CCR: A Concise Convolution Rule for Sparse Neural Network Accelerators

Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, Xiaowei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
{lijiajun, yan, luwenyan, jiangshuhao, gongshijun, wujingya, lxw}@ict.ac.cn

***Abstract*—Convolutional Neural networks (CNNs) have achieved great success in a broad range of applications. As CNN-based methods are often both computation and memory intensive, sparse CNNs have emerged as an effective solution to reduce the amount of computation and memory accesses while maintaining the high accuracy. However, dense CNN accelerators can hardly benefit from the reduction of computations and memory accesses due to the lack of support for irregular and sparse models. This paper proposed a concise convolution rule (CCR) to diminish the gap between sparse CNNs and dense CNN accelerators. CCR transforms a sparse convolution into multiple effective and ineffective ones. The ineffective convolutions in which either the neurons or synapses are all zeros do not contribute to the final results and the computations and memory accesses can be eliminated. The effective convolutions in which both the neurons and synapses are dense can be easily mapped to the existing dense CNN accelerators. Unlike prior approaches which trade complexity for flexibility, CCR advocates a novel approach to reaping the benefits from the reduction of computation and memory accesses as well as the acceleration of the existing dense architectures without intrusive PE modifications. As a case study, we implemented a sparse CNN accelerator, SparseK, following the rationale of CCR. The experiments show that SparseK achieved a speedup of $2.9\times$ on VGG16 compared to a comparably provisioned dense architecture. Compared with state-of-the-art sparse accelerators, SparseK can improve the performance and energy efficiency by $1.8\times$ and $1.5\times$, respectively.**

I. Introduction

Convolutional Neural Networks (CNNs) have achieved a state-of-the-art performance across a broad range of applications [1]. To handle the intensive computation and memory accesses of CNNs, a number of accelerators [2]–[6] have been proposed that deliver high computational throughput. However, with the networks going larger and deeper to further yield higher accuracy, it remains a big challenge to efficiently process such networks on existing accelerators.

Sparse CNNs have emerged as an effective solution to address this challenge which reduce the amount of computation and memory accesses while maintaining the high accuracy. Previous studies [7], [8] have proven that a large fraction of neurons and synapses can be pruned to zero without loss of accuracy. As demonstrated in Fig. 1, by using the pruning techniques, more than 30% of the neurons and 50% of the synapses in VGG16 are zero values, and more than 65% of the computations are unnecessary because of zero operands.

Accelerators which are good at processing regular and dense CNNs can benefit little from the reduction of both computation and data amount. It will inevitably introduce

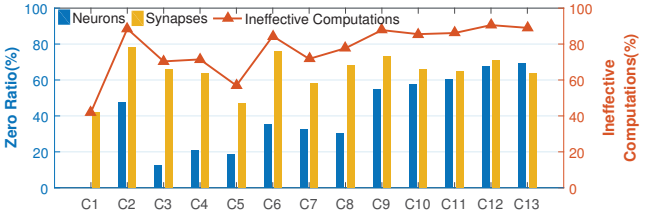

Fig. 1. The sparsity in the convolutional layers of VGG-16.

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 "dummy" computations which are essentially unnecessary. To overcome this problem, the prevailing method is to enable PEs with the flexibility to independently skip dummy computations [9]–[11]. However, it significantly increases the hardware complexity and thereby overhead. Moreover, these approaches incur performance degradation due to the unbalanced sparsity distribution among the PEs. It is observed that the attainable performance merely reaches up to 50% of the nominal performance. Unlike the prior approaches which trade complexity for flexibility, we take an alternative by smartly partitioning the computation workload. By doing so, the sparse CNNs can be easily mapped to existing dense CNN accelerators, obviating the intrusive PE modifications.

In this paper, we propose a concise convolution rule (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 neurons or synapses 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 neurons and synapses are dense can be easily mapped to the existing dense CNN accelerators. CCR advocates a novel approach to reaping the benefits from the reduction of computation and memory accesses as well as the acceleration of the existing dense architectures.

