2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach

Xuda Zhou[1,2,3], Zidong Du[2,3], Qi Guo[2,3], Shaoli Liu[2,3], Chengsi Liu[6], Chao Wang[1], Xuehai Zhou[1], Ling Li[4], Tianshi Chen[2,3], Yunji Chen[2,5]*

[1] University of Science and Technology of China [3] Cambricon Technologies Co. Ltd.
[2] Intelligence Processor Research Center, Institute of Computing Technology, CAS
[2] State Key Laboratory of Computer Architecture, Institute of Computing Technology, CAS
[4] Institute of Software, CAS [5] CAS Center for Excellence in Brain Science [6] Michigan State University
Email: anycall@mail.ustc.edu.cn {duzidong, guoqi}@ict.ac.cn liuche11@msu.edu
{cswang, xhzhou}@ustc.edu.cn liling@iscas.ac.cn {chentianshi, cyj}@ict.ac.cn

***Abstract*—Neural networks have become the dominant algorithms rapidly as they achieve state-of-the-art performance in a broad range of applications such as image recognition, speech recognition and natural language processing. However, neural networks keep moving towards deeper and larger architectures, posing a great challenge to the huge amount of data and computations. Although sparsity has emerged as an effective solution for reducing the intensity of computation and memory accesses directly, irregularity caused by sparsity (including sparse synapses and neurons) prevents accelerators from completely leveraging the benefits; it also introduces costly indexing module in accelerators.**

In this paper, we propose a cooperative software/hardware approach to address the irregularity of sparse neural networks efficiently. Initially, we observe the local convergence, namely larger weights tend to gather into small clusters during training. Based on that key observation, we propose a software-based coarse-grained pruning technique to reduce the irregularity of sparse synapses drastically. The coarse-grained pruning technique, together with local quantization, significantly reduces the size of indexes and improves the network compression ratio. We further design a hardware accelerator, Cambricon-S, to address the remaining irregularity of sparse synapses and neurons efficiently. The novel accelerator features a selector module to filter unnecessary synapses and neurons. Compared with a state-of-the-art sparse neural network accelerator, our accelerator is $1.71\times$ and $1.37\times$ better in terms of performance and energy efficiency, respectively.

I. Introduction

Neural network (NN) algorithms gain increasing attention in academia and industry widely as they achieve state-of-the-art performance in a broad range of applications, including image recognition [1], speech recognition [2]/synthesis [3] and natural language processing [4]. One important reason for such dominance is the huge amount of data and computations in neural networks. Due to the tightening energy constraints and performance requirements, customized accelerators have emerged as a high-performance and cost-efficient alternative to traditional CPUs/GPUs. Recently, several different accelerators have been proposed to address the efficiency issue [5], [6], [7], [8].

Meanwhile, neural networks keep advancing towards larger and deeper architectures as they are involved in more sophisticated processing tasks in a broader scope of scenarios. Large neural networks intensify the computation and memory accesses with increasing amount of data, i.e., synapses and neurons. For example, AlexNet [9] in 2012, has $650k$ neurons; the number further increased to a million [10] then ten millions [11] in 2013. The number of synapses is much larger: 60 million [9], 1 billion [10] and 10 billions [11], respectively. Despite the computation, such huge amount of data bring challenges for off-chip and on-chip memory bandwidth in accelerator architecture design.

Recently, several researchers rise to such challenges by reducing the computation/data amount. Sparsity has emerged as a direct and effective solution among various techniques. Pruning synapses and neurons in networks can lead to more than $10\times$ data reduction [12] with negligible accuracy loss. Weight encoding via quantization and entropy coding has also been investigated to further shrink the weight size, e.g., $35\times$ [13] smaller for AlexNet. Other techniques work at the circuit-level, e.g., using shorter bit-width [14] (even 1-bit [15]) operators or approximate computing.

However, irregularity caused by sparsity (in both synapses and neurons) prevents accelerators from fully leveraging the computation and data reduction. DianNao [5], DaDianNao [6] and ShiDianNao [7] cannot benefit from sparsity at all. Cambricon-X [16] can only benefit synapse sparsity while Cnvlutin [17] can only benefit neuron sparsity. Although EIE [18] can exploit neuron sparsity and synapse sparsity, it only aims at the *fully-connected* layer.

In this paper, we propose a cooperative software/hardware approach to address the irregularity of sparse neural networks efficiently. Initially, we make the key observation of local

*Yunji Chen (cyj@ict.ac.cn) is the corresponding author.

