SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules

Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, *Member, IEEE* and Xiaowei Li, *Senior Member, IEEE*

Abstract—Convolutional Neural Networks (CNNs) have been widely used in machine learning tasks. While delivering state-of-the-art accuracy, CNNs are known as both compute- and memory-intensive. This paper presents the SqueezeFlow accelerator architecture that exploits sparsity of CNN models for increased efficiency. Unlike prior accelerators that trade complexity for flexibility, SqueezeFlow exploits concise convolution rules to benefit from the reduction of computation and memory accesses as well as the acceleration of existing dense architectures without intrusive PE modifications. Specifically, SqueezeFlow employs a PT-OS-sparse dataflow that removes the ineffective computations while maintaining the regularity of CNN computations. We present a full design down to the layout at 65 nm, with an area of 4.80 mm^2 and power of 536.09 mW. The experiments show that SqueezeFlow achieves a speedup of $2.9\times$ on VGG16 compared to the dense architectures, with an area and power overhead of only 8.8% and 15.3%, respectively. On three representative sparse CNNs, SqueezeFlow improves the performance and energy efficiency by $1.8\times$ and $1.5\times$ over the state-of-the-art sparse accelerators.

Index Terms—Convolutional neural networks, accelerator architecture, hardware acceleration

1 Introduction

Convolutional neural networks (CNNs) have achieved an unprecedented accuracy on many machine learning applications, from object recognition and detection to scene understanding [1]–[5]. CNNs are known to be both compute- and memory-intensive, making it difficult to efficiently handle large scale CNNs on CPUs or GPUs [6]–[8]. Hence, a number of customized accelerators [9]–[30] have been proposed to deliver high computational throughput. However, with CNNs going larger and deeper to further yield higher accuracy, e.g., the amount of synaptic weights in [31], [32] reaches up to 10 billion, it remains a big challenge to efficiently process such networks even on state-of-the-art accelerators.

To deal with this problem, researchers have proposed many effective techniques to compress CNN models and reduce computation while maintaining comparably high accuracy with original CNNs. Previous studies [33]–[36] have demonstrated that a large fraction of input activations and weights can be pruned to zero without loss of accuracy. As shown in Fig. 1, more than 30% of the input activations and 50% of the weights in the convolutional layers (CVLs) of VGG16 are zero values, and more than 65% of the computations are unnecessary due to zero operands. Because zero values and their corresponding computations contribute nothing to the final results, the pruning techniques significantly shrink the amount of total computations in CNN inference.

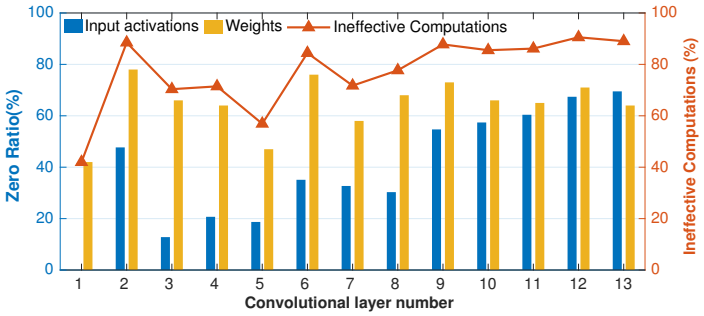


Fig. 1. The sparsity in the convolutional layers of VGG16.

However, such improvements are at the cost of regularity in CNN computations, making it hard to improve the performance and energy efficiency using existing accelerators which are good at processing regular and dense CNNs. The irregularity in sparse CNNs will inevitably introduce conditional branches for CNN computations to utilize the sparsity. Although enabling conditional branches is trivial for CPU-based solutions, it is hardly applicable for accelerators which are designed for fine-grained data or thread parallelism, rather than flexible data path control. The zeros are still fed into the accelerators to carry out unnecessary computations. Hence, dense accelerators hardly benefit from the sparsity in CNNs.

To overcome this problem, the prevailing method is to enable PEs with the flexibility to independently skip unnecessary computations [21]–[24]. Eyeriss [21] exploits sparsity by saving computation energy for zero-valued input activations and compressing weights stored in DRAM. Cnvlutin [22] and Cambricon-X [23] use a compressed encoding of activations and weights, respectively, in their dataflow to remove the corresponding ineffective computations and memory accesses. SCNN [24] eliminates ineffective computations from both

- *The authors are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100190, China. Part of this work was done while Jiajun Li was at YUSUR Technology, Co. Ltd.*
  E-mail: {lijiajun, jiangshuhao, gongshijun, wujingya, yanjunchao, yan, lxw}@ict.ac.cn
- *This work is supported by the National Natural Science Foundation of China under Grant Nos. 61872336, 61532017, 61572470, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000.*

Manuscript received 21 Nov. 2018; revised 11 June 2019; accepted 16 June 2019. (Corresponding authors: Guihai Yan and Xiaowei Li.)

zero-valued weights and activations simultaneously. However, these solutions significantly increase the hardware complexity. Moreover, all of them incur performance degradation due to the unbalanced sparsity distribution among the PEs. It is observed that the attainable performance merely reaches no more than 50% of their nominal performance.

Unlike prior approaches that trade complexity for flexibility, this paper takes an alternative by smartly partitioning sparse CNNs. We propose concise convolution rules (CCR), to diminish the gap between sparse CNNs and dense CNN accelerators. CCR transforms a sparse convolution into multiple effective and ineffective sub-convolutions. The ineffective convolutions in which either the activations or weights are all zero values do not contribute to the final results and their computations can be eliminated, while the effective ones in which both the activations and weights are dense can be easily mapped to existing dense CNN accelerators. By doing so, sparse CNNs can be easily mapped to existing dense CNN accelerators, obviating the intrusive PE modifications. CCR advocates a novel approach to benefit from the reduction of computation and memory accesses as well as the acceleration of existing dense architectures.

Furthermore, we present SqueezeFlow, an efficient CNN inference accelerator that exploits sparsity in CNNs following the architectural implications of CCR. Unlike previous convolutional dataflow that depends on complex PE designs to remove ineffective computations, SqueezeFlow employs a PT-OS-sparse dataflow that delivers input activations and only nonzero weights to the PE array, ensuring that only effective multiplications are fed into the PE array. Since the ineffective computations are already removed before forwarding computations to PEs, SqueezeFlow dataflow supports sparse CNNs with comparably light-weighted PEs to that in dense CNN accelerators. SqueezeFlow also employs compressed encoding of both activations and weights to reduce DRAM accesses. Meanwhile, inter-PE propagations is enabled to further reduce the expensive on-chip memory accesses.

To evaluate SqueezeFlow, we present a concrete design down to the layout in a 65 nm CMOS technology. We empirically evaluate SqueezeFlow as well as previous accelerators on a set of representative CNN models. The experimental results show that SqueezeFlow achieves $2.9\times$ speedup on VGG16 over a dense CNN accelerator with the same computational resources, with an area and power overhead of only 8.8% and 15.3%, respectively. Compared with the state-of-the-art sparse accelerators, SqueezeFlow improves the performance and energy efficiency by $1.8\times$ and $1.5\times$, respectively. Furthermore, SqueezeFlow has considerable merit on the scalability to PE array scale and compatibility to both sparse and dense CNNs.

The rest of this paper is organized as follows: Section 2 reviews the related works. Section 3 delves into the details of CCR including three cases on sparse CNNs. Section 4 introduces the architectural implications of CCR. In Section 5, we describe the dataflow and the detailed architecture design of the SqueezeFlow accelerator. Section 6 describes the experimental methodology. In Section 7, we implement SqueezeFlow and compare the results to state-of-the-art dense and sparse CNN accelerators. Section 8 discusses the future work. Section 9 concludes this paper.

2 Related Works

In recent years, accelerating CNNs has been one of the hottest research tops in computer architecture. This section reviews the related works on CNN accelerators and presents the motivation of building a new accelerator for sparse CNNs.

2.1 Dense CNN Accelerators

There are many ASIC accelerators for dense CNNs. The DianNao family is a series of hardware accelerators dedicated for neural networks. DianNao [12] is the first member focusing on memory bandwidth utilization. DaDianNao [13] is proposed for efficiently processing large-scale neural networks with sufficient on-chip memory. ShiDianNao [14] is designed to completely eliminate off-chip memory accesses in embedded systems. Besides the DianNao family, there are many accelerators focusing on optimizing CNNs from computing perspective or memory perspective. Eyeriss [21] is an ASIC CNN accelerator that couples a compute grid with a NoC, enabling flexibility in scheduling CNN computation. FlexFlow [20] exploits the different parallelism schemes in CVLs to boost PE utilization. DNA [18] is a DNN accelerator that leverages Weight, Input, and Output Reuse within the same fabric. MAERI [25] is a recent DNN accelerator for mapping arbitrary dataflow that arises in DNNs due to its topology or mapping.

FPGAs have also been widely used to build CNN accelerators as they are flexible to accommodate different dataflows based on their reconfigurable substrate. Zhang *et al.* [15] focuses on the balance between computing resource and memory bandwidth for CVLs. Shen *et al.* [37] has explored CNN architectures that focus on cross-CVL optimizations over FPGAs. Qiu *et al.* [16] proposed a CNN accelerator to address the limited bandwidth problem by a dynamic-precision data quantization flow. Suda *et al.* [17] proposed a methodology to find the optimal accelerator design for any CNN model implementation.

2.2 Sparse CNN Accelerators

Sparsity has proven to be an effective approach to reduce computation and memory accesses in CNNs. Many works have investigated how to compress CNN models while maintaining the accuracy, such as Sparse Coding [38] and Auto Encoder/Decoder [33]. A state-of-the-art pruning technique is proposed by Han *et al.* [35] that learns only important connections in CNNs and prunes the unimportant connections. It shrinks the amount of synaptic weights by about $10\times$ with negligible accuracy loss. Meanwhile, activation sparsity stems from zero-padding and the activation functions such as the rectified linear unit (ReLU).