The architectural implications of CCR guide to cope with sparse CNNs using the existing dense accelerators at low hardware overhead. As a case study, we implemented a sparse CNN accelerator, SparseK, following the rationale of CCR. The experiments show that SparseK can achieve a speedup

978-3-9819263-0-9/DATE18/©2018 EDAA

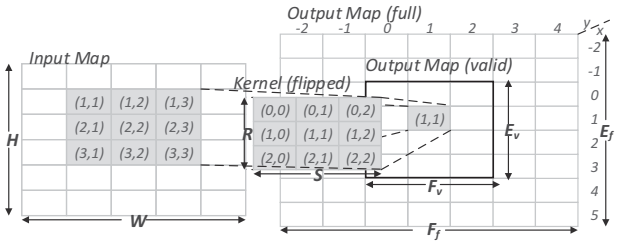

Fig. 2. 2D convolution in valid and full mode.

of 2.9× on VGG16 compared to a comparably provisioned dense architecture. Compared with state-of-the-art sparse accelerators, SparseK can improve the performance and energy efficiency by 1.8× and 1.5×, respectively.

II. Background

A. 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 as shown in Fig. 2, where 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 other modes can be also obtained through 'full' mode by cropping the *omaps*. Otherwise specified, the convolutions in the following texts are in full mode.

The computation of a 2D convolution is defined as

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

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 $(1-R, 1-S)$.

B. 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 number 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 can overlap with others, the overlapped values should be accumulated when concatenated. The triple format provides a representation for sparse matrices since they can be represented as nonzero (dense) matrices and zero ones.

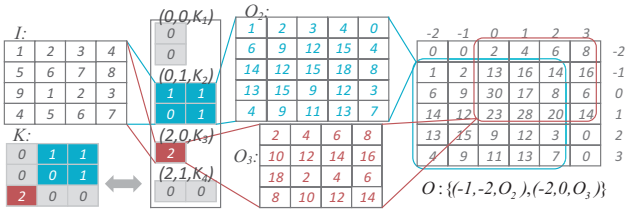

Fig. 3. An example of CCR on sparse kernels.

III. A Concise Convolution Rule

In this section, we present the concise convolution rule, which smartly partitions sparse convolutions into effective and ineffective sub-convolutions.

A. CCR on Sparse Kernels

CCR based on sparse kernels is defined as follows:

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

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

R_l, S_l denotes the height and width of $\mathcal{K}_l$. The following are the proof of CCR-1.

Proof of CCR-1.

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

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

Since the starting coordinate of $\mathcal{O}_l$ is $(1-R_l, 1-S_l)$, which corresponds to coordinate $(-r_l, -c_l)$ in $\mathcal{O}$. When concatenating the partial omaps, there exists an offset denoted as α, β, which can be calculated by (4). Therefore,

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

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 can be reduced. By separating a sparse kernel into nonzero and zero sub-kernels using the triplet representation, the convolution can be transformed into effective sub-convolutions (with nonzero sub-kernels) and ineffective ones (with zero sub-kernels). Since the ineffective convolutions do not contribute to the final results, their

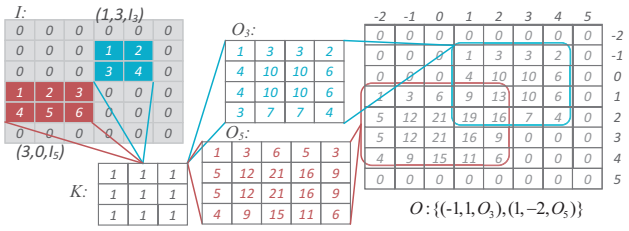

Fig. 4. An example of CCR on sparse input maps.