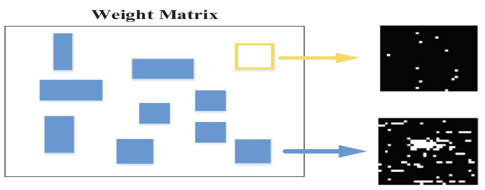


Fig. 1. Local convergence illustrated using a fully-connected layer (weights whose absolute value in the top 10% of total weights are plot in white pixel).

convergence that larger weights tend to gather into small clusters during training, as illustrated in Figure1. Hence we propose a software-based coarse-grained pruning technique to reduce the irregularity of sparse synapses. Based on our proposed measurement, the irregularity reduces 20.13× on average. The coarse-grained pruning technique, together with local quantization, significantly reduces the size of indexes and improves the network compression ratio. For example, we obtain 102.82× reduction in index size and a 79× compression ratio on AlexNet with negligible accuracy loss. After reducing the irregularity of sparse synapses, we further design a hardware accelerator, Cambricon-S, to address the remaining irregularity of sparse synapses and neurons efficiently. The novel accelerator features a neuron selector module (NSM) and several additional synapse selector modules (SSMs) to filter unnecessary neurons and synapses. Compared with state-of-the-art sparse neural network accelerator, our accelerator is 1.71× and 1.37× better in terms of performance and energy efficiency at the cost of $6.73mm^2$ area and $798.55mW$ power at 65nm, respectively.

We make the following key contributions: 1) We make the key observation that local convergence can be leveraged to reduce the irregularity in sparse neural networks, which is the major challenge of efficient acceleration. 2) Based on the observation of local convergence, we propose a coarse-grained pruning technique to reduce the irregularity of sparse synapses drastically. (3) We thoroughly analyze our proposed coarse-grained sparsity with additional dynamic neuron sparsity thus for design principles. (4) Although the proposed pruning technique reduces the irregularity of sparse synapses significantly, we further design a hardware accelerator to address the remaining irregularity with a selector module.

II. Background and Motivation

A. Primer on neural networks

Neural Networks. Modern neural networks consist of multiple types of layers, where inputs are passed and processed through the layers, thus to be classified or recognized. In each layer, each neuron receives multiple inputs for processing and sends the output to next layer via connections. The connections among neurons, i.e., so called synapses, usually have dependent/independent weights. State-of-the-art networks for image applications are convolutional neural networks (CNNs) [21] and deep neural networks (DNNs) [22], [23], where *convolutional*, *pooling*, *normalization* and *fully-connected* layers are usually included. Recurrent neural networks (RNNs) [24] are widely used for speech applications. Long Short-Term Memory (LSTM) [25] is one popular type of RNNs.

TABLE I
Comparison of existing accelerators.

	SSS	SNS	DNS	Note
Cambircon-X [16]	✔	✗	✗	
Cnvlutin [17]	✗	✔	✔	
EIE [18]	✔	✔	✔	Only FC layer
ESE [19]	✔	✗	✗	Only LSTM
SCNN [20]	✔	✔	✔	Extra costs for coordinates
Ours	✔	✔	✔	Coarse-grained Sparsity

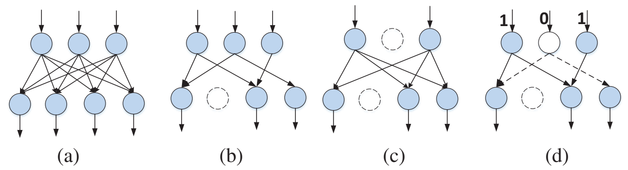


Fig. 2. (a) Dense neural network. (b) Static sparsity: synapse. (c) Static sparsity: neuron. (d) Dynamic sparsity.

Sparsity in Neural Networks. Although modern neural networks are dominant in many applications, the over-parameterized architectures lead to a heavy burden for computation, memory capacity and memory accesses in processing. Various efforts have been made to rise to such challenges, including algorithm-level (e.g., dropout [26], sparsity [12]), architecture-level (e.g. short bit-width operators [14] even 1-bit [15], inexact computing [27]) and physical-level (e.g., dynamic voltage scaling [28]) techniques. Among them, the revisited sparsity technique turns out to be the most effective approach. Researchers have proven the effectiveness of sparsity in resolving the overfitting issue in early works [29]. For modern neural networks, researchers proposed a number of training techniques such as Sparse Coding [30], Auto Encoder/Decoder [31], [32], and Deep Belief Network (DBN) [33], to prune redundant synapses and neurons without loss of accuracy. Recently reserchers have proposed a new pruning method that achieves the best sparsity of CNNs, i.e, 11.15% on AlexNet [9], 7.61% on VGG16 [34] (as shown in Table V where we report the detailed pruning results).