The sparsity in CNNs will inevitably result in irregular workloads, which are difficult to accelerate by accelerators dedicated for dense and regular models. Lots of CNN accelerators have exploited sparsity to reduce energy and save computation time. Cnvlutin [22] stores sparse activations in compressed format and skips computation cycles for zero-valued activations to improve both performance and energy efficiency. Cambricon-X [23] exploits sparsity by compressing the pruned weights, skipping computation cycles for zero-valued weights. SCNN [24] leverages the sparsity in both weights and activations, exploiting an algorithmic dataflow

that eliminates ineffective computations from both zero-valued weights and activations simultaneously. EIE [39] performs inference on the compressed fully-connected layers and accelerates the resulting sparse matrix-vector multiplication. Mao *et al.* [40] explores the granularity of sparsity in CNNs and shows that coarse-grained sparsity is more hardware-friendly and energy-efficient for sparse CNN accelerators. UCNN [41] exploits weight repetition to save energy and improve performance during CNN inference. These methods all enable PEs with the flexibility to independently skip unnecessary computations, thereby reaping the benefits of sparsity.

However, these solutions significantly increase the underlying hardware complexity. Moreover, they incur performance degradation due to the unbalanced sparsity distribution among the PEs. Because their PEs are usually tightly-coupled together to generate the results, early-finishing PEs has to stay idle while waiting for the laggards. This load imbalance results in severe PE under-utilization thus degrading the performance. For example, it is confirmed by the experimental results that SCNN can only reach up mo more than 50% of its nominal performance.

To tackle this problem, this paper presents a novel accelerator architecture, called SqueezeFlow, to exploit sparsity of CNN models for increased efficiency. SqueezeFlow exploits concise convolution rules to benefit from the reduction of computation and memory accesses as well as the acceleration of existing dense architectures without intrusive PE modifications. Hence, SqueezeFlow exploits sparsity while being free from the problems that prior solutions incurred.

3 Concise Convolution Rules

In this section, we present the concise convolution rules, which smartly partitions sparse convolutions into effective and ineffective sub-convolutions. We first introduce the formulated representation for convolutions to better clarify the rationale behind CCR. Then, we describe the formulated representation of CCR and the proof.

3.1 Formulated Representation for Convolution

2D Convolution. As a building block for CNNs, 2D convolution is the process of adding each element of the image to its local neighbors weighted by the kernel. Specifically, the input map (*imap*) $\mathcal{I}$ is convolved by a kernel $\mathcal{K}$ and generates the output map (*omap*) $\mathcal{O}$, denoted as $\mathcal{O} = \mathcal{I} * \mathcal{K}$.

2D convolution has three modes to control the output shape, i.e., 'full', 'same' and 'valid'. In this paper, we focus on the 'full' mode, since the results in other modes can be also obtained through that in 'full' mode by cropping the *omaps*. Unless otherwise specified, the convolutions in the following texts are in full mode.

The computation of a 2D convolution is defined as:

$$\begin{aligned} \mathcal{O}(x,y) = \sum_{j=0}^{S-1}\sum_{i=0}^{R-1} \mathcal{K}(i,j)\cdot\mathcal{I}(x+i,y+j), \\ -R < x < H, \ \ -S < y < W, \\ E = H+R-1, \ \ F = W+S-1 \end{aligned} \tag{1}$$

where $H/W, R/S, E/F$ denote the height/width of the *imaps*, *kernel* and *omaps*, respectively. Note that the starting coordinate of $\mathcal{O}$ is not $(0,0)$ but $(1-R, 1-S)$, while its ending coordinate is $(H-1, W-1)$. We adopt such coordinate system to better formulate CCR.

Triplet Format for Sparse Matrix Representation. Before delving into the properties of 2D convolution, we firstly introduce the triplet format for a matrix representation. A matrix can be separated into multiple sub-matrices while recording their coordinates, i.e., the row and column index of the first element in each sub-matrix. Take a matrix $\mathcal{K} : (k_{ij})_{4\times4}$ as an example, it can be represented as follows,

$$\begin{aligned} &\mathcal{K} : \{(0,0,\mathcal{K}_1),(2,0,\mathcal{K}_2),(0,2,\mathcal{K}_3),(3,0,\mathcal{K}_4)\} \\ &or\ \mathcal{K} = \biguplus_{i=1}^{4}(r_i, c_i, \mathcal{K}_i) \end{aligned} \tag{2}$$

where

$$\begin{aligned} &r_1=0, c_1=0, r_2=2, c_2=0, r_3=0, c_3=2, r_4=3, c_4=0 \\ &\mathcal{K}_1 = \begin{bmatrix} k_{00} & k_{01} \\ k_{10} & k_{11} \end{bmatrix}, \mathcal{K}_2 = \begin{bmatrix} k_{20} & k_{21} \end{bmatrix}, \\ &\mathcal{K}_3 = \begin{bmatrix} k_{02} & k_{03} \\ k_{12} & k_{13} \\ k_{22} & k_{23} \end{bmatrix}, \mathcal{K}_4 = \begin{bmatrix} k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix} \end{aligned} \tag{3}$$

Similarly, the sub-matrices can be concatenated to generate the original one. Note that the sub-matrices may overlap with others. If this happens, the overlapped values should be accumulated when concatenated; otherwise, the sub-matrices should keep the original. The triple format provides a representation for sparse matrices since they are represented as combination of nonzero (dense) and zero matrices.

3.2 CCR on Sparse Kernels

CCR based on sparse kernels is defined as follows:

Property 1 (CCR-1)**.**

$$\begin{aligned} &if\ \ \mathcal{K} = \biguplus_{l=1}^{p}(r_l, c_l, \mathcal{K}_l),\ \ then\ \ \mathcal{I}*\mathcal{K} = \biguplus_{l=1}^{p}(\alpha_l, \beta_l, \mathcal{I}*\mathcal{K}_l) \\ &where\ \ \alpha_l = 1 - r_l - R_l,\ \ \beta_l = 1 - c_l - S_l \end{aligned} \tag{4}$$

where p denotes the number of sub-matrices decomposed from $\mathcal{K}$, r_l/c_l denote the coordinates (row and column index) of sub-matrix $\mathcal{K}_l$ in $\mathcal{K}$, R_l/S_l denote the height and width of sub-matrix $\mathcal{K}_l$. The proof of CCR-1 is as follows:

Proof of CCR-1.

$$\begin{aligned} \mathcal{O}(x,y) &= \sum_{j=0}^{S-1}\sum_{i=0}^{R-1}\mathcal{K}(i,j)\cdot\mathcal{I}(x+i,y+j) \\ &= \sum_{l=1}^{p}\sum_{j=c_l}^{c_l+S_l-1}\sum_{i=r_l}^{r_l+R_l-1}\mathcal{K}(i,j)\cdot\mathcal{I}(x+i,y+j) \\ &= \sum_{l=1}^{p}\sum_{j=0}^{S_l-1}\sum_{i=0}^{R_l-1}\mathcal{K}(i+r_l,j+c_l)\cdot\mathcal{I}(x+r_l+i,y+c_l+j) \\ &= \sum_{l=1}^{p}\sum_{j=0}^{S_l-1}\sum_{i=0}^{R_l-1}\mathcal{K}_l(i,j)\cdot\mathcal{I}(x+r_l+i,y+c_l+j) \\ &= \sum_{l=1}^{p}\mathcal{O}_l(x+r_l,y+c_l) \end{aligned} \tag{5}$$

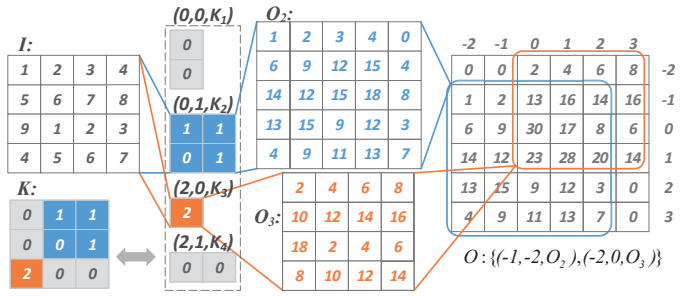


Fig. 2. An example of CCR on sparse kernels. The kernel $\mathcal{K}$ is decomposed to $\mathcal{K}_2$ and $\mathcal{K}_3$ to convolve $\mathcal{I}$, respectively.

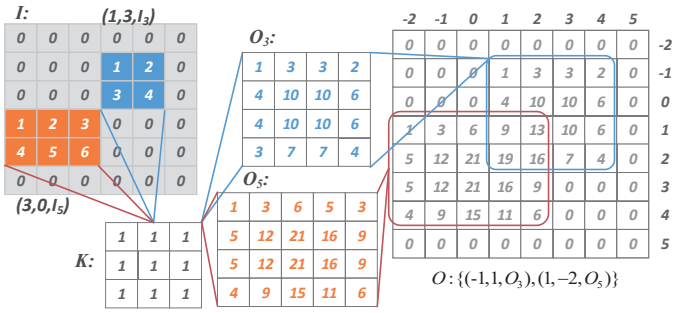


Fig. 3. An example of CCR on sparse input maps. The input map $\mathcal{I}$ is decomposed to $\mathcal{I}_3$ and $\mathcal{I}_5$ to be convolved with $\mathcal{K}$, respectively.

where $\mathcal{O}_l = \mathcal{I} * \mathcal{K}_l$.

According to (5), there exists an offset $(-r_l, -c_l)$ between the coordinates in $\mathcal{O}$ and $\mathcal{O}_l$. The starting coordinate of $\mathcal{O}_l$, i.e., $(1-R_l, 1-S_l)$, corresponds to coordinate $(1-R_l-r_l, 1-S_l-c_l)$ in $\mathcal{O}$, which is exactly the coordinate when concatenating the partial omaps. Therefore,

$$\mathcal{O} = \biguplus_{l=1}^{p} (1 - R_l - r_l, 1 - S_l - c_l, \mathcal{O}_l) \tag{6}$$

□

CCR-1 transforms a convolution into multiple sub-convolutions, each with the original imap and a sub-kernel. Based on CCR-1, the computation and memory accesses of convolutions with sparse kernels are reduced. By decomposing a sparse kernel into nonzero and zero sub-kernels using the triplet representation, the convolution is transformed into effective sub-convolutions (with nonzero sub-kernels) and ineffective ones (with zero sub-kernels). Since the ineffective sub-convolutions do not contribute to the final results, their computation and memory accesses can be eliminated. Fig. 2 demonstrates an example. The kernel is decomposed into $(0,0,\mathcal{K}_1), (0,1,\mathcal{K}_2), (2,0,\mathcal{K}_3), (2,1,\mathcal{K}_4)$, where $\mathcal{K}_1$ and $\mathcal{K}_4$ are zero sub-kernels and do not contribute to the final results. We only need to compute the effective sub-convolutions with $\mathcal{K}_2, \mathcal{K}_3$ and generate the sub-omap $\mathcal{O}_2, \mathcal{O}_3$, then concatenate them to the final results using the offsets calculated as:

$$\begin{aligned} \alpha_2 &= 1 - r_2 - R_2 = 1 - 0 - 2 = -1 \\ \beta_2 &= 1 - c_2 - S_2 = 1 - 1 - 2 = -2 \\ \alpha_3 &= 1 - r_3 - R_3 = 1 - 2 - 1 = -2 \\ \beta_3 &= 1 - c_3 - S_3 = 1 - 0 - 1 = 0 \end{aligned} \tag{7}$$

In this example, the computation volume quantified by multiply and accumulation (MAC) operations is significantly reduced. The original convolution contains $4\times4\times3\times3 = 144$ MACs, while the MAC operations aggregated by the effective sub-convolutions is $4\times4\times2\times2+4\times4\times1\times1 = 80$, achieving a computation reduction up to 44.4%. Meanwhile, the memory footprint is also reduced in CNN inference since we only need to record the nonzero sub-kernels and their coordinates.