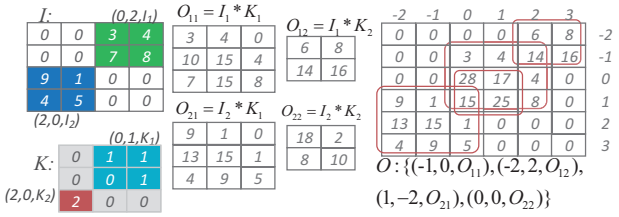

Fig. 5. An example of CCR on sparse kernels and input maps.

computation and memory accesses can be eliminated. Fig. 3 demonstrates an example. The kernel is separated 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 result. We only need to compute the effective 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:

$$\begin{aligned}\alpha_2 &= 1-r_2-R_2 = 1-0-2 = -1\\ \beta_2 &= 1-c_2-S_2 = 1-1-2 = -2\\ \alpha_3 &= 1-r_3-R_3 = 1-2-1 = -2\\ \beta_3 &= 1-c_3-S_3 = 1-0-1 = 0\end{aligned} \tag{6}$$

In the above 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 can also be reduced 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 convolutions derived from CCR-1 maintains the same imap size with the original one, which reveals that they can be easily mapped to the existing dense accelerators at low hardware overhead. The architectural implications will be detailed in Section IV.

B. CCR on Sparse Input Maps

Similarly, CCR based on sparse imaps can be defined as:

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

$$\begin{aligned}&if\ \ \mathcal{I}=\biguplus_{l=1}^{q}(m_l,n_l,\mathcal{I}_l),\quad then\ \ \mathcal{I}*\mathcal{K}=\biguplus_{l=1}^{q}(\gamma_l,\theta_l,\mathcal{I}_l*\mathcal{K})\\ &where\ \ \gamma_l=1+m_l-R,\ \ \theta_l=1+n_l-S\end{aligned} \tag{7}$$

The proof of CCR-2 is similar to the one of CCR-1 and is omitted due to the space limitations. According to CCR-2, a convolution can be transformed to multiple convolutions, each with a sub-imap and the original kernel. The computations of convolutions with sparse imaps can be reduced in a similar way as CCR-1. As exemplified in Fig. 4, 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 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 convolutions maintains the same kernel size with the original one, which indicates that only limited hardware modifications to the previous accelerators are required to support the separation of sparse convolutions.

C. CCR on Sparse Kernels and Input Maps

CCR-1 and CCR-2 can simplify the convolutions when kernels or imaps are sparse, from which CCR based on both sparse kernels and sparse imaps can be derived as follows:

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

$$\begin{aligned}&if\ \ K=\biguplus_{i=1}^{p}(r_i,c_i,K_i),\ \ \mathcal{I}=\biguplus_{j=1}^{q}(m_j,n_j,\mathcal{I}_j)\\ &then\ \ \mathcal{I}*\mathcal{K}=\biguplus_{j=1}^{q}(\gamma_j,\theta_j,\mathcal{I}_j*\mathcal{K})\\ &=\biguplus_{j=1}^{q}(\gamma_j,\theta_j,\biguplus_{i=1}^{p}(\alpha_i,\beta_i,\mathcal{I}_j*\mathcal{K}_i))\\ &=\biguplus_{j=1}^{p}\biguplus_{i=1}^{q}(\alpha_i+\gamma_j+R-1,\beta_i+\theta_j+R-1,\mathcal{I}_j*\mathcal{K}_i)\\ &=\biguplus_{j=1}^{p}\biguplus_{i=1}^{q}(m_j-r_i-R_i+1,n_j-c_i-S_i+1,\mathcal{I}_j*\mathcal{K}_i)\end{aligned} \tag{8}$$

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. 5 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 $\mathcal{O}_{11},\mathcal{O}_{12},\mathcal{O}_{21},\mathcal{O}_{22}$. Then the sub-omaps are 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.

D. Theoretical Reduction on Computation and Data Volume

Through CCR, we can calculate the theoretical reduction on computation and data volume. Fig. 6(a) depicts the computation reduction rate of CCR for the layers of VGG16. CCR-3, no surprising, 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. In Fig. 6(b), we report the data volume reduction including input maps and kernels. CCR-1, CCR-2, and CCR-3 achieve 39.8%, 11.5%, 51.2% on data volume reduction, respectively.

