Sense: Model Hardware Co-design for Accelerating Sparse Neural Networks

WENHAO SUN, University of Science and Technology of China, China
DENG LIU, University of Science and Technology of China, China
ZHIWEI ZOU, University of Science and Technology of China, China
WENDI SUN, University of Science and Technology of China, China
JUNPENG WANG, University of Science and Technology of China, China
YI KANG, University of Science and Technology of China, China
SONG CHEN, University of Science and Technology of China, China

Sparsity is an intrinsic property of neural network (NN). Many software researchers have attempted to improve sparsity through pruning, for reduction on weight storage and computation workload, while hardware architects are working on how to skip redundant computations for higher energy efficiency, but there always exists overhead, causing many architectures suffering from only minor profit. Therefrom, systolic array becomes a promising candidate for the advantages of low fanout and high throughput. However, sparsity is irregular, making it tricky to fit in with the rigid systolic tempo. Thus, this paper proposed a systolic-based architecture, called Sense, for both sparse input feature map (IFM) and weight processing, achieving large performance improvement with relatively small resource and power consumption. Meanwhile, we applied channel rearrangement to gather IFMs with approximate sparsity and co-designed an adaptive weight training method to keep the sparsity ratio (zero element percentage) of each kernel at 1/2, with little accuracy loss. This treatment can effectively reduce the irregularity of sparsity and help better fit with systolic dataflow. Additionally, a dataflow called Partition Reuse is mapped to our architecture, enhancing data reuse, lowering 1.9x–2.6x DRAM access reduction compared with Eyeriss and further reducing system energy consumption. The whole design is implemented on ZynqZCU102 and performs at a peak throughput of 409.6 GOP/s, with power consumption of 11.2W; compared with previous sparse NN accelerators based on FPGA, Sense takes up 1/5 less LUTs and 3/4 less BRAMs, reaches 2.1× peak energy efficiency and achieves 1.15×–1.49× speedup.

CCS Concepts:
• Computer systems organization → Systolic arrays;
• Hardware → Hardware accelerators.

Additional Key Words and Phrases: sparsity, neural networks, systolic array, FPGA

ACM Reference Format:
Wenhao Sun, Deng Liu, Zhiwei Zou, Wendi Sun, Junpeng Wang, Yi Kang, and Song Chen. 2018. Sense: Model Hardware Co-design for Accelerating Sparse Neural Networks. J. ACM 37, 4, Article 111 (August 2018), 22 pages. https://doi.org/10.1145/1122445.1122456
1 INTRODUCTION

Nowadays, deep learning has been widely applied in numerous areas, such as image recognition[15], [25],[26],[12], speech recognition[24], object detection[23], computer vision[13], etc. Neural network (NN) mainly consists of convolutional (CONV) layers and fully connected (FC) layers; the former conduct convolution and the latter conduct matrix multiplication, both of which are basically multiply-add operations. With higher precision and more complication comes heavier workload for network computing. Therefore, researchers began to bend their efforts for exploration of neural network hardware accelerators.

Although NNs bring intensive computations, there exists monotonous operations and a huge amount of data reuse; thus, many hardware architects attempt to reduce runtime by improving computation parallelism and arrange dataflow to cut down on data movement power consumption. For example, Eyeriss[4] applies Row-stationary dataflow in an array of 168 PEs, to minimize movement consumption; TPU[14] revives Systolic Array to realize high energy efficiency and throughput with structured dataflow and low bandwidth; Thinker[28] supports bit-width adaptive computing and on-demand array partitioning and reconfiguration, to maximize resource utilization; Tianjic[6] builds an unified model description framework and an unified processing architecture to accommodate a wide range of network computation.

Those works mainly actualize acceleration through parallelism improvement and dataflow optimization, but ignore that sparsity is an intrinsic property of NN, which can be utilized to leave out redundant storage and computation. Therefore, based on previous works, researchers developed various sparsity-aware NN accelerators and pruning, quantization methods[22],[5],[7],[17],[32],[9]. However, for the great cost of sparsity handling, some architectures cannot fully support sparse processing or harvest little improvement. Such as EIE[11], it supports both sparse weight and IFM processing, but only focuses on FC layers; though SCNN[21] supports arbitrary sparse patterns, it suffers from high sparsity processing cost; and Cambricon-S[30] employs customized coarse-grained pruning to reduce the irregularity of weights while suffering from accuracy drop. Cambricon-X[29] and Cnvlutin[1], respectively, take advantage of sparse weights and IFMs to eliminate operations induced by zero. Zhu et al.[31] processes each weight individually with large logic resource and bandwidth overhead. Soon afterwards, to further reduce sparsity handling cost, researchers try to fit sparsity into systolic architecture, but its structured dataflow contradicts with the irregularity of sparsity. Swallow[18] realizes sparsity processing in systolic array and supports weight and IFM sparsity in FC and CONV layers, but lacks of sparsity irregularity reduction for systolic dataflow. FESA[27] trains the kernels into a small number of formalized patterns to regularize dataflow for systolic tempo, but only implemented on Cifar-10 and Cifar-100[8] currently for its precise training object. Thus, how to obtain balance between sparsity irregularity tolerance and hardware cost is still worth exploring.

This paper aims to fit sparsity processing into systolic array, realizing acceleration with low bandwidth consumption and high throughput. We proposed a systolic-based architecture, called Sense, for both sparse IFM and weight processing, achieving large performance improvement with relatively small resource and power consumption. Meanwhile, we applied channel rearrangement to gather IFMs with approximate sparsity and co-designed an adaptive weight training method to keep the sparsity ratio of each kernel at 1/2, with little accuracy loss. This treatment greatly improve synchronization across PEs. Additionally, we map the Partition Reuse dataflow into Sense, to increase data reuse and reduce DRAM access. Our main contributions are as follows:
1) We designed a systolic-based architecture for both sparse IFM and weight processing with reasonable overhead, which can accelerate process by compressing data and eliminating operations induced by zero in systolic array. Moreover, channel rearrangement is applied for IFMs to further reduce sparsity irregularity. Compared with Zhu et al[31], we consume 1/5 less LUTs and 3/4 less BRAMs at the same peak throughput, and achieve 1.15×−1.49× speedup.

2) We co-designed an adaptive weight training method to keep the sparsity ratio of each kernel at 1/2 with little accuracy loss, which can highly reduce the weight sparsity irregularity, achieving 2.36−2.63x speedup compared with Swallow[18].

3) We proposed a Partition Reuse dataflow to improve data reuse rate and further reduce DRAM access, and a mapping strategy to adapt various NNs. Based on the on-chip buffer size, we partition the IFMs and weights into tiles for independent computation and determine the reuse strategy according to the sizes of IFMs and weights, achieving 1.9×−2.6x DRAM access reduction compared with Eyeriss.