Most importantly, CCR-1 diminishes the gap between sparse and dense convolutions. The effective sub-convolutions derived from CCR-1 maintains the same imap size with the original one, which reveals that they can be easily mapped to existing dense accelerators at low hardware overhead. The architectural implications will be introduced in Section 4.

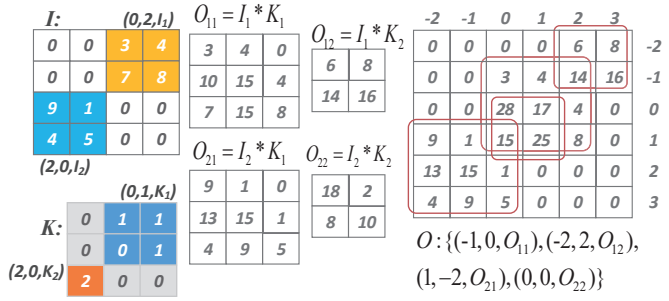


Fig. 4. An example of CCR on sparse kernels and input maps. The input map $\mathcal{I}$ is decomposed to $\mathcal{I}_1$ and $\mathcal{I}_2$, the kernel $\mathcal{K}$ is decomposed to $\mathcal{K}_1$ and $\mathcal{K}_2$. Each sub-imap is convolved with each sub-kernel, respectively.

3.3 CCR on Sparse Input Maps

Similarly, CCR based on sparse imaps is defined as:

Property 2 (CCR-2)**.**

$$\begin{aligned} &if\ \ \mathcal{I} = \biguplus_{l=1}^{q} (m_l, n_l, \mathcal{I}_l),\ \ then\ \ \mathcal{I} * \mathcal{K} = \biguplus_{l=1}^{q} (\gamma_l, \theta_l, \mathcal{I}_l * \mathcal{K}) \\ &where\ \ \gamma_l = 1 + m_l - R,\ \ \theta_l = 1 + n_l - S \end{aligned} \tag{8}$$

where q denotes the number of sub-imaps decomposed from $\mathcal{I}$, m_l/n_l denote the coordinates (row and column index) of sub-imap $\mathcal{I}_l$ in $\mathcal{I}$, R/S denote the height and width of kernel $\mathcal{K}$.

The proof of CCR-2 is similar to the one of CCR-1 and is omitted due to space limitations. According to CCR-2, a convolution is transformed to multiple sub-convolutions, each with a sub-imap and the original kernel. The computations of convolutions with sparse imaps are reduced in a similar way as CCR-1. As exemplified in Fig. 3, the imap $\mathcal{I}$ is partitioned into seven sub-imaps, where $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_4, \mathcal{I}_6, \mathcal{I}_7$ are zero sub-imaps that do not contribute to the final results. We only need to compute the effective sub-convolutions with $\mathcal{I}_3, \mathcal{I}_5$ and generate $\mathcal{O}_3, \mathcal{O}_5$, then concatenate them to generate the output map. The above example achieves a computation reduction up to 72.2%. The effective sub-convolutions maintain the same kernel size with the original one, which indicates that only limited hardware modifications to the previous accelerators are required to support the effective sub-convolutions.

3.4 CCR on Sparse Kernels and Input Maps

CCR-1 and CCR-2 simplify the computation of convolutions with sparse kernels or imaps, from which CCR based on both sparse kernels and sparse imaps is derived as follows:

Property 3 (CCR-3)**.**

$$
\begin{aligned}
&if\ \ K = \biguplus_{i=1}^{p}(r_i, c_i, K_i),\ \ \mathcal{I} = \biguplus_{j=1}^{q}(m_j, n_j, \mathcal{I}_j) \\
&then\ \ \mathcal{I} * \mathcal{K} = \biguplus_{j=1}^{q}(\gamma_j, \theta_j, \mathcal{I}_j * \mathcal{K}) \\
&= \biguplus_{j=1}^{q}(\gamma_j, \theta_j, \biguplus_{i=1}^{p}(\alpha_i, \beta_i, \mathcal{I}_j * \mathcal{K}_i)) \\
&= \biguplus_{j=1}^{p}\biguplus_{i=1}^{q}(\alpha_i + \gamma_j + R - 1, \beta_i + \theta_j + R - 1, \mathcal{I}_j * \mathcal{K}_i) \\
&= \biguplus_{j=1}^{p}\biguplus_{i=1}^{q}(m_j - r_i - R_i + 1, n_j - c_i - S_i + 1, \mathcal{I}_j * \mathcal{K}_i)
\end{aligned}
\tag{9}
$$

According to CCR-3, every sub-imap is convolved by every sub-kernel and generates a sub-omap. The convolutions with either zero sub-imaps or zero sub-kernels can be eliminated. Fig. 4 demonstrates an example, the nonzero sub-imaps $\mathcal{I}_1, \mathcal{I}_2$ are convolved by the nonzero sub-kernels $\mathcal{K}_1, \mathcal{K}_2$ and generate four sub-omaps $\mathcal{O}_{11}, \mathcal{O}_{12}, \mathcal{O}_{21}, \mathcal{O}_{22}$, which are then concatenated to generate the final results. CCR-3 maximally removes the ineffective computations since it exploits the sparsity in both imaps and kernels. It achieves a reduction on MAC operations up to 72.2% in the above example.

Since CCR decomposes 2D convolutions into matrix multiplications, we would like to mention the essential differences between CCR and prior matrix computation optimizations, such as [42]–[44]. It is well known that 2D convolution can be performed using just one matrix multiplication, by converting one of the input matrices, e.g. the imap, to a Toeplitz matrix [45]. This involves replicating the data elements in imap multiple times across different matrix columns. After constructing the Toeplitz matrix, convolution can be implemented using prior matrix multiplication optimizations. The major downside of such representation is the space explosion when building the column matrix. For a convolution with a 2D $k \times k$ kernel matrix, the column matrix is k^2 times larger than the original imap. The increased memory requirements will significantly increase the memory traffic, and lead to increased execution time. Therefore, this representation is rarely adopted by CNN accelerators which usually have limited on-chip local memories. Instead, the common method is using for-loops to perform 2D convolutions. In this scenario, a 2D convolution contains a set of matrix multiplications, where there exists plenty of data reuse opportunity. Prior matrix computation optimizations [42]–[44] only optimize for a single matrix multiplication, while CCR can be considered as a co-optimization of a set of matrix multiplications by exploiting data reuse. CCR eliminates the need for data replication on the input, which significantly shrinks the memory footprint. Because of the limited on-chip memory budget, using CCR approach to build CNN accelerators is easier to achieve high performance and energy efficiency.

3.5 Theoretical Reduction on Computation and Data Volume

Through CCR, we can calculate the theoretical reduction on computation and data volume. We take CVLs in VGG16 as an example. CCR-3 eliminates the most computations for all layers (79.7% on average) since it exploits the sparsity in both input maps and kernels. CCR-1 and CCR-2 remove 67.2% and 39.7% of the total computations, respectively. Meanwhile, CCR-1, CCR-2, and CCR-3 achieve 39.8%, 11.5%, 51.2% on data volume reduction, respectively.

4 Architectural Implications

This section introduces the architectural implications of CCR and presents how CCR bridges the gap between sparse CNNs and dense CNN accelerators. We start from a typical dense architecture and investigate how it evolves to support CCR without intrusive PE modifications.

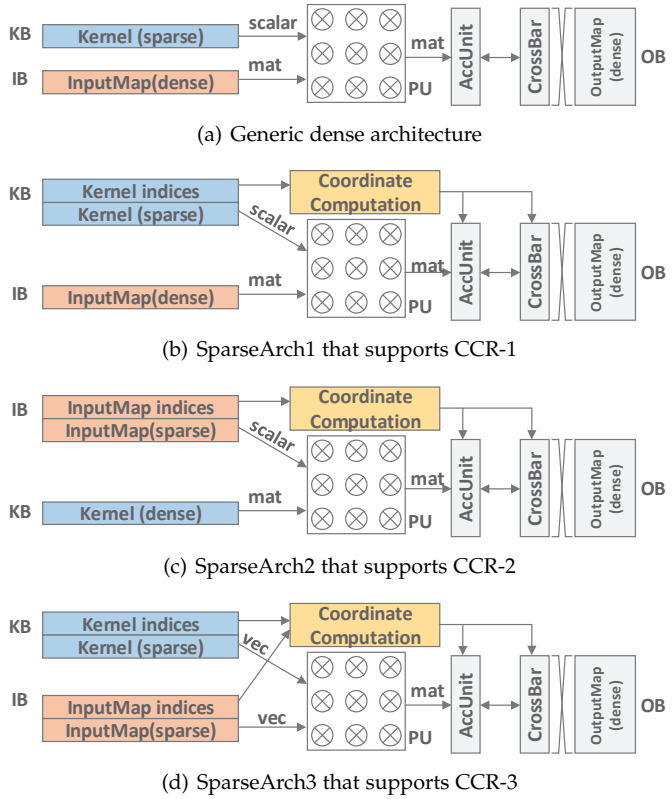
4.1 Generic Dense Architecture

A generic architecture of state-of-the-art dense accelerators [14] can be modeled as Fig. 5(a). It comprises a weight buffer (KB), input activation buffer (IB), output activation buffer (OB), a Processing Unit (PU), an accumulation unit (AccUnit) and a scatter crossbar. The PU is a 2D mesh of $P_x \times P_y$ Processing Elements (PEs). To process a convolution, a scalar of kernel and a matrix block of input map are fetched from their respective buffers and fed into the PU which computes a scalar matrix multiplication, i.e., every element in the matrix is multiplied by the scalar to form a $P_x \times P_y$ products. The products are delivered into AccUnit to accumulate with the corresponding partial sums fetched from OB, the result is then routed to OB by the crossbar. Because of the regular addressing pattern of OB, the output coordinates can be derived from the loop indices in a state machine (not shown in the figure). Note that the architecture we described is quite simple for the sake of investigating how it evolves to support CCR. Some unique features of prior solutions optimized for data movement are not enabled, more details can refer to [14].