We classify sparsity into two categories: *static sparsity* and *dynamic sparsity*. For *static sparsity*, synapses (Figure 2 (b)) and neurons (Figure 2 (c)) are permanently removed from the network. *Dynamic sparsity* occurs when neurons have output values as 0 (Figure 2 (d)) that contribute nothing to follow-up neurons and thus can be taken as pruned. Since neuron outputs vary with input data, the sparsity is changing dynamically. Note that *static sparsity* consists of *neuron sparsity (SNS)* and *synapses sparsity (SSS)*, while *dynamic sparsity* only consists of *neuron sparsity (DNS)*.

Weight Encoding. To further compress data, researchers applied image compression techniques such as quantization [35], entropy coding [36] to the remaining weights. Recent work on Deep Compression [13] applied fine-grained pruning, quantization, and Huffman coding to obtain a 35× compression ratio on AlexNet. Another effective compression scheme is CNNpack [37]. It tackled the issue in the frequency domain and finally obtained a 40× compression ratio, thereby producing the highest CNNs compression ratio to the best of our knowledge (Table V). We describe the weight encoding procedure in Figure 3, which consists of quantization and entropy coding. First, a clustering algorithm (e.g., the K-means clustering)

We denote the sparsity as the ratio of the remaining neurons/synapses to the total neurons/synapses.

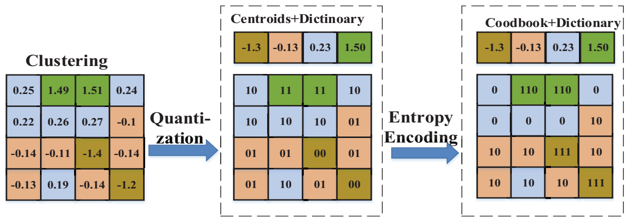


Fig. 3. Weight encoding via quantization and entropy coding.

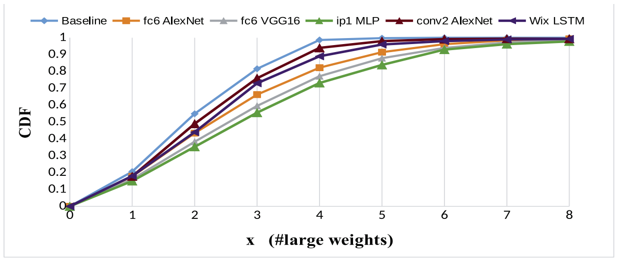
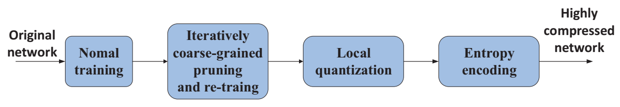


Fig. 4. CDF of the larger weights distribution.



Original network
Nomal training
Iteratively coarse-grained pruning and re-traing
Local quantization
Entropy encoding
Highly compressed network

Fig. 5. Compression flow.

clusters scattered weights into K clusters. Each cluster has a centroid value that has minimal total distance to all weights within the cluster. Therefore, each weight can be represented with its cluster centroid value and a unique index; thus, a dictionary with $log(K)$ bits per weight and a codebook with K unique values can represent the whole weights. Since the occurrence probabilities of elements in the codebook are unbalanced, the entropy coding (e.g., Huffman encoding) is employed to further reduce the bits representing the weights. It encodes the symbols with variable-length codewords where the more common symbols are represented with fewer bits.

B. Motivation

Irregularity. To leverage the benefit of reduced computation and memory in sparse networks, it is required to process sparsity efficiently. However, current processing platforms fail to process sparsity efficiently due the companied irregularity. In line with [16], we observe that for CPUs/GPUs, the performance of processing sparse networks is even worse than that of processing dense versions. Customized accelerators without sparsity support can not benefit from sparsity [5], [7]. A few accelerators supporting sparsity [16], [17], [18], [19], [20] still suffer from the irregularity. They either have notable costs for sparsity processing or fail to fully support the sparsity, as shown in Table I. Particularly, Cambricon-X [16] achieves 7.23× speedup and 6.43× energy saving over DianNao which does not support sparsity. The key feature of the accelerator is the Indexing Module (IM), which selects and transfers the needed neurons to connected PEs thus focusing on synapse sparsity only; but it takes up to 31.07% of the total area and 34.83% of the energy cost. Cnvlutin [17], an accelerator supporting neuron sparsity, improves performance by 1.37× with an area overhead of 4.49%. However, it fails to leverage the benefit of synapse sparsity. SCNN [20] exploits synapse sparsity and neuron sparsity, but it introduces in extra costs for storing and computing the coordinates, thus hindering the efficiency. SCNN achieves only 79% of the performance, but consumes 33% more energy when processing dense networks. It only gains 2.7× and 2.3× improvements in performance and energy efficiency from sparsity. EIE [18] supports *static sparsity* and *dynamic neuron sparsity*. It achieves 2.9×, 19× and 3× better throughput, energy efficiency, and area efficiency, respectively, when compared to DaDianNao [6]. However, it only targets the *fully-connected* layers of DNNs, which only takes a small ratio of the computation time of CNNs. ESE [19] is implemented on FPGA focusing on the sparse LSTM model. It only aims at the LSTM layer and is not suitable for the *convolutional* layer which dominates the computation of CNNs. As a result, existing architectures cannot well process irregularity and thus cause inefficiency.