The rest of this article is organized as follows. Section II provides the theory of sparse NNs acceleration and proposes some solutions to the sparsity irregularity reduction. Section II provides the methodology of sparse processing for CONV layer and FC layer, and unstructured pruning method for sparsity regularity. Section IV details the design of accelerator architecture. Section V introduces the Partition Reuse dataflow and mapping strategy used in our architecture to reduce DRAM access. Section VI evaluates our design compared with the baseline based on benchmarks. And finally Section VII concludes this article.

2 THEORY

According to the computing process of NN, skipping zeros of IFMs and weights can effectively accelerate execution. However, for systolic array, there exists problems of dataflow synchronization—that is, PEs with high sparsity rate must wait for those with low sparsity rate to finish. Thus, lots of PE will be idle, reducing array utilization rate. Additionally, when a certain PE contains weights with really low sparsity rate, it greatly block the overall acceleration. To solve those problems, we applied channel rearrangement to gather IFMs with approximate sparsity, and trained the sparsity ratio of each kernel at a fixed value of 1/2, equalizing and increasing sparsity ratios across PEs. Based on this training model, an IFM and weight sparsity-aware systolic array is designed to actualize steady acceleration and high PE utilization at relatively low hardware cost and power consumption. Moreover, we customized an adaptive network mapping strategy for sparse processing on our architecture, decreasing DRAM access and data movement.

2.1 Computing Process of NN

The computing process of NN is shown in Fig.1(a). A Conv layer receives \( C_i \) input channels(ICs) of IFMs with size of \( H_i \times W_i \). Each OFM of \( C_o \) output channels(OCs), with size of \( H_o \times W_o \), is generated by sliding a \( H_k \times W_k \) window on the IFM with a stride size of \( S \). Each of \( C_o \) groups of kernels corresponds to one OFM and each of \( C_i \) kernels within a group corresponds to one IFM. After convolving by sliding windows and accumulating across \( C_i \) ICs, we obtain the final OFMs of \( C_o \) OCs. To unify computing process, for Conv layer with stride \( S \) larger than 1, we split the single IFM, \( I_x[H_i,W_i] \), and kernel, \( K_y[H_k,W_k] \), respectively into \( S \times S \) sub-blocks, \( I_{sub}[S,S,H_{is},W_{is}] \) and \( K_{sub}[S,S,H_{ks},W_{ks}] \), shown in Equ.1 and Equ.2. The partial sum(Psum) produced by each sub-blocks are accumulated to get the convolution results of current IFM, shown in 1(b). Set IFMs as matrix \( IM[C_i,S,S,H_{is},W_{is}] \), weights as \( WM[C_o,C_i,S,S,H_{ks},W_{ks}] \) and outputs as \( OM[C_o,H_o,W_o] \), the process in Fig.1 can be described as Equ.3, where \( 0 \leq o < C_o,0 \leq s_r,s_c < S,0 \leq x < H_{is},0 \leq y < W_{is},0 \leq a < H_{ks},0 \leq b < W_{ks} \).

\[
I_{sub}[s_r,s_c,x,y] = I_x[S \times x + s_r,S \times y + s_c]
\]

\( J. ACM, Vol. 37, No. 4, Article 111. Publication date: August 2018. \)
\[ W_{sub}[s_r, s_c, a, b] = W_s[S \times a + s_r, S \times b + s_c] \]  
(2)

\[ OM[o, H_o, W_o] = \sum_{i=0}^{C_i} \sum_{s_r=0}^{S} \sum_{s_c=0}^{S} \sum_{x=0}^{W_{is}} \sum_{y=0}^{H_{ks}} IM[i, s_r, s_c, x, y] \times WM[i, o, s_r, s_c, a, b] \]  
(3)

Assuming there is only one MAC (Multiplication and Accumulation) unit, computing duration of one layer can be described as Equ. 4.

\[ T = C_o \times C_i \times S \times H_{is} \times W_{is} \times H_{ks} \times W_{ks} \]  
(4)

![Fig. 1. (a) Convolution process. (b) Decomposition of IFM and kernel.](image)

### 2.2 Issues & Solutions to Sparsity Irregularity

To realize acceleration, we need to reduce related factors, except for structure parameters of network, \( S, C_o, C_i \) in Equ. 4. Based on the intrinsic sparsity of NN, when skipping zero computations of IFMs and weights, we manage to cut down the sizes of IFMs and kernels, \( H_{is}, W_{is}, H_{ks}, W_{ks} \) in Equ. 4, reducing processing duration. Generally, mapping all the computations on a PE array can greatly improve parallelism, but it becomes tricky taking sparsity irregularity into account. There are mainly two issues as follows:

1) Assuming the numbers of nonzero elements of each kernel is \( W_{num} \), the computing duration of the whole array is determined by the largest \( W_{num_{max}} \); when it approaching \( H_k \times W_k \), it means less speedup can be brought through sparsity.

2) For \( W_{num_{max}} \) differ, PEs with small \( W_{num} \) must wait until other large ones finish, which causes unbalanced loads, lowering general PE utilization rate.

To break performance bottleneck, we designed to keep sparsity ratio of each kernel at 1/2 through training, to guarantee appreciable speedup for the whole array. To handle unbalanced loads, we
trained the numbers of nonzero elements of each kernel to be as close to each other as possible, balancing loads across array. As shown in Fig.2, kernel $K_0$ and $K_1$ are respectively loaded in $PE_0$ and $PE_1$. Assume the computing time needed for a single weight is $T_k$, and the total numbers of nonzero elements in Fig.2(a) and Fig.2(b) are both 8. In Fig.2(a), the maximum number of nonzero elements in $K_0$ and $K_1$ is 6, so we have the total computing time of $6T_k$. For there is a great gap between the numbers of nonzero elements in $K_0$ and $K_1$, $PE_1$ will be idle for $4T_k$ to synchronize with $PE_0$. However, in Fig.2(b), the numbers of nonzero elements of two kernels are both 4, eliminating idle PEs. Therefore we obtain the total computing time of $4T_k$ and greatly improve PE utilization rate.

Moreover, channel rearrangement is applied to gather IFMs with approximate sparsity, further increasing sparsity ratio and decreasing irregularity of IFMs. As shown in Fig.3, two of IFMs, $I_0$, $I_1$, $I_2$ and $I_3$, of different ICs will be loaded to PE each time, assuming the computation time of a single IFM element is $T_i$ and the numbers of nonzero elements in each IFM are respectively 8, 12, 8, 12. If we load the IFMs in order of $I_0$ and $I_1$, $I_2$ and $I_3$, as shown in Fig.3(a), the total computing time is $24T_i$ and $PE_0$ will be idle for $4T_i$ each time; while if we load them in an ascending order of the numbers of nonzero elements, as shown in Fig.3(b), the total computing time reduces to $20T_i$ and the PE utilization rate reaches 100%. In both ways, we managed to decrease sparsity irregularity of weights and IFMs, and greatly improve array acceleration and PE utilization rates.