In this architecture, the input maps and kernels are stored in uncompressed format and they proceed in lock-step. Even if the scalar from the kernel is zero, it is still fed into the PU to carry out unnecessary computations. Thus, the dense architectures cannot benefit from the reduction of both computation and memory access.

4.2 Sparse Architectures Supporting CCR

We firstly demo a sparse architecture that supports CCR-1, denoted as SparseArch1. As stated in the analysis of CCR-1, if a kernel is sparse, we only need to record the nonzero values and their coordinates. Thus the kernels are stored in a compressed format, i.e., only the nonzero values and their coordinates appear in KB, as shown in Fig. 5(b). The nonzero values are fed into the PU, while the corresponding coordinates are fetched into a Coordinate Computation Unit (CCU) to compute the coordinate of the output according to (4). The $P_x \times P_y$ products lies in a rectangle block in the output maps, the addressing pattern of OB is still regular and only one coordinate needs to be calculated, i.e., the coordinate of the first element. Then, the $P_x \times P_y$ products are delivered into the AccUnit indexed by the coordinates and are accumulated to the corresponding partial sums fetched from OB. Finally, the accumulated results are routed to OB by the crossbar. Since the sub-convolutions derived from CCR-1 remain the same input map size with the original convolution, the mapping



(a) Generic dense architecture

(b) SparseArch1 that supports CCR-1

(c) SparseArch2 that supports CCR-2

(d) SparseArch3 that supports CCR-3

Fig. 5. Sparse architectures supporting CCR.

of the sub-convolutions onto the architecture exhibits little difference with the mapping to the dense architecture [14].

Compared with the dense architecture, the coordinates are not derived from a state machine but a dedicated CCU. The CCU only needs to calculate one coordinate for $P_x \times P_y$ products and can be implemented at low overhead. Accordingly, SparseArch1 efficiently supports sparse convolutions, obviating intrusive PE modifications to the dense architecture.

The architecture supporting CCR-2 (SparseArch2) is derived following the similar rationale, as shown in Fig. 5(c). To support CCR-3, the SparseArch3 architecture is more complicated with higher overhead than SparseArch1 and SparseArch2. Since the sub-convolutions derived from CCR-3 differ in both imap size and kernel size, the complexity significantly increases when mapping them to the same hardware. A feasible solution is that the imaps and kernels are all partitioned into sub-imaps and sub-kernels of 1×1 shape. The convolution with a 1×1 imap and 1×1 kernel thus equals to the scalar multiplication. As every scalar in $\mathcal{I}$ (a 1×1 imap) will be multiplied by every scalar in $\mathcal{K}$ (a 1×1 kernel), the original convolution then, is transformed to $Cartesian\ product$ of vectors. The CCU has to calculate $P_x \times P_y$ coordinates for each $P_x \times P_y$ product since the coordinates of the products are randomly distributed, hence the complexity for its CCU is $O(P_x \times P_y)$. Notably, the randomness of the output coordinates will cause bank contention in AccUnit and the Crossbar since multiple partial sums may hash to the same buffer banks.

Table 1 summarizes the characteristics of these architectures. Clearly, SparseArch3 eliminates the most ineffective computations. However, the hardware design is the most

TABLE 1
The sparse architecture design specifications.

Architectures	DA†	SA1†	SA2	SA3
IB format	uncomp‡	uncomp	comp‡	comp
KB format	uncomp	comp	uncomp.	comp
OB format	uncomp	uncomp	uncomp	uncomp
Read from IB	matrix	matrix	scalar	vector
Read from KB	scalar	scalar	matrix	vector
Computation in PU	$scalar \times mat$	$scalar \times mat$	$scalar \times mat$	$Cartesian\ Product$
CCU complexity	-	$O(1)$	$O(1)$	$O(P_x \times P_y)$

† "DA": short for Dense Architectures, "SA": short for Sparse Architectures
‡ uncomp: short for "uncompressed", comp: short for "compressed"

complicated. Moreover, SparseArch3 has a poor compatibility with the dense models since the Cartesian Product cannot well match the dense convolutions. Although unable to remove the most ineffective computations, SparseArch1 is remarkably efficient and economic since it achieves a comparably high computation reduction rate at a lower overhead than SparseArch3. Therefore, we opted for SparseArch1 as the architecture to efficiently cope with not only dense but also sparse CNN models.

5 SqueezeFlow Accelerator Architecture

This section presents a dataflow accelerator based on CCR-1, called SqueezeFlow. We first describe the PT-OS-sparse dataflow which employs CCR-1 as its inner core, followed by the full SqueezeFlow accelerator that employs such dataflow.

5.1 PT-OS-sparse Dataflow

A complete CNN dataflow requires a deep nested loop structure, and defines how the loops are ordered, partitioned, and parallelized [21]. Similar to dense architectures, sparse architectures also employ specialized dataflows to optimize performance and energy efficiency. We present a specific dataflow PT-OS-sparse. This sparse dataflow enables input data reuse while minimizing the psum accumulation cost. The implementation of the PT-OS-sparse dataflow is inspired by the idea of CCR that decomposes sparse convolutions into dense sub-convolutions. The dense sub-convolutions are mapped to PEs using dense dataflows, which have been extensively in prior works.

Fig. 6 shows the loop nest in the PT-OS-sparse dataflow. Note that the output and input channel (K and C) dimension has already been factored into K/T_k and C/T_c output/input channel groups. Such loop tiling increases data reuse and improves energy efficiency. The pseudo code only shows the inner loops of the complete dataflow for brevity, the omitted outer loops can be spatially and/or temporally spread across the PEs, which have been extensively studied in prior works. We focus on the inner loops to demonstrate how this dataflow leverages CCR-1 to exploit sparsity.

We employ a spatial tiling strategy to spread the work across an array of PEs so that each PE operates independently. The $W \times H$ element activation plane is partitioned into

```
    BUFFER wt_buf[Tc][Tk][V];
    BUFFER wt_idx_buf[Tc][Tk][V];
    BUFFER in_buf[Tc][W][H];
    BUFFER out_buf[Tk][W+R-1][H+S-1];
(A) for k = 0 to Tk-1
    {
(B)   for c = 0 to Tc-1
      {
(C)     for w' = 0 to W/Tw-1
        {
(D)       for h' = 0 to H/Th-1
          {
            for v = 0 to V-1
            {
(E)           wt = wt_buf[c][k][v];
(F)           wt_idx = wt_idx_buf[c][k][v];
(G)           parallel_for (w = 0 to Tw-1) x (h = 0 to Th-1)
              {
                x = Xcoord(w', w, v, wt_idx);
                y = Ycoord(h', h, v, wt_idx);
(H)             out_buf[k][w'*Tw+w][h'*Th+h] += in[c][x][y]*wt;
    } } } } } }
```

Fig. 6. PT-OS-sparse dataflow.

smaller $Tw \times Th$ element planar tiles (PT) that are distributed across the PEs. We employ an output-stationary (OS) computation order in which the psums are held stationary at the computation units for accumulation to minimize the psum accumulation cost. In the PT-OS-sparse dataflow, the sparse convolutions are decomposed into matrix scalar products, which is considered as convolutions with kernel size equal to one. Therefore, from the perspective of CCR-1, the kernels are partitioned into multiple sub-kernels of size 1×1.

The PT-OS-sparse dataflow contains looping over the K and C dimension (A,B), blocking in the W and H dimension (C,D), fetching scalar of weights (E) and the corresponding index (F), computing the coordinates of activations, fetching the matrix of activations according to the coordinates, computing the matrix scalar product in parallel (G,H). The $Xcoord()$, $Ycoord()$ functions compute the x, y coordinate using the temporal loop indices w', v, the spatial loop indices w, the RLC index w_idx, and the known kernel width and height.

For easier decoding, the weights are grouped into compressed blocks, with one $Tc \times Tk \times R \times S$ weights encoded into one compressed block, and we denote the length of each compressed $R \times S$ block as V, since the 2-dimension feature map will be compressed into 1-dimension vector. By contrast, the input activations are stored in uncompressed format, and the size of each block is $Tc \times W \times H$. At each access, the weight buffer delivers a scalar of nonzero filter weight along with its coordinate in the $Tc \times Tk \times R \times S$ region. Similarly, the input buffer delivers a matrix of $Tw \times Th$ input activations. The coordinates of this matrix block is computed using $Xcoord$ and $Ycoord$ functions. Then, the multiplier array computes the matrix scalar product to produce the $Tw \times Th$ partial sums, corresponding to the output block determined by the loop indices w' and h'. Unlike prior Cartesian Product based dataflows such as SCNN, the multiplier outputs in PT-OS-sparse are still contiguous as they correspond to a matrix block in output feature maps, which significantly reduces the complexity when scattering the output psums into the output buffers (will be discussed in Section 5.4). Since we use output-stationary computation order, the output matrices are natively accumulated in the PE array. Meanwhile, the loop indices w, h and coordinate computations ensure that these output matrices are concatenated correctly to generate the final output results. When the computation for the output channel group has been completed, the output buffer will be compressed and written back to DRAM.