Observation. Previous pruning approaches take synapses as independent elements and thus ignore the potential relationship among synapses. We fully analyze weight distributions in neural networks and observe an interesting phenomenon. While the weights are stored into matrices—such as in *fully-connected* layers, where each row contains weights connected to the corresponding input neuron and each column corresponds to an output neuron, local weights tend to gather into clusters, which we call *local convergence* (Figure 1). We exploit the local convergence as follows. First, we slide a window of size k across the weight matrix, and count the number of larger weights in the window. The larger weights are defined as those whose absolute value are in the top m% of the total weights. A window with x larger weights will be classified with a label of x. We select five representative layers—*fc6* in AlexNet, *fc6* in the VGG16, *ip1* in MLP, W_{ix} in LSTM and *conv2* in AlexNet—as driving examples. We set $k = 4$ for the sliding window size for the former four layers and $k = 2$ for the *conv2* layer. Additionally, We set $m = 10$ for larger weights. In Figure 4, we plot the cumulative distributions of above classification for the the five trained layers and an randomly initialized layer for comparison. We observe that in no case more than four larger weights are distributed in the block in the initialized layer. But, the five trained layers have more than *six* larger weights distributed in the block. That is, larger weights are more likely to gather during the training.

III. Compressed Neural Networks

In this section, we present the entire flow of our proposed compression algorithm, including the coarse-grained pruning, weight quantization, and entropy coding, as shown in Figure 5.

A. Coarse-grained pruning

Instead of pruning synapses independently, our proposed coarse-grained pruning prunes several synapses together. The synapses are firstly divided into blocks; a block of synapses will be permanently removed from the network topology if it meets specific criteria. We then employ the fine-tuning approach to retain the network accuracy. Note that we apply the coarse-grained pruning iteratively in training to achieve better sparsity and avoid the accuracy loss.

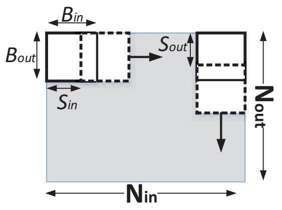


Fig. 6. Coarse-grained pruning in *fully-connected* layer.

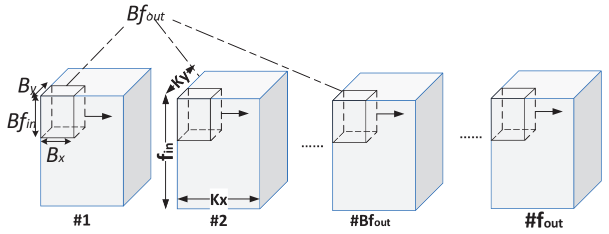


Fig. 7. Coarse-grained pruning in *Convolutional* layer.

1) Pruning methodology: To clearly explain the coarse-grained pruning technique, we use the *fully-connected* and the *convolutional* layers as driving examples. In a *fully-connected* layer, output neurons (N_{out}) are connected to input neurons (N_{in}) via synapses with independent weights $W_{i,j}$, i.e., the 2D matrix (N_{in},N_{out}) in Figure 6. During pruning, we slide the block of size $B_{in} \times B_{out}$ with a stride of size S_{in} and S_{out} along the two dimensions, respectively. Once the block meets certain conditions, all the synapses in the block will be pruned simultaneously. During the iterative pruning, the sliding block will jump over pruned synapses to have all the pruned blocks in the same size thus for simple indexing. Two pruning metrics can be applied, i.e., *max pruning* and *average pruning*. For *max pruning*, a block will be pruned if the weight with maximum absolute value is smaller than a predefined threshold (W_{th}). For *average pruning*, a block will be pruned if the average absolute value of all weights is smaller than a predefined threshold.

Regarding the *convolutional* layer, output neurons in an output feature map are connected to small windows of neurons in each input feature map via shared synapses. Thus, weights in the *convolutional* layer can be expressed as a four-dimensional tensor, i.e., (N_{fin}, N_{fout}, K_x, K_y), where N_{fin} is the number of input feature maps, N_{fout} is the number of output feature maps and K_x,K_y is the size of the convolution kernel, as shown in Figure 7. During pruning, we slide the block of size (B_{fin}, B_{fout}, B_x, B_y) with a stride of size S_{fin}, S_{fout} ,S_x and S_y along the four dimensions, respectively. *Max pruning* and *average pruning* can also be applied.