Fig. 3. (a) IFMs without channel rearrangement. (b) IFMs with channel rearrangement.
3 METHODOLOGY

Since CONV layers conduct convolution and FC layers conduct matrix multiplication, we applied different sparsity processing method separately. For CONV layer, to average numbers of nonzero elements and reduce compression cost, we directly compress IFM and the whole kernel; therefore weight-oriented dataflow[31] is introduced. For FC layers, whose computation pattern is matrix multiplication, with $H_{ik}=1$, $W_{ks}=1$, $W_{is}=1$, $H_{ks}=1$, to better fit with sparsity computing, we use outer product[20], so weight matrix and IFMs are compressed by column.

3.1 Sparsity Processing of CONV Layers

For CONV layer computing, we applied weight-oriented dataflow, shown in Fig.4. It takes one weight each time to compute with the whole IFM, repeatedly, whose results will be accumulated until the final results are obtained. The compression format of IFM and weight is shown as Fig.5, which contains the length of nonzero elements, bitmap[10] and nonzero values. The length, $I_{num}$ and $W_{num}$, record the number of nonzero elements, bitmap records data zero flags(0 for zero, 1 for nonzero), and nonzero values are represented as $I_{nz}$ and $W_{nz}$. For kernels with stride $S > 1$, we compress the decomposed blocks as in Fig.1(b), which can simplify sparsity processing with consequent data. After compression, the input and weight matrices are turned into $IM_{nz}[C_i,S,S,I_{num}]$ and $WM_{nz}[C_o,C_i,S,S,W_{num}]$, and the length of each sub-block of IFM and weight are represented as $IM_{num}[C_i,S,S]$ and $WM_{num}[C_o,C_i,S,S]$, so we have the computing duration description as shown in Equ.5.

$$T = \sum_{o=0}^{C_o} \sum_{i=0}^{C_i} \sum_{r=0}^{S} \sum_{c=0}^{S} IM_{num}[i,s_r,s_c] \times WM_{num}[o,i,s_r,s_c]$$

Fig. 4. Weight-oriented dataflow.

Fig. 5. Compression pattern of IFM and weight for CONV layer.
For sparse processing of CONV layers, we first traverse IFMs then weights and decompress corresponding bitmap into location info, \((I_{row}, I_{col})\) and \((W_{row}, W_{col})\). Based on that, output location \((P\text{sum}_{row}, P\text{sum}_{col})\) is calculated and set as invalid when out of boundary. Then according to the output location, we obtain buffer address, \(P\text{sum}_{addr}\), of accumulated partial sums for the product of \(I_{nz}\) and \(W_{nz}\), shown in detail as Equ.6-9.

\[
P\text{sum} = +I_{nz} \times W_{nz}
\]

\[
P\text{sum}_{row} = I_{row} - W_{row}(0 \leq P\text{sum}_{row} < H_{o})
\]

\[
P\text{sum}_{col} = I_{col} - W_{col}(0 \leq P\text{sum}_{col} < W_{o})
\]

\[
P\text{sum}_{addr} = P\text{sum}_{row} \times W_{o} + P\text{sum}_{col}
\]

Fig. 6 shows an example of sparse convolution computing. The decompression of IFMs is shown in Fig.6(a). Assuming \(H_{o} = 3, W_{o} = 3\), we have \(I_{nz} = \{10, 20, 30, 40\}\), whose location info are \((I_{row}, I_{col}) = \{(0, 0), (1, 1), (2, 2), (3, 3)\}\) respectively. Fig.6(b) shows the decompression of kernels. Assuming \(W_{nz} = \{10, 20\}\), we have location info , \((W_{row}, W_{col}) = \{(0, 0), (1, 1)\}\). The computing flow is shown in Figure 6(c). Assuming the initial value of Psum is zero, at \(T_{0}\), according to Equ.6-9, we fetch \(I_{nz} = 10, (I_{row}, I_{col}) = (0, 0)\), and \(W_{nz} = 10, (W_{row}, W_{col}) = (0, 0)\) and calculate out \(P\text{sum} = 100, \text{and} P\text{sum}_{addr} = 0\), so value 100 is accumulated in address 0 of Psum buffer. Similarly at \(T_{1}, T_{2}\), value 400 and 300 are accumulated in address 4 and 9 respectively. Since \(P\text{sum}_{addr}\) is out of boundary at \(T_{3}\) and \(T_{4}\), operations are regarded as invalid. Finally at \(T_{5}, T_{6}\) and \(T_{7}\), 400, 600, 400 are accumulated with previous values in address 0, 4, 9 respectively. The whole process takes 8 cycles according to Equ.5, 8x faster than 64 cycles of the original process.

![Fig. 6. Sparse computing process of CONV layer.](image)

### 3.2 Sparse Processing of FC Layers

For FC layer, we applied outer product as shown in Fig.7. It takes one element of IFM each time to compute with the corresponding column of weight matrices, repeatedly, whose results will be accumulated until all inputs are traversed. IFMs and weights are all compressed by column, similar with CONV layer, as shown in Fig.8. Then the total computing duration is shown in Equ.10.

\[
T = \sum_{a=0}^{C_{o}} \sum_{i=0}^{C_{i}} WM_{num}[o, i](IM_{nz}[i]! = 0)
\]
The computing process of one single element of IFM and one column of weights is similar to CONV layers. First, decompress the location info of IFMs and weights. Then fetch nonzero IFM element, $I_{nz}$ with location info of $I_{col}$, and nonzero weights of the $I_{col}$ column of weight matrix with location info of ($W_{row}$,$W_{col}$). Eventually, we further calculate the Psum$_{addr}$ of Psum Buffer for Psum accumulation. Details are shown in Equ.11.

$$Psum_{addr} = Psum_{col} = W_{row}$$ (11)

Fig. 7. Process of Outer Product.

Fig. 8. Compression pattern of IFM and weight for FC layer.