5.2 Overview of SqueezeFlow Architecture

Fig. 7 demonstrates the full SqueezeFlow accelerator architecture based on PT-OS-sparse dataflow. We call it SqueezeFlow because it is a dataflow architecture that "squeezes" dense CNNs from sparse CNNs. SqueezeFlow consists of an accelerator chip and off-chip memory (usually DRAM). The accelerator chip is primarily composed of Processing Unit (PU) and a global buffer (GLB), two decoders, an encoder, a Control Unit (CU), a Buffer Controller (BC), a Coordinate Computation Unit (CCU), and a Post-Processing Unit (PPU). The PU contains a PE array which are connected with each other via a network-on-chip (NoC) [46] and support high compute parallelism to perform the massive convolution operations. The PPU is responsible to apply the accumulation (the AccUnit is integrated into PPU), non-linear activation (e.g. ReLU), pooling, and dropout functions. The accelerator provides four levels of storage, including DRAM, GLB, inter-PE connections and Register Files in each PE. GLB is used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. The Encoder/Decoder is used to reduce DRAM accesses, so that the data transferred between on-chip GLB and off-chip DRAM are all in compressed format. By doing so, the off-chip memory accesses of the accelerator are largely reduced.

To accomplish a CVL computation, the activations and weights are brought from DRAM into GLB, both are in compressed format. Then the compressed weights are fed into the KB directly. The activations, however, are decompressed by the decoder and fed into IB. Then, the BC selects needed activations and weights from IB and KB respectively, based on the loaded instructions in the CU, and transfers those activations and weights to PU to perform sparse convolutions following PT-OS-sparse dataflow. The non-linear activation, pooling or dropout functions will be performed by PPU if necessary. When executing a CVL composed of multiple output feature maps with multiple input feature maps, the accelerator continuously performs the computations of an output feature map, and will not move to the next output feature map until the current map has been constructed. The rest of this section describes the design details of SqueezeFlow.

5.3 Processing Unit

Fig. 7 also presents the PU structure. The PU operates the weights and activations in the order defined by the PT-OS-sparse dataflow to produce the psums. The PU simultaneously read weights and input activations fetched from BC, and then distribute them to different PEs. In addition, the PU contains local storage structures in each PE, which is used to enable local propagation of input activations between PEs (see Section 5.5) to reduce the data transfer between PEs and the GLB. After performing the multiplication between activations and weights, the PU delivers the partial products into PPU. PPU collects these products and accumulate them to generate the final results.

Processing elements. Fig. 7 also shows the PE micro-architecture. Each PE executes fixed multiply-accumulation

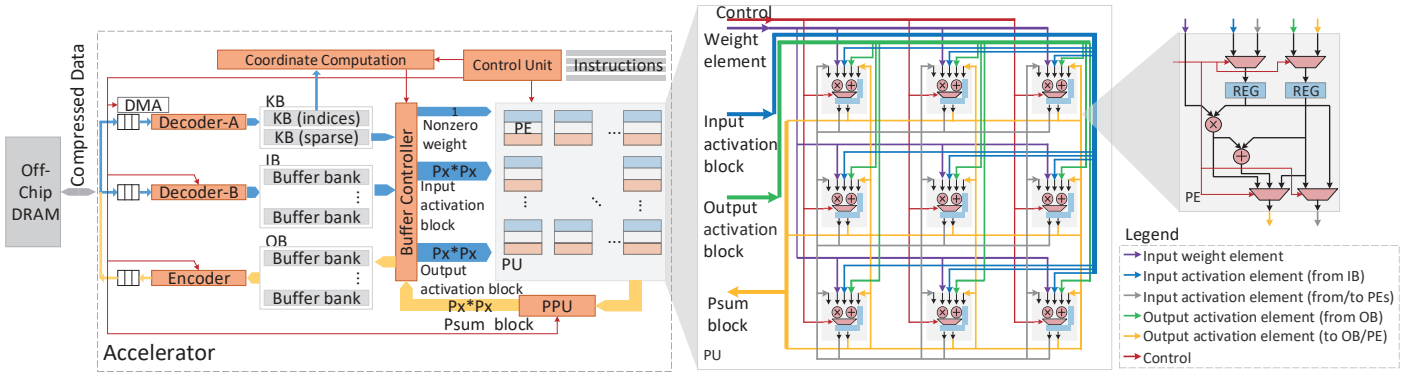


Fig. 7. Complete SqueezeFlow accelerator architecture.

operations. $PE_{i,j}$, which is the PE at the i-th row and j-th column of the PU, has three inputs: one input for receiving the control signals; one input for reading weights (e.g. kernel values of CVLs) from KB; and one input for reading activations from IB/OB, or from other PEs depending on the control signal. The $PE_{i,j}$ has two outputs: one output for writing computation results to OB/IB; one output for propagating locally-stored activations to other PEs to efficiently reuse the data. When executing a CVL, each PE continuously accommodates a single output activation, and will not switch to another output activation until the current one has been computed. By distributing the nonzero kernel element to all the PEs, the PEs will compute different activations of the same output feature map simultaneously.

Inter-PE propagation. CCR implies that each matrix scalar product requires data from a rectangular window of input maps, and the results also lie in a rectangular window of the output maps. In dense convolutions, such windows are significantly overlapping for adjacent activations, because of the sliding window characteristics of convolution. Although all required data are available from IB/OB, repeatedly reading them from the buffer to different PEs requires a high bandwidth. Therefore, inter-PE propagation is used to support efficient data reuse [14].

In sparse convolutions, inter-PE propagation can also be used to enable data reuse. However, because the overlapping pattern is randomized due to random distribution of zeros, only enabling the PEs to send locally-stored input activations to its left and lower neighbors (like dense architectures) is not efficient. We have to enable that the PEs can send the locally-stored input activation to any PE. Thus, we implement a Network-on-Chip to connect these PEs using fat tree architectures, which supports the data transfer among the PEs. Note that although the reuse pattern is randomized, it is deterministic given the overlapping pattern between the adjacent input activation blocks. Hence, the CU generates the control signals according to the overlapping pattern to determine how the NoC propagates the data among the PEs. Furthermore, unlike ShiDianNao that leverages FIFO structures, we only use a register in each PE for inter-PE propagation to reduce the complexity and area of PEs (see Section 5.5).

5.4 Coordinate Computation Unit

In dense accelerators, the addresses of the outputs are usually generated from a finite state machine (FSM), because of the regular data access patterns of dense convolutions. However, a simple FSM can hardly support address generation for sparse accelerators. Hence, we design a dedicated CCU for address generation. Since coordinate computation is tightly related to the compression techniques, we firstly introduce how we compress activations and weights using Run Length Compression (RLC). It should be noted that because the selection of specific format is orthogonal to the sparse architecture itself, we can also use other data compression techniques to construct SqueezeFlow, as long as its decoding can yield nonzero values and the corresponding coordinate.

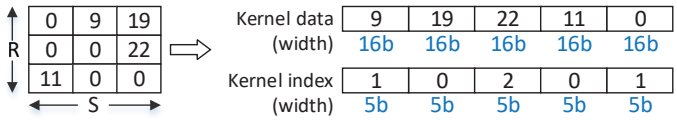


Fig. 8. Run-length encoding for sparse kernels.

Fig. 8 presents an example of RLC. It records the nonzero values and the count of zeros between adjacent nonzero values. There are two parts in RLC encoding, "Run" and "Length", storing the nonzero values and the zero counts between adjacent nonzero elements, respectively. The weights are compressed in RLC format and stored in KB.

Firstly, CCU calculates the original coordinates of nonzero weights in a recursive fashion, i.e., the coordinate of the current nonzero weight is computed using its distance to the prior nonzero element (the number of zeros) and the dividing by the W dimension. For example, for weights $(\cdots, R_{i-1}, L_{i-1}, R_i, L_i)$ store in KB, suppose that the coordinate of R_{i-1} is known and denoted as $(\alpha_{i-1}, \beta_{i-1})$, since there are L_{i-1} zeros between R_{i-1} and R_i, the coordinate of R_i is derived as follows:

$$\begin{aligned} \alpha_i &= \alpha_{i-1} + (\beta_{i-1} + L_{i-1} + 1) \;/\; W \\ \beta_i &= (\beta_{i-1} + L_{i-1} + 1) \; mod \; W \end{aligned} \tag{10}$$

Secondly, the CCU computes the coordinates of the input activation matrix as we use an output-stationary dataflow. Given the coordinates of the output activation matrix (loop indices w and h) and the nonzero weight, the coordinates of the input activations is easily calculated according to the convolution characteristics. Because the coordinates of the input activations are still contiguous in W/H dimension according to CCR, CCU only computes the coordinate of the first input

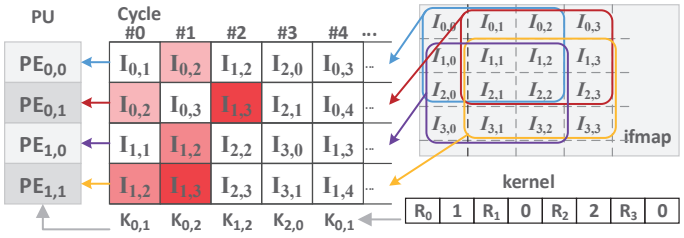


Fig. 9. Algorithm-hardware mapping of convolution operations. The red marks indicate that the input activations are reused by different PEs.

activation using the coordinates of weight and the output activations (loop indices w and h). The coordinates of the rest input activations are generated based on the first coordinate and W/H dimension.

5.5 Sparse CNN Mapping

In this subsection, we show in detail how SqueezeFlow supports to remove the ineffective computations in sparse CVLs. In Fig. 9, we consider a snapshot of the sparse convolution processing. Without losing any generality, we consider a small design having 2×2 PEs ($PE_{i,j}$ indicating the PE located at the i_{th} row and j_{th} column), and a CVL with 3×3 kernel size with four nonzero values, and 1×1 stride size. For the sake of brevity, we describe the first five cycles to analyze how the accelerator works cycle by cycle, as demonstrated in Fig. 10.