IV. 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.

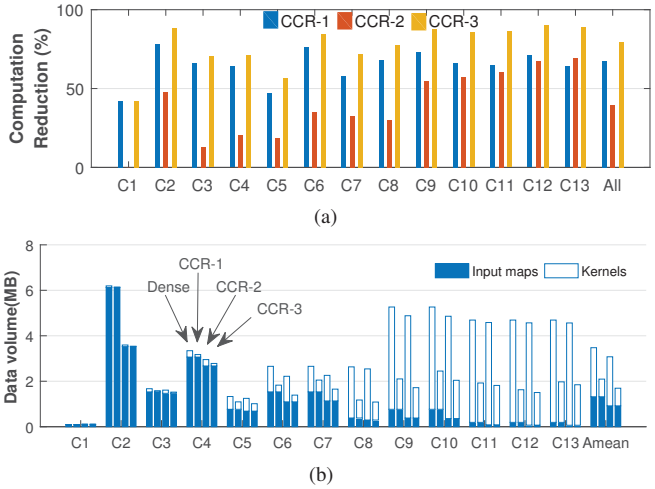

Fig. 6. The computation and data volume reduction rate on VGG16.

A. Generic Dense Architecture

A generic architecture of state-of-the-art accelerators [4] for regular and dense CNNs can be modeled as Fig. 7(a). It comprises a kernel/imap/omap buffer (KB/IB/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. Then they are 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 $P_x \times P_y$ 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. Due to 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 [4].

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 the dummy computations which are essentially unnecessary. Thus, the dense architectures cannot benefit from the reduction of both computation and memory access.

B. Sparse Architectures Supporting CCR

We firstly demo a sparse architecture that supports CCR-1, called SparseK. 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 can be stored in a compressed format, i.e. only the nonzero values and their coordinates appear in KB, as shown in Fig. 7(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

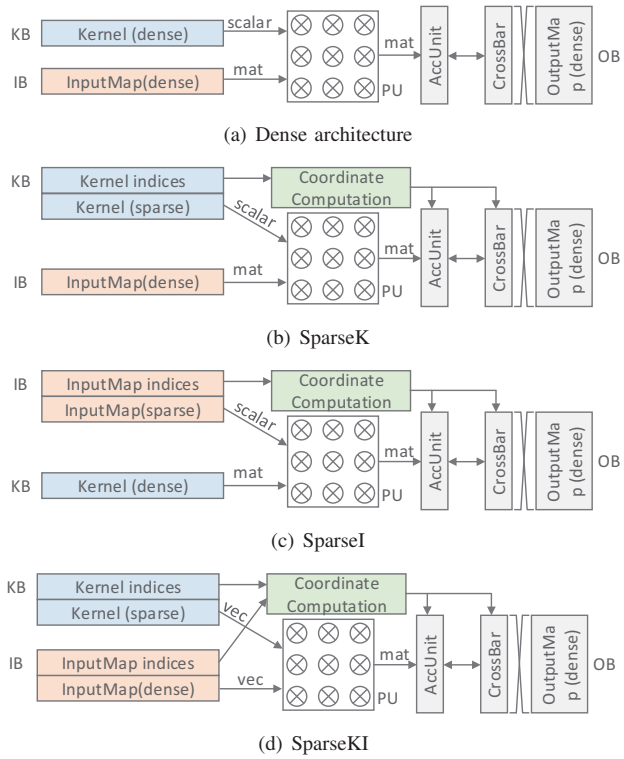

Fig. 7. Sparse architectures supporting CCR.

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 of the sub-convolutions onto the architecture exhibits little difference with the mapping to the dense architecture [4].

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, SparseK can efficiently support sparse convolutions, obviating intrusive PE modifications to the original dense architecture. Actually, there are many optimizing opportunities for SparseK, such as maximizing the data reuse in the PU. The optimizing techniques are similar to those for dense architectures and the details are beyond the scope of this paper.

The architecture supporting CCR-2 (SparseI) can be derived following the similar rationale, as shown in Fig. 7(c). To support CCR-3, the SparseKI architecture is more complicated with higher overhead than the former two. Since the sub-convolutions derived from CCR-3 differ in both input map size and kernel size, the complexity significantly increases when mapping them to the same hardware. A feasible solution is that the input maps and kernels are all partitioned into sub-imaps and sub-kernels of 1×1 shapes. The convolution with a 1×1 input map and 1×1 kernel thus equals to the scalar multiplication. As every scalar in $\mathcal{I}$(a 1×1 input map)

TABLE I
THE SPARSE ARCHITECTURE DESIGNS.