Fig.9 shows an example of FC layer computing, decompression of IFMs and weights in Fig.9(a) and 9(b), computing process in Fig.9(c). For the second and forth elements of input column are zero, only the first and third column of the weight matrices will be fetched. Assuming the initial value of partial sum is zero, at $T_0$, according to Equ.6, we have $I_{nz} = 10$, $I_{col} = 0$, $W_{nz} = 10$ and ($W_{row}$,$W_{col}$) = (0, 0), then $Psum = 100$ and $Psum_{addr} = 0$, so value 100 is accumulated in address 0 of Psum buffer. Sequentially at $T_1$, we have $I_{nz} = 10$, $I_{col} = 0$, $W_{nz} = 20$ and ($W_{row}$,$W_{col}$) = (2, 0), then $Psum = 200$ and $Psum_{addr} = 2$, so value 100 is accumulated in address 2 of Psum buffer. Thus similarly at $T_2$, $Psum = 400$ and $Psum_{addr} = 0$, and at $T_3$, $Psum = 700$ and $Psum_{addr} = 2$. The whole process takes 4 cycles according to Equ.10, 4× faster than 16 cycles of the original process.

Fig. 9. Sparse computing process of FC layer.
3.3 Sparse Training

NNs contain many redundant weights, which can be eliminated to enforce sparsity. Thus, to fit with systolic dataflow, we set the value of a half of kernel elements as zero based on unstructured pruning, referring to PyTorch[19] framework. Specifically, weights of relatively small values are regarded as redundancy and set zero, while those of large values are remained. In CONV layers, taking kernels of size 3x3 for example, we traverse all 9 elements and set the top 5 smallest absolute values as zero; while in FC layers, each column of weight matrix is sorted by absolute values according to ascending order and the top 1/2 are set zero. To maintain accuracy, we re-trained the pruned weights through backward propagation.

4 ARCHITECTURE

To achieve high speedup and energy efficiency at low hardware cost and power consumption, we chose systolic array as mainstay. Then, how to efficiently fit in sparsity processing becomes the major problem. We equipped compressing units to store data in compressed format, and added channel rearrangement units to gather IFM computations with approximate sparsity ratio, improving overall synchronization. Accordingly, decompressing units are designed for IFM and weight inputs to decompress location info. Moreover, based on our dataflow, we surrounded the array with input, weight and output buffers to maximize data reuse and minimize DRAM access.

4.1 Overview

As shown in Fig.10, the top level architecture mainly consists of input-weight buffers, a PE array, post-processing units, output buffers, compressing units, a top controller, channel rearrangement units and DDR-AXI interface.
Compressed data are first fetched from DRAM by input and weight buffers, then decompress them to get the none-zero data and location info for PE array. After parallel MAC operations by PEs, post-processing units take in the output to perform ReLU, pooling, etc., whose results are buffered in the output buffers. Before storing back to DRAM, the OFMs are compressed by the compressing units. Meanwhile, channel rearrangement units rearrange output channels according to the sparsity ratio of OFMs. Top controller masters the whole computing flow of network and DDR-AXI interface transports data on/off-chip.

4.2 On-chip Buffer
To reduce DRAM access, we need to buffer IFMs, weights and OFMs. Thus, on-chip buffers mainly consist of input-weight buffers and output buffers; input-weight buffer consists of two $4K \times 16b$ BRAMs for IFMs and weights, two $256K \times 8b$ LUTRAMs for location info and a decompressing unit for bitmaps, and output buffer consists of two $1K \times 16b$ BRAMs for OFMs. To avoid network computation blocking from data loading, we apply two BRAMs as Ping-Pong buffers for each unit, one for DRAM access and the other for PE transporting. Inputs of each row of PE array are supplied by one input-weight buffer; outputs of each column are collected by one output buffer. Additionally, to make full use of on-chip buffer, storage can be re-allocated on demand for IFMs and weights.

4.3 PE Array
The PE array, in charge of MACs, is composed of $N_{PE} \times N_{PE}$ PEs, among which PEs of the same row share IFMs of one IC and PEs of the same column process OFMs of one OC. Each PE contains a MAC unit, an address computing unit and a Psum buffer, as shown in Fig.11. During execution, we multiply IFMs with weights and calculate location info based on Equ.6-9,11. Results are accumulated with temporarily stored Psums of corresponding address in Psum Buffer. When all ICs of this output block are finished, we pause the computation, accumulate across PEs, and finally send the results to post-processing units. To fully utilize buffer resources, Psum buffers will be cleared for the next run; meanwhile to reduce power consumption, the MAC unit will be gated when the IFM/weight is zero or the location info is invalid. Considering a trade-off between resource and performance, we set the Psum buffer as a $256 \times 16b$ LUTRAM. Moreover, in the case of low sparsity, we switch the PE to dense mode and accumulate the products of IFM and weight with adjacent PE, which can shield the address calculation unit and Psum buffer, thereby reducing the power consumption of PEs.

![Fig. 11. PE unit.](image)

4.4 Post-processing Unit
Psums need activating like ReLU to obtain the final OFMs, or sometimes pooling. Thus, post-processing unit mainly consists of a ReLU unit, a pooling unit and a Ping-Pong data buffer of size...
256 × 16b, to keep the pipeline flowing, one for receiving outputs from the array and the other for Post-processing.

4.5 Compressing Unit
To eliminate redundant operations and reduce storage cost, we compress OFMs in compressing unit, which consists of compressing logic circuits and a compressing buffer. If the total number of nonzero elements in output buffer and bitmap is smaller than that of the original OFM, we regard this tile as sparse OFM for it takes up less storage. Then the nonzero elements will be read out from output buffer; meanwhile the compressing circuits will record the length of nonzero elements and bitmap. All above are sent to the compressing buffer subsequently. When the OFM is finished compressing, we set the number of nonzero elements of the entire OFM as the DRAM data transmission length, reducing DRAM accesses. If the OFM is classified as dense, they will be read out directly from output buffer and written back to DRAM. Additionally, there exists overlap between edges of each tile, so we double the edge data in DRAM of sequential address, for the convenience of next layer access.

4.6 Channel Rearrangement Unit
To reduce irregularity of IFMs, we rank the ICs by the numbers of nonzero elements of IFMs, so that channels with approximate sparsity ratio will be gathered in channel rearrangement unit, which consists of a channel index buffer, an nonzero number buffer and a ranking unit. Since only $N_{PE}$ OFMs are produced each time, we divide and rule, ranking each group of OFMs separately; thus Merge Sort is applied and results are stored in channel index buffer. When processing the next layer, IFMs are read in based on ranking indexes, which gathers IFMs with approximate sparsity ratio, further improving synchronization across PEs.

5 DATAFLOW
For limited on-chip buffers, Partition Reuse dataflow is proposed to minimize DRAM access and improve data reuse rate. According to the size of on-chip buffer, we partition the IFMs and weights into tiles and decide the reuse strategy according to DRAM access of IFMs and weights. Additionally, a mapping strategy is also designed to map various networks on our architecture, based on provided network parameters.