2) Pruning characteristics:

Block size. Finding the optimal block size that balances the compression ratio and accuracy is crucial. Pruning in an overly large block size will cause the inability to represent fine-grained weight connectivity, thus losing accuracy. Pruning in an overly small block size does not fully exploit the local convergence of weights, thus failing to achieve high sparsity. Note that if the block size is set 1 in each dimension, the coarse-grained pruning turns out to be the element-wise fine-grained pruning in [12].

In this paper, we prune different networks by carefully exploring the block sizes for different types of layers in neural networks. Initially, we should select the block sizes for different layers instead of different types of layers. But considering the extreme long training time in huge design space and efficiency for hardware, we focus on different types of layers in the network. For example, since weights in *convolutional* layers are more sensitive than that in *fully-connected* layers, we employ the coarse-grained pruning only in a certain dimension for *convolutional* layers. We use AlexNet as a driving example to illustrate the procedure clearly while maintaining the accuracy (top-1 error of 42.8%, in Table II). Here we use N to denote the pruning block size as $(1, N, 1, 1)$ and (N, N) in *convolutional* layers and *fully-connected* layers, respectively. In Table II, weights in *convolutional* layer and *fully-connected* layer are quantized with 8 bits and 4 bits, respectively, and then encoded with Huffman encoding. When N varies from 1 to 64, the compression ration (r_c) first increases rapidly from 40× to 79×, then drops rapidly to 65×. The compression ratio increases as block size grows from 1 to 16 because less memory is needed to store the indexes. The compression ratio drops when the block size grows from 16 to 64, because the sparsity increases rapidly in order to maintain the same accuracy, thus hindering the compression ratio. To best trade off the compression ratio and accuracy, we set the block size in the *convolutional* layer as $(1, 16, 1, 1)$, the block size in $fc6$, $fc7$ and $fc8$ as $(32, 32)$, $(32, 32)$ and $(16, 16)$, respectively. More importantly, with coarse-grained pruning, the index size is only 29.38KB, 102.82× reduction when compared to 2.95 MB using fine-grained pruning [13].

Neuron sparsity. In the coarse-grained pruning, neurons are not directly pruned as we only prune synapses. However, *dynamic sparsity* where the neuron values are zero takes a great ratio, especially for large networks. In Table III, we list static sparsity and dynamic sparsity as a comparison when averaging on the entire input dataset. In the large and deep neural networks, e.g., AlexNet, VGG16 and ResNet-152, the *static neuron sparsity* is rather high, i.e., 100% in *convolutional* layers. However, the *dynamic neuron sparsity* is promising—62.37% in AlexNet and 40.52% in VGG16, which offers a great opportunity for performance gain and energy saving.

Avg. vs. max pruning. *Average pruning* and *max pruning* are two straightforward coarse-grained pruning strategies. The main characteristics of the two pruning strategies are quite different.

TABLE II
SPARSITY AND COMPRESSION RATIO OF ALEXNET USING DIFFERENT PRUNING BLOCK SIZES (C: *convolutional* LAYER; F: *full-connected* LAYER; r_c: OVERALL COMPRESSION RATIO).

N	1	2	4	8	16	32	64
C:W%	35.01	35.01	35.11	35.23	35.25	40.05	45.55
F:W%	10.01	10.02	10.02	10.03	10.05	10.09	12.22
Weight (MB)	2.86	2.86	2.87	2.87	2.91	3.01	3.59
Index (MB)	2.95	0.76	0.22	0.07	0.03	0.01	0.005
r_c	40×	64×	75×	79×	79×	77×	65×

TABLE III
SPARSITY IN NNS.

–	LeNet5	MLP	Cifar10	AlexNet	VGG16	ResNet152
C: SSS (%)	11.02	–	7.92	35.25	35.17	54.31
SNS (%)	74.61	–	88.51	100	100	100
DNS (%)	100.00	–	69.39	62.37	40.52	49.70
F: SSS (%)	8.53	9.87	6.01	10.07	4.84	100
SNS (%)	52.30	59.75	42.27	84.25	77.82	100
DNS (%)	88.50	33.69	80.07	60.73	56.97	75.90

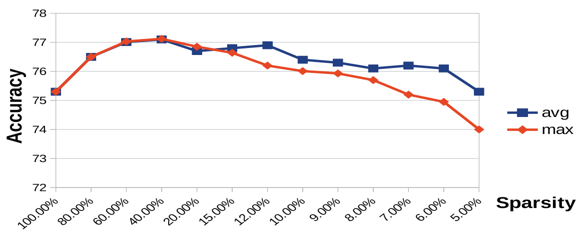


Fig. 8. Max pruning vs. avg pruning on Cifar10 quick model.