Cycle #0: Each PE processes the partial products of the first block in the output feature map, i.e., $O_{0,0}$, $O_{0,1}$, $O_{1,0}$ and $O_{1,1}$. All four PEs share the same weight from KB, or the first nonzero element in the convolution kernel, i.e., R_0. Since it is the first cycle, the coordinate of the prior nonzero weight is initialized as $(0, -1)$. According to (10), the coordinate of R_0 is calculated as follows:

$$\begin{aligned} x_{wt} &= \alpha_{i-1} + (\beta_{i-1} + L_{i-1} + 1) \;/\; W = 0 \\ y_{wt} &= (\beta_{i-1} + L_{i-1} + 1) \; mod \; W = 1 \end{aligned} \tag{11}$$

Hence, the coordinate of R_0 is (0,1). Then we calculate the coordinate of the first element in the input activation block using the coordinate of output activation and weight as follows:

$$\begin{aligned} x_{in} &= x_{out} + x_{wt} = 0 + 0 = 0 \\ y_{in} &= y_{out} + y_{wt} = 0 + 1 = 1 \end{aligned} \tag{12}$$

where x_{out}/y_{out} are the coordinates of the first output activation in the matrix block, i.e., (0,0). So the coordinate of the first element in the input activation block is $I_{0,1}$. Because the coordinates of the input activations are still contiguous in W/H dimension, the coordinates of the rest input activations are easily calculated, i.e., $I_{0,2}$, $I_{1,1}$ and $I_{1,2}$, which will be fetched into the four PEs. Each PE performs a multiplication between the received input activation and weight, and store the result in its local register, i.e., $psum\_new$ register. Then, the result will be accumulated with the data stored in $psum\_old$ register, which holds the psum value derived from previous activations and weights. The accumulated result will be stored locally in the $psum\_old$ register for future accumulation. In addition, each PE collects its received input activation in its $input\_reg$ for future inter-PE propagation.

Cycle #1: All four PEs share the second nonzero weight, i.e., R_1. Similar to Cycle #0, CCU computes its coordinate as follows:

$$\begin{aligned} x_{wt} &= \alpha_{i-1} + (\beta_{i-1} + L_{i-1} + 1) \;/\; W = 0 \\ y_{wt} &= (\beta_{i-1} + L_{i-1} + 1) \; mod \; W = 2 \end{aligned} \tag{13}$$

Then, the coordinate of the first element in the input activation block is $x_{in} = 0 + 0 = 0, y_{in} = 0 + 2 = 2$. The four PEs respectively read their required input activations ($I_{0,2}$, $I_{0,3}$, $I_{1,2}$ and $I_{1,3}$). Because $I_{0,2}$ and $I_{1,2}$ are already stored in the $input\_reg$ of $PE_{0,1}$ and $PE_{1,1}$ respectively, $PE_{0,0}$ and $PE_{1,0}$ enable inter-PE propagation by reading the corresponding input activations from the $input\_regs$ of their right PEs, while $PE_{0,1}$ and $PE_{1,1}$ read their input activations from IB. Then the multiplication is performed in each PE and the results and input activations will be locally stored in the corresponding registers. So far each PE has processed the first row of the convolutional window, and will move to the second row of the convolutional window in the next cycle.

Cycle #2: Similar to Cycle #1, the coordinate of the third nonzero weight is calculated, i.e., (1,2). The four PEs respectively read their required input activations according to the index of the weight ($I_{1,2}$, $I_{1,3}$, $I_{2,2}$ and $I_{2,3}$). Because $I_{1,3}$ is already stored in the $input\_reg$ of $PE_{1,1}$, $PE_{0,1}$ enables inter-PE propagation by reading the corresponding input activations from the $input\_regs$ of its bottom PEs. Meanwhile, the other PEs read their input activations from IB. The PEs perform the multiplications and the results and input activations will be locally stored in the corresponding registers.

Cycle #3: Similar to Cycle #1, all four PEs share the fourth nonzero weight, i.e., $K_{2,0}$, and perform similar operations. So far each PE has processed the partial products of the first block in the output feature map, i.e., $O_{0,0}$, $O_{0,1}$, $O_{1,0}$ and $O_{1,1}$, and will move to the second block of the convolutional window in the next cycle. Hence, in this cycle, the partial products stored in $psum\_old$ registers will be written back to OB.

Cycle #4: The processing of the second block of the convolutional windows begins at this cycle, i.e., $O_{0,2}$, $O_{0,3}$, $O_{1,2}$ and $O_{1,3}$. All four PEs share the first nonzero weight from KB, i.e., $K_{0,1}$ (the coordinate calculation is the same to Cycle #0). Then, the coordinate of the first element in the input activation block is $x_{in} = 0 + 0 = 0, y_{in} = 2 + 1 = 3$. The coordinates of the rest input activations is easily calculated, i.e., $I_{0,4}$, $I_{1,3}$ and $I_{1,4}$. The following operations are similar and omitted for the sake of brevity.

So far we described the first five cycles when mapping sparse CVLs to SqueezeFlow. This example also clarifies that why FIFO structures are not used for inter-PE propagation. Firstly, using FIFOs will significantly increase the area overhead of PEs. Secondly, because nonzero weights are randomly distributed, the data access of input activations is not so regular as in dense CNNs, using FIFOs does little help as it does not match the access patterns of sparse convolutions.

6 Experimental Methodology

Implementation. We implement SqueezeFlow in Synopsys design flow on TSMC 65 nm technology: simulating with Synopsys Verilog Compile Simulator (VCS), synthesizing with Synopsys Design Compiler (DC), analyzing power with Synopsys PrimeTime (PT), and placing them with Synopsys IC Compiler (ICC). We used CACTI 6.0 to estimate the energy

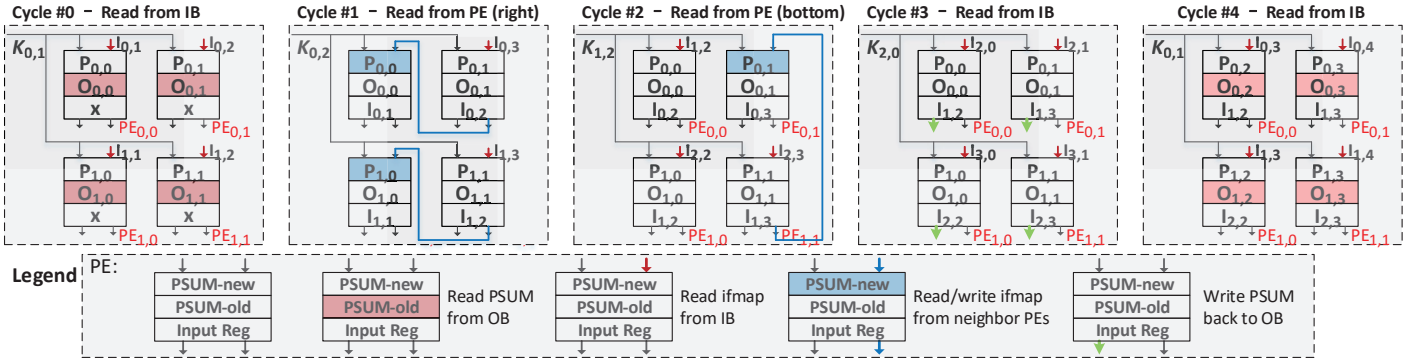


Fig. 10. Algorithm-hardware mapping of convolution operations. The red marks indicate that the psum has to initialized using data fetched from OB, otherwise using the value stored in the register as we use output-stationary dataflow. The blue marks indicate the inter-PE propagation.

cost of DRAM accesses [47]. We also resize and re-implement a dense accelerator, called DenseArch, following the rationale described in Section 4.

Baselines. We compare our design with the DenseArch and three representative sparse architectures: Cambricon-X [23], Cnvlutin [22] and SCNN [24]. We developed a custom-build cycle-level simulator to evaluate the performance and energy of the three baselines (Cambricon-X, Cnvlutin, SCNN) since we did not implement them with RTL configuration in Verilog. The simulator models their dataflow as well as the memory hierarchy and PE configurations. The simulator evaluates the performance by computing the number of cycles to process a given layer by the three accelerators. Meanwhile, the simulator collects the counts of MAC operations and memory accesses of different levels. These statistical data are used to build an energy model to estimate the energy consumption of the three baseline accelerators. Because of significant differences in dataflow, buffer sizing/organization, and implementation choices, our evaluated architectures cannot precisely represent the prior proposals.

Architectural configurations. SqueezeFlow is equipped with a PU containing an 8×8 array of PEs. IB and OB SRAM size are 64 KB, while KB SRAM size is 84 KB, containing 64 KB to store nonzero values and additional 20 KB to store the indices of sparse kernels. We use 16-bit fixed-point arithmetic units as it has been proved to be effective in CNN accelerators [12], [13]. To make a fair comparison, we also resize the baseline architecture to be equipped with the same number of multipliers with SqueezeFlow. Additionally, the working frequency of SqueezeFlow and the baselines are kept the same at 0.9 GHz. The nominal DRAM bandwidth configuration is 34.2 GB/s as we use dual-channel DDR3-2133.

Benchmarks. We benchmark the performance using three representative CNNs: VGG16 [48], AlexNet [1], GoogLeNet [49]. Note that for GoogLeNet, we primarily focus on the CVLs in the *inception* modules. These models provide a wide range of shapes and sparsity that are suitable for testing the adaptability and flexibility of our accelerator.

7 Experimental Results

This section compares SqueezeFlow with the baselines in terms of area, performance, power, energy and scalability.