5.1 Reuse Strategy
To avoid same IFM and weights being read repeatedly from DRAM and reduce Psum storage, we attempt to consume data as soon as possible once they were read out and write back only the final results to DRAM. Therefore, we apply Psum reuse dataflow[3], taking IFMs and weights of different input channels on-chip to compute and accumulate to obtain the OFM of this output channel. Additionally, based on the intrinsic property of systolic architecture, the same PE column shares the same IFM, which actualizes input reuse[3] and improves bandwidth utilization rate. As for a single PE, we station weights, which actualizes kernel reuse[3] when the whole IFM streams through, reducing weight movement. In this way, we maximize data reuse by fully applying IFM, weight and Psum reuse, to minimize the energy consumption of data on/off-chip movement.

5.2 IFM & Weight Partition
For limited on-chip buffer, IFMs and weights cannot all be stored on-chip, so we partition IFMs into independent sub-tiles. Based on convolution, there’s overlap between sub-tiles; to reduce overlapping area, we preferentially partition them into square shape. Considering the resources of Psum storage, the size of input sub-tiles is set no larger than 16x16, and limited by array size, we
only process \( N_{PE} \) input and output channels concurrently. To obtain complete OFMs, all the input sub-tiles will convolve with kernels in turn. Assuming the numbers of IFM tiles on row/column dimension are \( T_{ifm\_row} \) and \( T_{ifm\_col} \), the numbers of input/output channel tiles are \( T_{ic} \) and \( T_{oc} \), and the numbers of weights and IFMs of one layer are \( I_{mem} \) and \( W_{mem} \), if we station IFMs and calculate out sub-tiles of all output channels, the total DRAM access is \( W_{mem} \times T_{ifm\_row} \times T_{ifm\_row} + I_{mem} \); if we station kernels and calculate out all the sub-tiles of one OFMs, the total DRAM access is \( I_{mem} \times T_{oc} + W_{mem} \). Therefore, weight and IFM reuse strategy is determined by minimum DRAM access \( D_{mem} \), shown in Equ.12. However, when weight can be entirely stored on-chip buffer, we can reuse IFMs without loading weight repeatedly. Then the total DRAM access is \( I_{mem} + W_{mem} \). Additionally, for FC layer, the weight matrix is partitioned into \( T_{ic} \) columns and \( T_{oc} \) rows, which are traversed by row first and then column. As \( T_{ic} \) tiles of IFMs can be entirely stored on-chip, they can be reused with no repeated access, obtaining the total DRAM access of \( I_{mem} + W_{mem} \).

\[
D_{mem} = \min\{W_{mem} \times T_{ifm\_row} \times T_{ifm\_row} + I_{mem}, I_{mem} \times T_{oc} + W_{mem}\}
\] (12)

Fig.12 shows an example of Partition Reuse on different cases in CONV layers. Assuming the size of input sub-tiles is 16x16 and \( N_{PE} \) is 16, as shown in Tab.1, we will apply different reuse strategy based on different network parameters. In case 1, when \( T_{ifm\_row} \) and \( T_{ifm\_col} \) are both 4, \( T_{ic} \) and \( T_{oc} \) are both 8, according to Equ12, weights are preferentially reused; each time we load weights of 16 channels on-chip and traverse every input tile, and then move to the next 16 channels, obtaining the final DRAM access of 4.39MB. In case 2, when \( T_{ifm\_row} \) and \( T_{ifm\_col} \) are both 2, \( T_{ic} \) and \( T_{oc} \) are both 16, IFMs are preferentially reused; each time we load a single input tile of all input channels on-chip and traverse weights of all output channels, and then move to the next input tile, obtaining the final DRAM access of 2.5MB. In case 3, similar with case 2, when \( T_{ifm\_row} \) and \( T_{ifm\_col} \) are both 16, \( T_{ic} \) and \( T_{oc} \) are both 4, IFMs are preferentially reused, but weights can entirely be loaded on-chip for its small quantity, which can avoid repeated access of weights and obtain the final DRAM access of 4.04MB. Fig.13 shows an example of partition in FC layers, in which the weight matrix is divided into 256 rows and 128 columns, and the total DRAM access is 2MB.

| Case | \( T_{ic} \) | \( T_{oc} \) | \( T_{ifm\_col} \) | \( T_{ifm\_row} \) | \( K_h/K_w \) | \( I_{mem} \) | \( W_{mem} \) | Reuse Strategy | DRAM Access |
|------|-------------|-------------|-----------------|-----------------|-------------|-------------|-------------|----------------|-------------|
| 1    | 8           | 8           | 4               | 4               | 5           | 512KB       | 400KB       | weight reuse   | 4.39MB      |
| 1    | 16          | 16          | 2               | 2               | 3           | 256KB       | 576KB       | IFM reuse      | 2.5MB       |
| 3    | 4           | 4           | 16              | 16              | 3           | 4MB         | 36KB        | IFM reuse      | 4.04MB      |
5.3 Mapping Algorithm

Considering dataflow and partition method of IFM and weight, to mapping various networks on Sense, we designed a network mapping algorithm, called Partition Reuse, obtaining array configuration parameters through feeding network structure parameters. As shown in Tab.2, the implications of network structure parameters are identical to those in Fig.1, and the matrix of IFMs and weights turn into $IM_{nz}[T_{ci}, N_{PE}, S, S, T_{ifm\_row}, T_{ifm\_col}, I_{num}]$ and $WM_{nz}[T_{ci}, N_{PE}, T_{co}, N_{PE}, S, S, W_{num}]$ respectively.

Fig. 12. Partition of IFM and weight for CONV layers

Fig. 13. Partition of IFM and weight for FC layers
Table 2. Parameters of Mapping

| Parameter     | Explanation                        | Formula               |
|---------------|------------------------------------|-----------------------|
| $T_{oc}$      | Number of OC tile                  | $\lceil C_o/PE_N \rceil$ |
| $T_{ic}$      | Number of IC tile                  | $\lceil C_i/PE_N \rceil$ |
| $N_{ic}$      | Number of IC in PE array parallel computing | $C_i/T_{ic}$ |
| $N_{oc}$      | Number of OC in PE array parallel computing | $C_o/T_{oc}$ |
| $T_{ifm\_row}$ | Number of IFM tile on row dimension | $[H_i/8]$ |
| $T_{ofm\_row}$ | Number of OFM tile on row dimension | $[H_o/8]$ |
| $T_{ifm\_col}$ | Number of IFM tile on column dimension | $[W_i/16]$ |
| $T_{ofm\_col}$ | Number of OFM tile on column dimension | $[W_o/16]$ |
| $T_{ifm\_cmp\_row}$ | Number of IFM tiles for one OFM computing in row dimension | $T_{ifm\_row}/T_{ofm\_row}$ |
| $T_{ifm\_cmp\_col}$ | Number of IFM tiles for one OFM computing in column dimension | $T_{ifm\_col}/T_{ofm\_col}$ |
| $Reuse\_flag$ | Reuse strategy flag, 0: reuse weight 1: reuse IFM | / |
| $T_{oc\_outer}$ | The weight traverse order when reusing weight | $(reuse\_flag == 0)?T_{oc}:1$ |
| $T_{oc\_inner}$ | The weight traverse order when reusing IFM | $(reuse\_flag == 1)?T_{oc}:1$ |