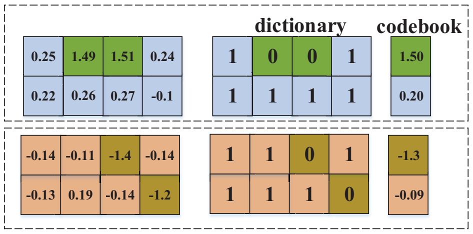


Fig. 9. Local quantization.

The *max pruning* indicates that only the largest weight inside the block will dominate the importance of the block. The *average pruning* indicates that all the weights inside the block will contribute to the importance of the block. We compare the accuracy achieved by the two methods by training the Cifar10 quick model with the same block size. Figure 8 shows that when the sparsity is below 15%, *average pruning* achieves higher accuracy than *max pruning*. Thus, we select *average pruning* in this paper.

Measurement of irregularity. To have a direct view of the effect of coarse-grained pruning, we propose a simple but effective method to measure the reduced irregularity. It can be computed as

$$R(Irr) = JBIG(I_f)/JBIG(I_c) \tag{1}$$

where $R(Irr)$ indicates the reduced irregularity, I_f and I_c indicate the indexes of sparse neural networks after fine-grained pruning and coarse-grained pruning, respectively, $JBIG()$ indicates lossless image compression standard from the Joint Bi-level Image ExpertsGroup [38]. The basic idea is based on the fact that regular data (especially binary matrix) contains more redundant information and thus can be represented with less data. Therefore, we take the synapses indexes as binary images and compress them with JBIG. The data size after compression can measure the irregularity in a way. Thus, we measure the reduced irregularity through the ratio of compressed index size by coarse-grained pruning to that by fine-grained pruning.

B. Local quantization

Network quantization and weight sharing are proven to be efficient to reduce the number of bits required to represent the weight [13]. To better leverage the local convergence, we propose local quantization, which employs weight sharing in a local area of the weight matrix instead of the whole weight matrix. As illustrated in Figure 9, the local quantization will first divide the weight matrix into two sub-matrices and then perform clustering seperately. In each sub-matrix, weights will be encoded into a codebook and a dictionary with 1 bit per weight (2 bits per weight using global quantization in Figure 3). Compared with global quantization, local quantization is able to exploit the local convergence to further shrink the number of bits for representing weights, thus obtaining a higher compression ratio. The overhead of local quantization is minor. Taking the $fc6$ layer of AlexNet as an example, while global quantization achieves 5 bits per weight dictionary with a 128B codebook and weight size of 2.0MB, local quantization achieves 4 bits per weight dictionary with a 4KB codebook for 64 sub-matrices and weight size of 1.6MB (19.81% smaller).

C. Entropy encoding

The entropy encoding is a lossless data-compression scheme that creates and assigns a unique prefix-free code to each unique symbol. Since the length of each codeword is approximately proportional to the negative logarithm of the probability, more common symbols are encoded with fewer bits. Two of the most common entropy encoding techniques are Huffman coding [39] and arithmetic coding [40].

D. Compression results

The detailed compression results of our algorithm on seven representative neural networks are shown in Table IV, including LeNet-5 [21], MLP (3-layer MLP, 300×100 hidden neurons), Cifar10 quick model [41], AlexNet [9], VGG16 [34] , ResNet152 [42] and LSTM [43]. In Table V, we compare our method with two existing state-of-the-art neural network compression methods, i.e., Deep Compression [13] (fine-grained pruning) and CNNpack [37] as mentioned in Section II-A. Note that we only focus on the LSTM layers in the LSTM model. Our proposed algorithm achieves a very promising compression ratio compared to Deep Compression and CNNpack, with negligible accuracy loss and the same sparsity as Deep Compression. For large neural networks, e.g., AlexNet and VGG16, the compression ratio almost doubles than that of Deep Compression and CNNpack. For Deep Residual Network, our algorithm and Deep Compression can only obtain a less than 10× compression ratio which is much lower than traditional DNNs. We hypothesize that the novel features including the shortcut connections and the batch normalization,

TABLE IV
SPARSITY, COMPRESSION RATIO AND REDUCED IRREGULARITY ACHIEVED BY OUR ALGORITHM (W_p: WEIGHT SIZE AFTER PRUNING; W_q: WEIGHT SIZE AFTER PRUNING AND LOCAL QUANTIZATION; W_c: WEIGHT SIZE AFTER PRUNING, LOCAL QUANTIZATION AND ENCODING; L: *LSTM* LAYER; C: *convolutional* LAYER; F: *fully-connected* LAYER; r_p: COMPRESSION RATIO ACHIEVED BY PRUNING; r_q: COMPRESSION RATIO ACHIEVED BY PRUNING AND LOCAL QUANTIZATION; r_c: OVERALL COMPRESSION RATIO; $R(Irr)$: REDUCED IRREGULARITY).