TABLE 2
Parameter settings of DenseArch and SqueezeFlow

Accelerator	#MULs	IB size	OB size	KB size
DenseArch	64	64 KB	64 KB	64 KB
SqueezeFlow	64	64 KB	64 KB	KB (sparse): 64 KB, KB (indices): 20 KB

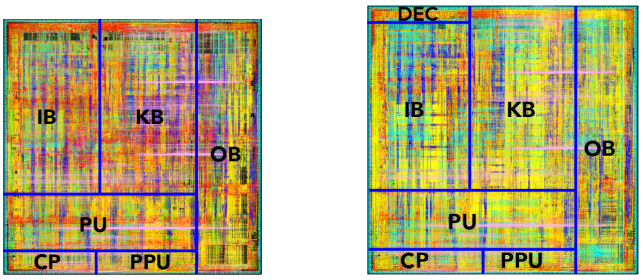
TABLE 3
Area and power comparison between DenseArch and SqueezeFlow

	DenseArch		SqueezeFlow	
	Area(mm^2)	Power(mW)	Area(mm^2)	Power(mW)
Total	4.41(100%)	464.82(100%)	4.80(100%)	536.09(100%)
PU	0.78(17.69%)	268.82(57.83%)	0.80(16.67%)	280.00(52.23%)
IB	1.12(25.40%)	55.53(11.95%)	1.12(23.33%)	60.18(11.23%)
OB	1.12(25.40%)	30.60(6.58%)	1.12(23.33%)	36.27(6.77%)
KB	1.12(25.40%)	48.60(10.46%)	1.28(26.67%)	58.92(10.99%)
CP	0.11(2.49%)	40.81(8.78%)	0.20(4.17%)	62.14(11.59%)
PPU	0.16(3.63%)	20.46(4.40%)	0.16(3.33%)	22.15(4.13%)
DEC	-	-	0.12(2.50%)	16.43(3.06%)

7.1 Layout Characteristics

Table 2 and Table 3 presents the parameter settings and layout characteristics of the baseline DenseArch and SqueezeFlow (see Fig. 11). Under the same computing resources and buffer size, SqueezeFlow increases total area by about 8.8% over DenseArch, with 4.80 mm^2 vs. 4.41 mm^2. Area compares across the two architectures as follows: 1) The buffers (IB, KB, OB) dominate the total area for both architectures, i.e., 76.2% and 73.3% for DenseArch and SqueezeFlow, respectively; 2) the area of PU remain almost the same for both architectures, since CCR enables to support sparse convolutions almost without intrusive PE modifications; 3) the main area overhead for SqueezeFlow stems from KB (increased 0.16 mm^2 to store indices of sparse kernels), CP (increased 0.09 mm^2 mainly because of CCU), DEC (Encoder/Decoder, 0.12 mm^2). Overall, SqueezeFlow only incurs a slight area overhead in order to efficiently support sparse CNNs.

The total power of SqueezeFlow is only 536.09 mW, which is 15.3% higher than DenseArch with 464.82 mW (averaged over the three benchmarks). The reason consists of two aspects. Firstly, the additional hardware logic to exploit sparsity increases power by about 37.76 mW, including the CP (increases 21.33 mW as it has integrated a Coordinate



(a) DenseArch (b) SqueezeFlow

Fig. 11. Layout of DenseArch and SqueezeFlow.

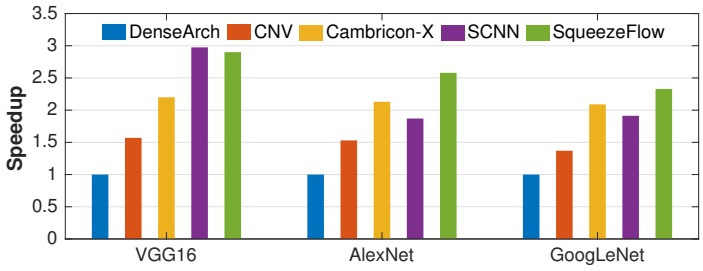


Fig. 12. Speedup over the baseline dense architecture across the models.

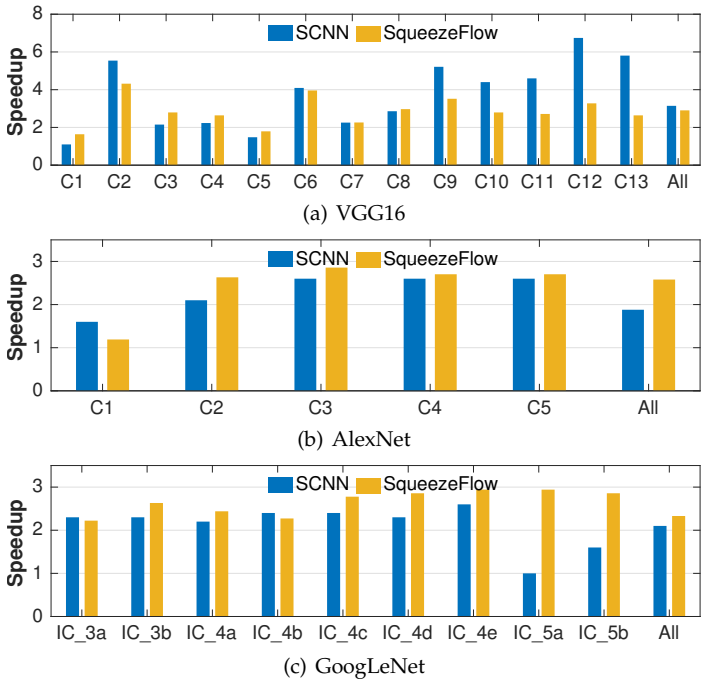


Fig. 13. Speedup over the baseline dense architecture across the models.

Computation Unit) and DEC (16.43 mW). Secondly, the on-chip buffer increases the power by 20.64 mW. The larger buffer size to hold the indices and the relatively randomized accesses of these buffers leads to 8.3%, 18.5%, and 21.2% increment of the power for IB/OB/KB, respectively.

7.2 Performance

Fig. 12 summarizes the speedups delivered by the sparse architectures over the baseline DenseArch. SqueezeFlow consistently outperforms the baselines (except for SCNN on VGG16) and achieves an average speedup of $2.6\times, 1.8\times, 1.3\times, 1.2\times$ over the baselines, respectively. The performance improvement of SqueezeFlow varies widely across the models. Specifically, SqueezeFlow improves the performance by 2.3–2.9$\times$ over DenseArch, 1.6–1.8$\times$ over Cnvlutin, 1.2–1.4$\times$ over Cambricon-X, 0.9–1.4$\times$ over SCNN. Although SCNN can theoretically achieve the highest performance for all models since it removes the most ineffective computations stemming from zeros in both kernels and imaps, it delivers a slight performance advantage over SqueezeFlow only on VGG16 but performed much worse than expected on AlexNet and GoogLeNet. The main reason is that SCNN incurs severe performance degradation from the unbalanced distribution of computations among the PEs. The results reveal that although SqueezeFlow is unable to remove all the ineffective computations, it is remarkably efficient across various models.

The performance results are better understood by looking at the performance breakdown of SqueezeFlow and SCNN across the CVLs as shown in Fig. 13. We take VGG16 as an example for further explanation. For bottom layers like C1 and C3, SqueezeFlow achieves a superior performance to SCNN. One reason is that the sparsity of kernels dominates these layers which well matches the advantages of SqueezeFlow, while input maps are not that sparse (only 5% sparsity in C1). As the sparsity increases with the layers going deeper, SCNN achieves an overwhelming performance over SqueezeFlow since imaps and kernels are both very sparse in the later layers. However, as the computation volume is mainly dominated by the bottom layers (larger input/output maps), SqueezeFlow achieves a comparable overall performance with SCNN due to the superior performance in the bottom layers.

For the other two benchmarks, SqueezeFlow significantly outperforms SCNN because SCNN hardly reaches its nominal performance. Although SCNN considers both activation and weight sparsity, it incurs significant performance degradation mainly because of load unbalancing since SCNN exploits *Cartesian Products* to perform convolution. However, as SqueezeFlow uses matrix scalar product, the load balancing problem is natively addressed under PT-OS-sparse dataflow.

It should be noted that the stride is four in C1 of AlexNet. In this layer, we first compute the corresponding convolution with the same input feature maps and filter weights but stride equal to one. Then, we extract the valid data from it using the stride (=4) to generate the final result, which is similar to the operations in pooling layer. Such implementation will increase the computation volume as it introduces unnecessary computations, but it can be easily implemented under existing dataflow. Furthermore, as such layer type is not common in the benchmarks, it impacts little on the overall performance.

Impact of DRAM Bandwidth. As DRAM bandwidth affects the latency of off-chip memory accesses, we tested the performance of SqueezeFlow under different bandwidth configurations through simulation. We observed that the performance of SqueezeFlow begins to degrade when the DRAM bandwidth drops to 6 GB/s. Since the nominal DRAM bandwidth configuration is 34.2 GB/s, it provides ample bandwidth to absorb the off-chip traffic.

In summary, SqueezeFlow achieves almost the highest speedup on average across the models and provides a tremendous performance advantage over the baselines.

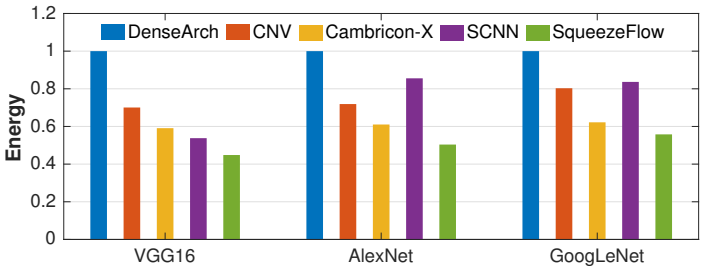


Fig. 14. Energy consumption normalized to the dense architecture.

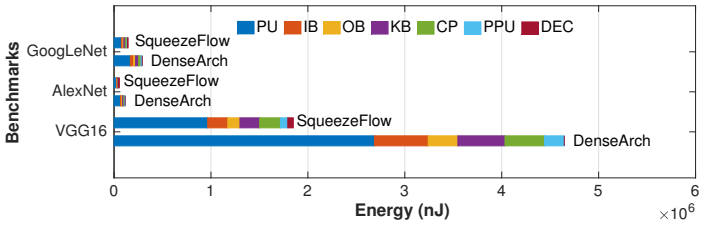


Fig. 15. Energy breakdown of DenseArch and SqueezeFlow.

7.3 Energy

In Fig. 14, we report the energy comparison of the architectures which has been normalized to the energy of DenseArch. It should be noted that the energy consumption does not include main memory accesses which usually dominates the total energy consumption. On average, SqueezeFlow improves energy efficiency by $2.0\times, 1.5\times, 1.3\times, 1.5\times$ over the baselines, respectively. The most striking result is that SqueezeFlow achieves an improvement of $2.23\times$ over DenseArch on VGG16. The improvement of energy efficiency varies widely across the models depending on the sparsity of the models. Specifically, the improvement over SCNN ranges from $1.7\times$ on AlexNet, to $1.2\times$ on VGG16. The high energy efficiency of SqueezeFlow stems from: 1) the improvement of performance, SqueezeFlow achieves a high computation reduction rate and free from performance degradation caused by irregular computation distributions; 2) low hardware overhead, the architecture of SqueezeFlow is much simpler than SCNN, thus it is implemented with lower power.

We further show the energy breakdown of our accelerator and DenseArch in Fig. 15. It is clear that PU consumes the most energy in both DenseArch and SqueezeFlow. PU consumes over 50% of the total energy in DenseArch, which is consistent with the results reported in prior work [14]. Because of the high performance of SqueezeFlow, the energy consumption of all the components in SqueezeFlow is lower than that in DenseArch.