With configuration parameters loaded, the controller can master the whole network computing flow. Dataflow description is shown in Tab.3; $x$, $y$ represent the row and column of PE array. The 1st and 6th row decides the reuse strategy according to $Reuse\_flag$; if $Reuse\_flag = 1$, we finish the OFM computing of all OCs for one output tile before switching to the next output tile; otherwise, we finish computing of all output tiles for one OC before switching to the next OC. The 2nd and 3rd row describe the partition of OFM, with row number of $T_{ofm\_row}$ and column number of $T_{ofm\_col}$. The 4th and 5th row represent, when stride>1 or pooling is expected, the size of OFM will shrink; thus, to obtain a complete output tile needs input tiles with size of $T_{ifm\_cmp\_row}$ and $T_{ifm\_cmp\_col}$. The 7th row drives the accumulation of input channels to get the final Psums. The 8th and 9th row determines the Psums of SxS sub-IFMs are needed to obtain the complete Psums of current IFMs. The 10th row and the 11th represent the numbers of nonzero elements of one tile of IFM and one kernel; $w_{num\_max}$ is loaded as a parameter for the weights from training are fixed and $I_{num\_max}$ is gained during data loading. The 12th row means the PE of No.$x$ row and No.$y$ column is performing MAC operation.
Table 3. Partition reuse Dataflow

| Line | Expression                                                                 |
|------|-----------------------------------------------------------------------------|
| line1 | $\text{for}(a = 0; a < T_{oc\_outer}; a = a + 1)$                           |
| line2 | $\text{for}(b = 0; b < T_{ofm\_row}; b = b + 1)$                           |
| line3 | $\text{for}(c = 0; c < T_{ofm\_col}; c = c + 1)$                           |
| line4 | $\text{for}(d = 0; d < T_{ifm\_cmp\_row}; d = d + 1)$                      |
| line5 | $\text{for}(e = 0; e < T_{ifm\_cmp\_col}; e = e + 1)$                      |
| line6 | $\text{for}(f = 0; f < T_{oc\_inner}; f = f + 1)$                         |
| line7 | $\text{for}(g = 0; g < T_{ic}; g = g + 1)$                                |
| line8 | $\text{for}(h = 0; h < S; h = h + 1)$                                     |
| line9 | $\text{for}(i = 0; i < S; i = i + 1)$                                     |
| line10| $\text{for}(j = 0; j < W_{num\_max}; j = j + 1)$                         |
| line11| $\text{for}(k = 0; k < l_{num\_max}; k = k + 1)$                         |
| line12| $P_{sum}[k] = W_{nz}[\text{Index}_{oc}, y, g, x, h, i, j] \times$         |
|       | $I_{nz}[g, x, \text{Tile}_{row}, \text{Tile}_{col}, k]$                   |

6 EXPERIMENTS

6.1 Implementation

We conduct functional simulation on Vivado 2021.1 and synthesis on ZynqZCU102, at frequency of 200MHz. Resource utilization, timing and power info are captured from synthesis reports. For DRAM power consumption, we simulate on catic[2] platform. Finally, the total system power consumption is calculated according to the ratio of DRAM access time. Baselines are as follows:

1) Eyeriss[4] is a non-sparse accelerator, using a proposed processing dataflow, called Row Stationary, on a spatial architecture with 168 processing elements. Row Stationary dataflow reconfigures the computation mapping of a given shape, which optimizes energy efficiency by maximally reusing data locally to reduce expensive data movement, such as DRAM accesses. Compared with Eyeriss, the dataflow and mapping strategy of our design can further reduce DRAM accesses.

2) Swallow[18] is a systolic accelerator for sparse processing. By comparison, our design achieves higher speedup through hardware-software co-design and channel rearrangement.

3) Zhu et al[31] is a sparsity-aware CNN accelerator based on FPGA, where each weight is processed individually, while it comes with relatively large resource overhead. By comparison, we achieve higher peak energy efficiency with lower resource overhead. Meanwhile, as Zhu et al[31] excludes IFM sparse processing, Sense managed to gain extra speedup.

4) FESA[27] proposes a software-hardware co-design to address the problem of no-load PEs on sparse networks and reduce pattern numbers of irregularly pruned networks, greatly improving PE utilization through weight training. But it cannot support training of complex networks, for its relatively complex training method. By comparison, we obtained approximate PE utilization with our training method, which is also capable of training for more complex networks.

As for benchmarks, we conduct inference on VGG16[25], AlexNet[15], LeNet5[16] and compare our design with each baseline. Tab.4 displays IFM and weight sparsity ratio in CONV and FC layer.
Table 4. IFM and Weight Sparsity Ratios and Accuracy Loss of Each NNs

|          | $W_{conv}$ | $W_{fc}$ | $IFM_{conv}$ | $IFM_{fc}$ | Top1 Accuracy Loss | Top5 Accuracy Loss |
|----------|------------|----------|--------------|------------|--------------------|--------------------|
| AlexNet  | 55.6%      | 55.6%    | 15.5%        | 75.8%      | 2.86%              | 0.10%              |
| VGG16    | 55.6%      | 55.6%    | 60.9%        | 79.0%      | 5.34%              | 0.71%              |
| LeNet5   | 55.6%      | 55.6%    | 12.3%        | 73.5%      | 1.21%              | /                  |

6.2 Energy Efficiency

As shown in Tab.5, which compares Zhu et al[31] with Sense on resources and energy efficiency, we consume 1/5 less LUT and 3/4 less BRAM resources but achieve 2.1× energy efficiency with $N_{PE} = 32$. Zhu et al[31], whose PEs are independent, requires independent input and output buffers. But our architecture is a systolic array, actualizing IFM reuse across row and OFM reuse across column, which consumes less BRAM resources. Additionally, to address channel for output data, the TLUT of Zhu et al[31] maps all inputs to each PE and CMUX maps outputs of 16 PEs to 1024 banks, while our design abandoned the complex MUX structure and applied sequential accessing for input and output data, consuming less LUT resources than design Zhu et al[31]. Moreover, the LUTs in TLUT and CMUX switch constantly for the needs of mapping in each cycle, which causes huge energy consumption, so by comparison, our design consumes less energy. Meanwhile, we reach a peak throughput of $2 \times 1024 \times 0.2GHz = 409.6GOPS$ with 1024 MAC units, which equals to Zhu et al[31]. Thus, Sense has a better performance than Zhu et al[31].