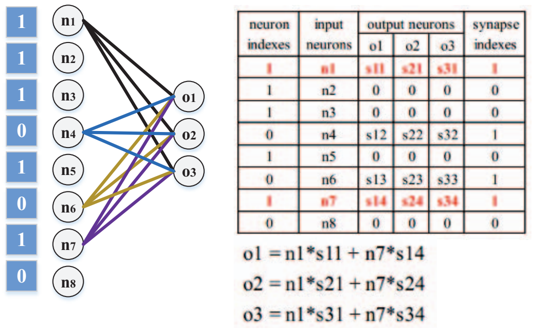
Model	Sparsity(%)	$W_p(B)$	r_p	$W_q(B)$	r_q	$W_c(B)$	r_c	$R(Irr)$
Alexnet	C: 35.25 F: 10.07	W: 25.65M I: 36.73K	9×	W: 3.60M I: 36.73K	64×	W: 2.90M I: 29.38K	79×	101.65×
VGG16	C: 35.17 F: 4.84	W: 42.72M I: 202.80K	12×	W: 7.80M I: 202.80K	66×	W: 5.25M I: 121.68K	98×	28.54×
LeNet5	C: 11.02 F: 8.53	W: 135.96K I: 1.67K	12×	W: 23.77K I: 1.67K	66×	W: 19.01K I: 1.39K	82×	8.87×
MLP	C: – F: 9.87	W: 102.54K I: 1.20K	10×	W: 19.23K I: 1.20K	51×	W: 12.01K I: 0.61K	82×	10.41×
Cifar10	C: 7.92 F: 6.01	W: 42.08K I: 0.48K	13×	W: 8.68K I: 0.48K	62×	W: 7.82K I: 0.42K	69×	7.61×
ResNet- -152	C: 54.31 F: 100.0	W: 134.10M I: 0.49M	1.7×	W: 33.50M I: 0.49M	7×	W: 23.44M I: 0.44M	10×	13.02×
LSTM	L: 12.56	W: 1.56M I: 25.96K	8.0×	W: 199.28K I: 25.96K	58×	W: 152.47K I: 17.84K	77×	50.51×

TABLE V

SPARSITY, COMPRESSION RATIO, AND ACCURACY COMPARISON (S%: SPARSITY; r_c: COMPRESSION RATIO).

model	Ref	Deep Cmp. [13]			CNNpack [37]			Ours		
	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)	S (%)	r_c	Top1-E(%)
AlexNet	42.78	11.15	35×	42.78	–	39×	41.60 (41.80*)	11.03	79×	42.72
VGG16	31.50	7.61	49×	31.17	–	46×	29.70 (28.50*)	8.07	98×	31.33
LeNet-5	0.80	8.43	39×	0.74	–	–	–	8.60	82×	0.95
MLP	1.64	8.18	40×	1.58	–	–	–	9.87	82×	1.91
Cifar10	24.20	5.02	45×	24.33	–	–	–	7.07	69×	24.22
ResNet152	25.00	55.00	8×	24.40	–	–	–	55.83	10×	25.05
LSTM	20.23	11.53	35×	20.52	–	–	–	12.56	77×	20.72

*Reference accuracies used in [37].



neuron indexes	input neurons	output neurons			synapse indexes
		o1	o2	o3	
1	n1	s11	s21	s31	1
1	n2	0	0	0	0
1	n3	0	0	0	0
0	n4	s12	s22	s32	1
1	n5	0	0	0	0
0	n6	s13	s23	s33	1
1	n7	s14	s24	s34	1
0	n8	0	0	0	0

o1 = n1*s11 + n7*s14

o2 = n1*s21 + n7*s24

o3 = n1*s31 + n7*s34

Fig. 10. A coarse-grained pruned layer (*left:* topology; *right:* corresponding connection table).

greatly reduce the overfitting. Thus, we can only get a very limited compression ratio on the Deep Residual Network.

We also compare the accuracy of our algorithm with Deep Compression and CNNpack in Table V where the reference accuracies are obtained from non-pruned networks. We can observe that the accuracy loss due to coarse-grained pruning is negligible, i.e., less than 0.2% on average compared to the reference accuracy, similar to Deep Compression (less than 0.1%) and less than CNNpack (0.7%).