Arch	Dense	SparseK	SparseI	SparseKI
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	-	$O(1)$	$O(1)$	$O(P_x \times P_y)$

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 I summarizes the characteristics of these architectures. Clearly, SparseKI can eliminate the most ineffective computations. However, the hardware design is the most complicated. Moreover, SparseKI 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, SparseK is remarkably efficient and economic since it can achieve a comparably high computation reduction rate (as reported in the theoretical estimation in Fig. 6(a)) at a lower overhead than SparseKI. Therefore, we opted for SparseK as the architecture to efficiently cope with not only dense but also sparse CNN models.

V. EVALUATION

A. Experimental Methodology

Implementation. We implement SparseK in Synopsys design flow on TSMC 65nm 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). SparseK is equipped with a PU containing PE array size of 8×8, and the buffers (IB, KB, OB) are all at 128KB. The detailed designs such as the architecture of the PE arrays and the buffer organization is implemented to match [4] as closely as possible.

Baselines. Shidiannao [4] is selected as the baseline dense architecture, Cambricon-X [9], Cnvlutin [10] and SCNN [11] are the baseline sparse architectures. All of them are equipped with the same number of multipliers for a fair comparison. We developed a cycle-level simulator to evaluate their performance by recording the cycles of simulation. An energy model is used to evaluate the energy consumption. Because of significant differences in dataflow, buffer sizing/organization, and implementation choices, our evaluated architectures cannot precisely represent the prior proposals.

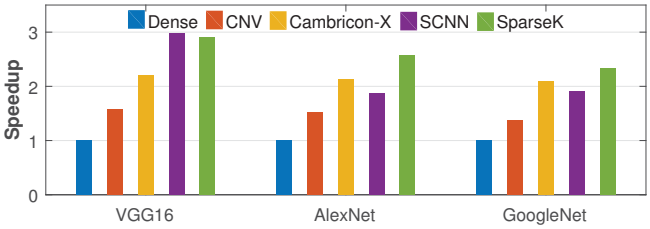

Fig. 8. Speedup over the baseline dense architecture across the models.

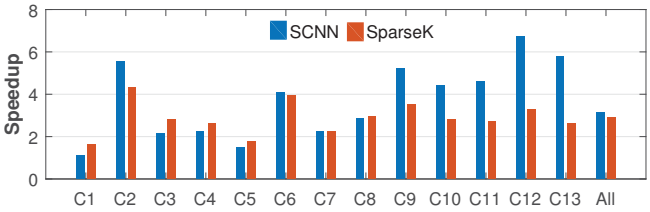

Fig. 9. Speedup over the dense architecture across the layers of VGG16.

Benchmarks. We benchmark the performance using three representative CNNs: VGG16 [12], AlexNet [13], GoogleNet [14]. They provide a wide range of shapes that are suitable for testing the adaptability.

B. Experimental Results

1) Performance: Fig. 8 summarizes the speedups offered by the sparse architectures over the baseline dense architecture. SparseK 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 SparseK varies widely across the models. Specifically, SparseK improves the performance by 2.3–2.9$\times$ over the dense architecture, 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, only on VGG16 it delivers a slight performance advantage over SparseK but performed much worse than expected on AlexNet and GoogleNet. The main reason is that SCNN suffers severe performance degradation from the unbalanced distribution of computations among the PEs. The results reveal that although SparseK is unable to remove all the ineffective computations, it is remarkably efficient across various models.