Table 5. Resource and Energy Comparison of Sense Against Zhu et al[31]

|          | LUT          | DSP          | BRAM         | Power | Peak Throughput | Energy Efficiency |
|----------|--------------|--------------|--------------|-------|-----------------|------------------|
| Zhu et al[31] | 552K(92%)    | 1144(48%)    | 912(48%)     | 23.2W | 409.6GOPS       | 17.7GOPS/W        |
| Sense    | 540K(75%)    | 1106(47%)    | 216(12%)     | 11.2W | 409.6GOPS       | 36.6GOPS/W        |

6.3 Performance

Fig.14 compares the overall performance of Sense against Swallow and Zhu et al[31], indicating Sense achieves 2.36×～2.63× speedup on VGG16 and 1.15×～1.49× speedup on AlexNet respectively. Fig.15 and Fig.16 show the performance improvement contribution of IFM sparsity and weight sparsity on VGG16 and AlexNet respectively. The 'Ideal' means the improvement contributed by the original network sparsity, which indicates the utilization rate of sparsity in each architecture. When processing CONV layers, Swallow takes in one row of kernel at a time, with uneven sparsity distribution, causing relatively inapparent acceleration from weight sparsity, which also exists in sparse IFMs. Thus, performance improvement for Swallow of each layer is insignificant. Zhu et al[31] independently handles each weight, omitting PE synchronization, but exclude IFM sparsity processing, which determines the main contribution of speedup is weight sparsity. In Sense, we keep the sparsity ratio of each kernel at 1/2, obtaining steady speedup. And channel rearrangement is applied to gather IFMs with approximate sparsity ratios, which can further improve overall speedup.
Fig. 14. Overall Performance improvement comparison of Sense against Zhu et al[31] and Swallow on AlexNet and VGG16.

Fig. 15. Performance improvement comparison of Sense against Zhu et al[31] and Swallow on VGG16. (a) Contribution of IFM sparsity. (b) Contribution of weight sparsity.
Fig. 16. Performance improvement comparison of Sense against Zhu et al[31] and Swallow on AlexNet. (a) Contribution of IFM sparsity. (b) Contribution of weight sparsity.

6.4 PE Utilization

The comparison between Sense and FEAS on PE no-load rate reduction is shown in Fig. 17, indicating $1.72\times\sim 2.12\times$ higher utilization rates. FESA trained kernels into some certain patterns, greatly reducing PE no-load rate, but with relatively restricted training object, it’s only implemented on Cifar-10 and Cifar-100[8]. However, our training method managed to keep the sparsity ratio of each kernel consistent, equalizing computing duration of a single kernel for the PE array, which further reduces idle duration of PEs and improves PE utilization rate. Meanwhile, we reduce PE no-load rate induced by sparse IFM through channel rearrangement. Thus, we achieve higher PE utilization rate. Moreover, our training method is adaptive to more complex datasets, obtaining higher universality for NNs.

Fig. 17. No-Load Rate Reduction Comparison of Sense Against FESA.

6.5 DRAM Access

Tab.6 and Tab.7 compare Sense with Eyeriss on DRAM access for AlexNet and VGG16, showing $1.9\times$ and $2.6\times$ reduction respectively. According to the overall architecture of Eyeriss, there exists a 108KB global buffer. To be fair, we shrink our array to $16 \times 16$, with a total data storage of 160KB.
Considering the Row Stationary dataflow of Eyeriss conducts inference by row and its buffers are mainly allocated to Psum, IFM reuse across rows will be reduced when processing images with larger width, because the wider images are, the fewer rows can be stored, increasing DRAM access. Meanwhile, weights will be repeatedly read for the constant switching of input and output channels, so DRAM access is increased when Eyeriss handles layers with a relatively large number of weights. Additionally, Eyeriss applies RLC that takes extra 5 bits to compress an nonzero elements. Based on a total sparsity ratio of 60% for the IFMs of AlexNet and VGG, the average cost of compression for each element is an extra of 2 bits. However, we apply Partition Reuse dataflow that provides friendly support for images of any width; meanwhile, we adjust reuse strategy based on the total numbers of IFMs and weights, further minimizing DRAM access. Moreover, the bitmap compression method costs only one extra bit for each element, introducing 50% reduction compared with RLC of Eyeriss. Therefore, our design achieves a better performance on DRAM access and compression overhead, compared with Eyeriss.

### Table 6. DRAM Access Comparison of Sense Against Eyeriss on VGG16

| Layer   | Parameter of Dataflow | DRAM Access |   |   |   |
|---------|-----------------------|-------------|---|---|---|
|         |                       | Eyeriss     | Sense (Dense) | Sense (Sparse) |
| CONV1-1 | 4 1 16 16 1           | 15.4MB      | 0.38MB        | 0.38MB         |
| CONV1-2 | 8 4 16 16 1           | 54.0MB      | 8.1MB         | 6.87MB         |
| CONV2-1 | 8 8 8 8 1             | 33.4MB      | 2.14MB        | 2.07MB         |
| CONV2-2 | 16 8 8 8 1            | 48.5MB      | 40.0MB        | 19.2MB         |
| CONV3-1 | 16 16 4 4 1           | 20.2MB      | 20.0MB        | 9.61MB         |
| CONV3-2 | 16 16 4 4 1           | 32.2MB      | 20.0MB        | 9.32MB         |
| CONV3-3 | 32 16 4 4 1           | 30.8MB      | 38.0MB        | 17.2MB         |
| CONV4-1 | 32 32 2 4 0           | 17.6MB      | 36.5MB        | 19.2MB         |
| CONV4-2 | 32 32 2 4 0           | 28.6MB      | 36.5MB        | 14.9MB         |
| CONV4-3 | 32 32 2 4 0           | 22.8MB      | 36.5MB        | 14.0MB         |
| CONV5-1 | 32 32 1 2 1           | 6.3MB       | 9.3MB         | 4.06MB         |
| CONV5-2 | 32 32 1 2 1           | 5.7MB       | 9.3MB         | 4.06MB         |
| CONV5-3 | 32 32 1 2 1           | 5.6MB       | 9.3MB         | 4.06MB         |
| Total   |                       | 321.1MB     | 265.8MB       | 124.9MB        |