7.4 Scalability and Compatibility

We evaluate the scalability of the architectures from the sensitivity to the scale of PE array to study the hardware scale-out merit. Fig. 16 compares the performance of the architectures on VGG16. With the scaling up of PE array, SqueezeFlow maintains a stable high performance. Because the imap size is usually larger than PE array size, SqueezeFlow maintains a high PE utilization. Cnvlutin and Cambricon-X suffer from a slow performance degradation with the increasing of the PEs, while SCNN incurs a much more severe performance

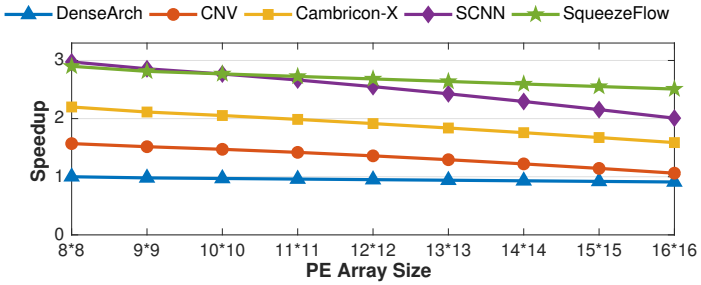


Fig. 16. Speedup on VGG16 for different scales of PE array.

TABLE 4
Speedup on dense and sparse VGG16 over DenseArch

	CNV	Cambricon-X	SCNN	SqueezeFlow
Sparse VGG16	1.6×	2.2×	3.0×	2.9×
Dense VGG16	1×	1×	0.8×	1×

degradation. When PE array is larger than 9×9, SqueezeFlow surpasses SCNN for the performance on VGG16. The result highlights that SqueezeFlow has good scalability.

We also evaluate the compatibility of these architectures by testing their performance on both dense and sparse VGG16, as listed in Table 4. SqueezeFlow achieves a comparable performance with DenseArch although it is not dedicated for dense models. However, SCNN incurs 20% performance degradation compared to DenseArch, which reveals that SqueezeFlow has better compatibility than SCNN.

8 Discussion

We have described the SqueezeFlow accelerator to exploit sparsity for increased efficiency. However, there are several limitations that prevent SqueezeFlow from optimal efficiency. The first limitation is that SqueezeFlow only removes the ineffective computations from zero weights, based on the observation that weight sparsity is usually higher than activation sparsity. It would be better if the accelerator can adaptively remove ineffective computations from zero weights or activations. The second limitation arises from the lack of efficient support for various convolution stride. SqueezeFlow handles this condition by introducing unnecessary computations. Although it is a common method and widely used in prior work, it would lead to better efficiency if there is dedicated support for various convolution strides, which might be a future work.

9 Conclusion

This paper firstly describes concise convolution rules which can smartly decomposes sparse convolutions into effective and ineffective sub-convolutions. The computations in ineffective sub-convolutions are eliminated while the effective ones can be easily mapped to existing dense CNN accelerators without intrusive PE modifications. Based on CCR, we propose SqueezeFlow accelerator architecture to exploit sparsity of CNNs. SqueezeFlow provides a tremendous performance and energy efficiency advantage over prior approaches. With a footprint of 4.80 mm^2 and 536.09 mW, SqueezeFlow achieves a speedup of $2.9\times$ on VGG16 over a comparably provisioned dense architecture. Compared with state-of-the-art

sparse accelerators, SqueezeFlow improves the performance and energy efficiency by $1.8\times$ and $1.5\times$, respectively.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61872336, 61532017, 61572470, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No. Y404441000. An earlier version of this work has been published in the Proceedings of 2018 Design, Automation & Test in Europe (DATE) [27]. This extended research presents a new accelerator architecture and new experimental verifications. The authors appreciate the constructive comments provided by the anonymous reviewers. A special thank you goes to Mr. Zou Dianzhe for his encouragement and support.

REFERENCES

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

[2] P. Simard, L. Bottou, P. Haffner, and Y. LeCun, "Boxlets: a fast convolution algorithm for signal processing and neural networks," in *Advances in Neural Information Processing Systems*, pp. 571–577, 1999.

[3] Y. Deng, F. Bao, Q. Dai, L. F. Wu, and S. J. Altschuler, "Scalable analysis of cell-type composition from single-cell transcriptomics using deep recurrent learning," *Nature Methods*, vol. 16, no. 4, pp. 311–314, 2019.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[5] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE Transactions on Neural Networks*, vol. 28, no. 3, pp. 653–664, 2017.

[6] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *international symposium on computer architecture*, vol. 38, no. 3, pp. 37–47, 2010.

[7] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *international joint conference on artificial intelligence*, pp. 1237–1242, 2011.

[8] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 257–260, 2010.

[9] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," *international symposium on computer architecture*, vol. 41, no. 3, pp. 24–35, 2013.

[10] P. Ienne, T. Cornu, and G. Kuhn, "Special-purpose digital hardware for neural networks: an architectural survey," *J. VLSI Signal Process. Syst.*, vol. 13, no. 1, pp. 5–25, 1996.

[11] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 13–19, IEEE, 2013.

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," *architectural support for programming languages and operating systems*, vol. 49, no. 4, pp. 269–284, 2014.

[13] 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 *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, 2014.

[14] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *field programmable gate arrays*, pp. 161–170, 2015.

[16] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, and S. Song, "Going deeper with embedded fpga platform for convolutional neural network," in *field programmable gate arrays*, pp. 26–35, 2016.

[17] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. B. K. Vrudhula, J. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *field programmable gate arrays*, pp. 16–25, 2016.

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

[19] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*, pp. 240–241, IEEE, 2017.

[20] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 553–564, IEEE, 2017.

[21] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE, 2016.

[22] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 1–13, IEEE Press, 2016.

[23] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[24] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 27–40, ACM, 2017.

[25] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (Williamsburg, VA, USA), pp. 461–475, ACM, 2018.

[26] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 343–348, IEEE, 2018.

[27] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, "Ccr: A concise convolution rule for sparse neural network accelerators," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 189–194, IEEE, 2018.

[28] J. Li, G. Yan, W. Lu, S. Gong, S. Jiang, J. Wu, and X. Li, "Synergyflow: An elastic accelerator architecture supporting batch processing of large-scale deep neural networks," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, pp. 1–27, 2018.

[29] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, J. Yan, and X. Li, "Tnpu: an efficient accelerator architecture for training convolutional neural networks," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, (Tokyo, Japan), pp. 450–455, ACM, 2019.

[30] W. Lu, G. Yan, J. Li, S. Gong, S. Jiang, J. Wu, and X. Li, "Promoting the harmony between sparsity and regularity: A relaxed synchronous architecture for convolutional neural networks," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 867–881, 2019.

[31] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, pp. III–1337–III–1345, JMLR.org, 2013.

[32] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8595–8598, 2013.

[33] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Efficient learning of sparse representations with an energy-based model," in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, (Canada), pp. 1137–1144, MIT Press, 2006.

[34] H. Lee, C. Ekanadham, and A. Y. Ng, "Sparse deep belief net model for visual area v2," in *Proceedings of the 20th International Conference on*

Neural Information Processing Systems, (Vancouver, British Columbia, Canada), pp. 873–880, Curran Associates Inc., 2007.
[35] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, pp. 1135–1143, 2015.
[36] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
[37] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 535–547, IEEE, 2017.
[38] 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, 1996.
[39] 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," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press, 2016.
[40] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1927–1934, 2017.
[41] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 674–687, 2018.
[42] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Transactions on Algorithms (TALG)*, vol. 1, no. 1, pp. 2–13, 2005.
[43] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *ieee international conference on high performance computing data and analytics*, pp. 1–11, 2009.
[44] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, IEEE, 2007.
[45] G. Strang, "A proposal for toeplitz matrix calculations," *Studies in Applied Mathematics*, vol. 74, no. 2, pp. 171–176, 1986.
[46] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, (Seoul, Republic of Korea), pp. 1–8, ACM, 2017.
[47] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pp. 3–14, 2007.
[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *international conference on learning representations*, 2015.
[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Jiajun Li received the B.E. degree from the Department of Automation, Tsinghua University, Beijing, China, in 2013. He is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include machine learning and heterogeneous computer architecture.

Shuhao Jiang received the B.Eng. degree in electronic engineering from Tsinghua University in 2014. He is currently a Ph.D. Candidate at the State Key Lab. of Computing Technology(ICT), Chinese Academy of Science(CAS). His research interests include machine learning and approximate computing techniques.

Shijun Gong received the B.E. degree from the School of Computer Science & Engineering, University of Electronic Science and Technology of China, Chengdu, China, in 2015. He is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include machine learning and heterogeneous computer architecture.

Jingya Wu received the B.E. degree from the College of Information and Electrical Engineering, China Agricultural University , Beijing, China, in 2015. She is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. Her current research interests include machine learning and approximate computing.

Junchao Yan received the B.E. degree from Southwest Jiaotong University, Chengdu, China, in 2016. He is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include machine learning and time series analysis.

Guihai Yan (M'10) received the B.Sc. degree in Electronics and Software Engineering (dual-degree) from Peking University, Beijing, China, in 2005, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2011. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences. He was awarded CCF (China Computer Federation) Outstanding Doctoral Dissertation, and Chinese Academy of Sciences Outstanding Dotoral Dissertation. His research interests include computer architecture, domain-specific microsystems, and energy-efficient computing.

Xiaowei Li (SM'04) received the B.Eng. and M.Eng. degrees in Computer Science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in Computer Science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991. He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has co-authored over 280 papers in journals and international conferences, and he holds 60 patents and 30 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks. Dr. Li has been the Vice Chair of the IEEE Asia & Pacific Regional Test Technology Technical Council (TTTC) since 2004. He is currently Vice Chair of IEEE TTTC. He was the Chair of the Technical Committee on Fault Tolerant Computing, China Computer Federation (2008-2012), and Steering Committee Chair of IEEE Asian Test Symposium (2011-2013). He was Steering Committee Chair of IEEE Workshop on RTL and High Level Testing (2007-2010). He services as Associate Editor of JCST, JOLPE, JETTA and TCAD.