M. Elad, "Image sequence denoising via sparse and redundant representations," *IEEE Trans. Image Process.*, vol. 18, no. 1, pp. 27–35, Jan. 2009.

[11] J. Yang, J. Wright, T. S. Huang, and Y. Ma, "Image super-resolution via sparse representation," *IEEE Trans. Image Process.*, vol. 19, no. 11, pp. 2861–2873, Nov. 2010.

[12] S. Yang, M. Wang, Y. Chen, and Y. Sun, "Single-image super-resolution reconstruction via learned geometric dictionaries and clustered sparse coding," *IEEE Trans. Image Process.*, vol. 21, no. 9, pp. 4016–4028, Sep. 2012.

[13] K. Kavukcuoglu, M. Ranzato, and Y. LeCun. (Oct. 2010). "Fast inference in sparse coding algorithms with applications to object recognition." [Online]. Available: https://arxiv.org/abs/1010.3467

[14] J. K. Kim, P. Knag, T. Chen, and Z. Zhang, "A 640Mpixel/s 3.65 mW sparse event-driven neuromorphic object recognition processor with on-chip learning," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2015, pp. 50–51.

[15] M. Yang, L. Zhang, J. Yang, and D. Zhang, "Robust sparse coding for face recognition," in *Proc. 24th IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2011, pp. 625–632.

[16] C. J. Rozell, D. H. Johnson, R. G. Baraniuk, and B. A. Olshausen, "Sparse coding via thresholding and local competition in neural circuits," *Neural Comput.*, vol. 20, no. 10, pp. 2526–2563, 2008.

[17] J. K. Kim, P. Knag, T. Chen, and Z. Zhang, "A 6.67 mW sparse coding ASIC enabling on-chip learning and inference," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2014, pp. 61–62.

[18] F. Heide, W. Heidrich, and G. Wetzstein, "Fast and flexible convolutional sparse coding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 5135–5143.

[19] P. F. Schultz, D. M. Paiton, W. Lu, and G. T. Kenyon. (Jun. 2014). "Replicating kernels with a short stride allows sparse reconstructions with fewer independent kernels." [Online]. Available: https://arxiv.org/abs/1406.4205

[20] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[21] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[22] A. Ardakani, C. Condo, M. Ahmadi, and W. J. Gross, "An architecture to accelerate convolution in deep neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 4, pp. 1349–1362, Apr. 2018.

[23] Y.-J. Lin and T. S. Chang, "Data and hardware efficient design for convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 5, pp. 1642–1651, May 2018.

[24] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2220–2233, Aug. 2017.

[25] P. Knag, C. Liu, and Z. Zhang, "A 1.40 mm^2 141 mW 898 GOPS sparse neuromorphic processor in 40 nm CMOS," in *Proc. IEEE Symp. VLSI Circuits*, Jun. 2016, pp. 180–181.

[26] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45 pJ per spike in 45 nm," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Oct. 2011, pp. 1–4.

[27] F. Akopyan *et al.*, "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1537–1557, Oct. 2015.

[28] J. Albericio, P. Judd, N. E. Jerger, T. Aamodt, T. Hetherington, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.

[29] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[31] I. M. Panades and A. Greiner, "Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in GALS architectures," in *Proc. Int. Symp. Netw. Chip (NOCS)*, May 2007, pp. 83–94.

[32] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Proc. IEEE Int. Conf. Vis. Pattern Recognit. (CVPR)*, Jun. 2012, pp. 3354–3361.

Chester Liu (S'09) received the B.S. degree in electrical engineering from National Tsing Hua University, Hsinchu, Taiwan, in 2008, and the M.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2010. He is currently pursuing the Ph.D. degree in electrical engineering and computer science with the University of Michigan, Ann Arbor, MI, USA.

From 2010 to 2013, he was with MediaTek Inc., Hsinchu, where he worked on the platform design and verification for smart phone system-on-chip (SoCs). He has been with the University of Michigan since 2014. His current research interests include neuromorphic computing, efficient hardware accelerator design for machine learning and robotics, and 2.5-D integration technologies.

Sung-Gun Cho (S'15) received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2010 and 2012, respectively. He is currently pursuing the Ph.D. degree in electrical and computer engineering with the University of Michigan, Ann Arbor, MI, USA.

From 2012 to 2015, he was with SK Hynix Inc., South Korea, where he was involved in system-on-chip (SoC) design and implementation of error control coding. His current research interests include energy-efficient, high-performance VLSI circuits and systems for neuromorphic computing, error control coding, and machine learning applications.

Zhengya Zhang (S'02–M'09–SM'17) received the B.A.Sc. degree in computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 2003, and the M.S. and Ph.D. degrees in electrical engineering from the University of California at Berkeley (UC Berkeley), Berkeley, CA, USA, in 2005 and 2009, respectively.

He has been a Faculty Member with the University of Michigan, Ann Arbor, MI, USA, since 2009, where he is currently an Associate Professor with the Department of Electrical Engineering and Computer Science. His current research interests include low-power and high-performance VLSI circuits and systems for computing, communications, and signal processing.

Dr. Zhang was a recipient of the David J. Sakrison Memorial Prize from UC Berkeley in 2009, the National Science Foundation CAREER Award in 2011, and the Intel Early Career Faculty Award in 2013. He was an Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART I: REGULAR PAPERS from 2013 to 2015 and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART II: EXPRESS BRIEFS from 2014 to 2015. He has been an Associate Editor of the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS since 2015.