The performance results can be better understood by looking at the performance breakdown across the layers in VGG16 as shown in Fig. 9. For the bottom layers like C1 and C3, SparseK achieves a superior performance to SCNN. One reason is that the sparsity of kernels dominates these layers which well matches the advantages of SparseK, while imaps are not so sparse (only 5% sparsity in C1). As the sparsity increases with the layers going deeper, SCNN achieves an overwhelming performance than SparseK since imaps and kernels are both very sparse in the later layers (the rightmost layers). However, as the computation volume is mainly dominated by the bottom layers (larger input/output maps), SparseK can achieve a com-

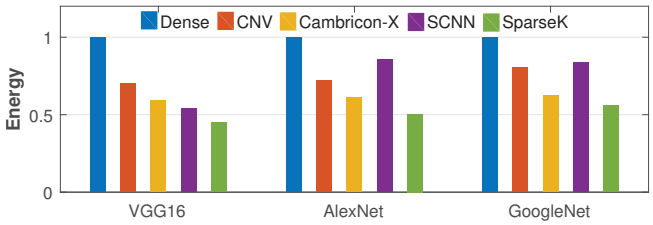

Fig. 10. Energy consumption normalized to the dense architecture.

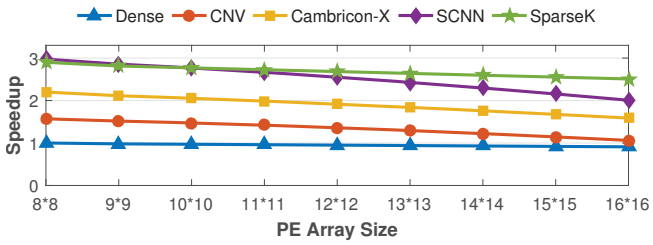

Fig. 11. Speedup on VGG16 for different scales of the PE array.

parable overall performance with SCNN due to the superior performance in the bottom layers.

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

2) Energy Consumption: In Fig. 10, we report the energy comparison of the architectures which has been normalized to the energy of the dense architecture. On average, SparseK 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 SparseK achieves an improvement of $2.23\times$ over the dense architecture 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 SparseK stems from: 1) the improvement of performance, SparseK achieves a high computation reduction rate and free from performance degradation caused by irregular computation distributions; 2) low hardware overhead, the architecture of SparseK is much simpler than SCNN, thus can implemented with lower power.

3) Scalability: We evaluate the scalability of the architectures from the sensitivity to the scale of PE array to study the hardware scale-out merit. Fig. 11 compares the performance of the architectures on VGG16. With the scaling up of PE array, SparseK maintains a stable high performance since the input map size is usually larger than PE array size, maintaining a high resource utilization. Cnvlutin and Cambricon-X suffer from a slow performance degradation with the increasing of the PEs, while SCNN incurs a much severe performance degradation. When PE array is larger than 9×9, SparseK surpasses SCNN for the performance on VGG16. The results highlights that SparseK has good scalability.

VI. Conclusion

This paper proposed a concise convolution rule (CCR), to diminish the gap between sparse CNNs and dense CNN accelerators. Unlike prior approaches which trade complexity for flexibility, we take an alternative by smartly partitioning sparse convolutions into effective and ineffective sub-convolutions. By doing so, the ineffective sub-convolutions can be removed while the effective ones can be easily mapped to existing dense CNN accelerators without intrusive PE modifications. CCR advocates a novel approach which reaps the benefits from the reduction of computation and memory accesses as well as the acceleration of the existing dense architectures. Inspired by CCR, the proposed accelerator (SparseK) provides a tremendous performance advantage over prior approaches and exhibits good scalability. SparseK achieves a speedup of $2.9\times$ on VGG16 compared to a comparably provisioned dense architecture. Compared with state-of-the-art sparse accelerators, SparseK can improve the performance and energy efficiency by $1.8\times$ and $1.5\times$, respectively.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61532017, 61572470, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000. The corresponding authors are Guihai Yan and Xiaowei Li.

References

[1] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout," in *International Conference on Acoustics, Speech and Signal Processing*, 2013.

[2] 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," in *Proceedings of the 19th ASPLOS*.

[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 FPGA*, 2015.

[4] 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," 2015.

[5] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, 2017.

[6] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE 23th HPCA*, 2017.

[7] 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.

[8] 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*, 2014.

[9] 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 *Proc. of MICRO*, 2016.

[10] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the 43rd ISCA*, 2016.

[11] A. Parashar et al., "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of 44th ISCA*, ACM, 2017.

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

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

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. of CVPR*, 2015.