### Table 7. DRAM Access Comparison of Sense Against Eyeriss on AlexNet

| Layer   | Parameter of Dataflow | DRAM Access |   |   |   |
|---------|-----------------------|-------------|---|---|---|
|         |                       | Eyeriss     | Sense (Dense) | Sense (Sparse) |
| CONV1   | 6 1 4 8 1            | 5.0MB       | 0.44MB        | 0.4MB          |
| CONV2   | 16 6 2 4 0           | 4.0MB       | 4.17MB        | 3.25MB         |
| CONV3   | 24 16 1 2 1          | 3.0MB       | 1.81MB        | 0.81MB         |
| CONV4   | 24 24 1 2 1          | 2.1MB       | 5.25MB        | 2.29MB         |
| CONV5   | 16 24 1 2 1          | 1.3MB       | 3.56MB        | 1.54MB         |
| Total   |                       | 15.4MB      | 15.2MB        | 8.29MB         |
6.6 Sparsity Sensitivity

Sparse processing greatly accelerates computation, but comes with extra operations, increasing 1/3 power consumption; thus, we observed the influence of sparsity variation on array performance, to determine a threshold of computing mode, whether dense or sparse. In this way, our architecture can handle data of any ratio of sparsity. As shown in Fig. 18, taking sparsity ratio of 10% as stride, we obtained energy efficiency on sparse mode and dense mode of Sense respectively, and when sparsity ratios are correspondingly under 30%, dense computing mode is more effective. Therefore, we combined dense and sparse processing and computing modes are switched based on sparsity ratios.

![Fig. 18. Improvement of speedup and energy efficiency based on different sparsity ratios.](image)

7 CONCLUSION

This paper proposed a sparsity-friendly systolic architecture, called Sense, for both sparse IFM and weight processing, achieving large performance improvement with relatively small resource and power consumption. Meanwhile, we applied channel rearrangement to gather IFMs with approximate sparsity and co-designed an adaptive weight training method to keep the sparsity ratio of each kernel at 1/2, with little accuracy loss. This treatment can effectively reduce the irregularity of sparsity and help better fit with systolic dataflow. Additionally, Partition Reuse mapping algorithm is applied to map various networks on Sense, enhancing data reuse, lowering DRAM access and further reducing system energy consumption. The whole design is implemented on ZynqZCU102 and performs at a peak throughput of 409.6 GOP/s, with power consumption of 11.2W; compared with previous sparse CNN accelerators base on FPGA, this architecture takes up 1/5 less LUT and 3/4 BRAM, reaches 2× peak energy efficiency and achieves 1.15×~1.49× average network acceleration.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). 1–13. https://doi.org/10.1109/ISCA.2016.11

[2] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. ACM Trans. Archit. Code Optim. 14, 2, Article 14 (Jun 2017), 25 pages. https://doi.org/10.1145/3085572

[3] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). 367–379. https://doi.org/10.1109/ISCA.2016.40

J. ACM, Vol. 37, No. 4, Article 111. Publication date: August 2018.
[4] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeri ss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[5] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 8609–8613. https://doi.org/10.1109/ICASSP.2013.6639436

[6] Lei Deng, Guanrui Wang, Guoqi Li, Shuangchen Li, Ling Liang, Maohua Zhu, Yujie Wu, Zheyu Yang, Zhe Zou, Jing Pei, Zhenzhi Wu, Xing Hu, Yufei Ding, Wei He, Yuan Xie, and Loping Shi. 2020. Tianjic: A Unified and Scalable Chip Bridging Spike-Based and Continuous Neural Computation. *IEEE Journal of Solid-State Circuits* 55, 8 (2020), 2228–2246. https://doi.org/10.1109/JSSC.2020.2970709

[7] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. 2017. Learning to Prune Deep Neural Networks via Layer-Wise Optimal Brain Surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) (*NIPS’17*). Curran Associates Inc., Red Hook, NY, USA, 4860–4874.

[8] Kostyantyn Filonenko, Robert Wisnovsky, Mohamed Chérriet (ecole De), Denis J. Dean, Charles W. Anderson, Yann LeCun, and Corinna Cortes. [n.d.]. techreport Learning Multiple Layers of Features from Tiny Images, by Alex.

[9] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. arXiv:1803.06365 [cs.LG]

[10] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) (*MICRO ’52*). Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[11] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture* (ISCA). 243–254. https://doi.org/10.1109/ISCA.2016.30

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition* (CVPR). 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) (*MM ’14*). Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[14] Norma P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hunt, Dan Hurt, Julian Ibarz, Aaron Jaffe, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucek, Alan Lundin, Gordon MacKeen, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snellham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture* (ISCA). 1–12. https://doi.org/10.1109/3079856.3080246

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (may 2017), 84–90. https://doi.org/10.1145/3065386

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. arXiv:1608.08710 [cs.CV]

[18] Bosheng Liu, Xiaoming Chen, Yinhe Han, and Haobo Xu. 2020. Swallow: A Versatile Accelerator for Sparse Neural Networks. . 4881–4893 pages. https://doi.org/10.1109/TCAD.2020.2978836

[19] Michela Paganini. [n.d.]. PRUNING TUTORIAL. https://pytorch.org/tutorials/intermediate/pruning_tutorial.html.

[20] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siyong Feng, Chaitali Chakrabarti, Hunsok Kim, David Blaauw, Trevor Mudge, and Ronald Dreilinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture* (HPCA). 724–736. https://doi.org/10.1109/HPCA.2018.00067

[21] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Bruceck Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional
neural networks. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). 27–40. https://doi.org/10.1145/3079856.3080254

[22] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2017. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. arXiv:1608.01409 [cs.CV]

[23] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 779–788. https://doi.org/10.1109/CVPR.2016.91

[24] Tara N. Sainath, Oriol Vinyals, Andrew Senior, and Haşim Sak. 2015. Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks. In 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). 4580–4584. https://doi.org/10.1109/ICASSP.2015.7178838

[25] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[26] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[27] Jingyu Wang, Songming Yu, Jinshan Yue, Zhe Yuan, Zhuqing Yuan, Huazhong Yang, Xueqing Li, and Yongpan Liu. 2020. High PE Utilization CNN Accelerator with Channel Fusion Supporting Pattern-Compressed Sparse Neural Networks. In 2020 57th ACM/IEEE Design Automation Conference (DAC). 1–6. https://doi.org/10.1109/DAC19072.2020.9218630

[28] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Shixuan Zheng, Tianyi Lu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. 2018. A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications. IEEE Journal of Solid-State Circuits 53, 4 (2018), 968–982. https://doi.org/10.1109/JSSC.2017.2778281

[29] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[30] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 15–28. https://doi.org/10.1109/MICRO.2018.00011

[31] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. 2020. An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28, 9 (2020), 1953–1965. https://doi.org/10.1109/TVLSI.2020.3002779

[32] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv:1710.01878 [stat.ML]