In Table IV we also report the reduced irregularity by coarse-grained pruning. It is notable that coarse-grained pruning can reduce the irregularity by 20.13× on average. Also, consistent with intuition, the larger the block size is, the higher the reduced irregularity is. For small networks, e.g. LeNet-5, MLP and Cifar10, the reduced irregularity is 8.89× on average. Contrarily, large networks, e.g. AlexNet, VGG16 and LSTM, the reduced irregularity is much larger, 52.71× on average. Additionally, for ResNet, we can only prune it with small block size, thus 13.02× irregularity reduction. The results further validate that coarse-grained pruning can significantly reduce irregularity.

IV. DESIGN PRINCIPLES

Here we analyze the coarse-grained pruned network for accelerator design principles, using a *fully-connected* layer as a driving example (see Figure 10). First, indexes are shared among many output neurons. As shown in Figure 10, connections between input neurons $n2, n3$ and output neurons $o1, o2, o3$ are pruned together, as well as connections between input neurons $n5, n8$ and all output neurons. As output neurons share the same connection topology, they share the same indexing representation ("synapse indexes" in Figure 10).

Second, the selected input neurons, i.e., their values, are shared among multiple output neurons (e.g, $n1, n4, n6, n7$). Without loss of generality, given the pruning block size as (B_{in}, B_{out}) in the fully-connected layers, a group of B_{out} adjacent output neurons will share the same input neurons. The observation is consistent with the convolutional layers for B_{fout} adjacent output neurons, if given the block size as $(B_{fin}, B_{fout}, B_x, B_y)$.

Third, dynamic sparsity is crucial for high efficiency. As input neurons $n4, n6$ are zero, they contribute nothing to all the output neurons and can be ignored (i.e., dynamic sparsity). Regarding the example in Figure 10, the actual operations regarding to static sparsity are 12 multiplications, 9 additions with 16 input data, or 6 multiplications, 3 additions with 8 input data when considering additional dynamic sparsity. In comparison, the dense layer has 24 multiplications, 21 additions with 32 input data, which has 2.14×/5.00× more operations and 2×/4× more data than static sparsity/static+dynamic sparsity, respectively. Thus, it is crucial to leverage dynamic sparsity for high efficiency (almost 2× improvement in above example). Note that, even with dynamic sparsity, output neurons still share the indexes and the selected input neurons.

Fourth, the computations are balanced among the output neurons as they share the same input neurons. Each output neuron requires 2 multiplications and 1 addition for computation for the example in Figure 10, which can avoid load imbalance that will hinder the performance [19].

Therefore, we consider the following principles when designing an accelerator to maximumly utilize the efficiency. We reply on that the accelerator (1) should store and compute only the existing non-zero neurons and synapses, maximizing the usage of all computation operators; (2) should be able to leverage the shared indexes which require much smaller size for storage; (3) should be able to leverage the dynamic sparsity for further efficiency improvement; (4) should leverage the load balance among the adjacent output neurons.

V. ACCELERATOR ARCHITECTURE

In this section, we present the detailed architecture of our accelerator, which efficiently addresses the remaining irregularity of coarse-grained pruned sparse networks.

Overview. As depicted in Figure 11, we present the proposed accelerator architecture. Following the presented principles, we design a neuron selector module (NSM)—the key component of our accelerator—to process the *static* sparsity with shared indexes. We design a neural functional unit (NFU) that has multiple processing elements (PEs) to compute different output neurons in parallel. Each PE contains a local synapse selector module (SSM) to process the *dynamic* sparsity. The storage module consists of two neural buffers (NBin and NBout) and

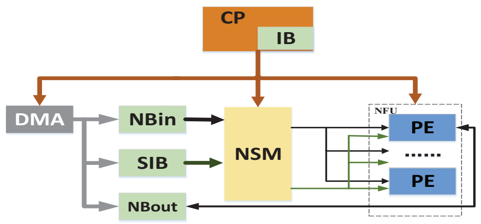


Fig. 11. Accelerator architecture.

a synapse index buffer (SIB). The control module consists of a control processor (CP) and an instruction buffer (IB). The CP decodes various instructions from IB efficiently into detailed control signals for all other modules. Here we define a VLIW-style instruction set for the accelerator.

A. *Processing with sparsity*

The accelerator is designed to exploit (1) static sparsity and (2) dynamic sparsity, as well as (3) the compression method for performance gain and energy reduction. In accelerator, sparsity is processed by NSM and NFU together. The NSM receives input neurons from the NBin and synapse indexes from the SIB, then produces the filtered neurons (*static* sparsity) and the indexing string that are broadcast to all the PEs in NFU. Each PE filters out the needed synapses locally (*dynamic* sparsity), thus avoiding useless computations and data transfers.

Indexing. Before elaborating the NSM and NFU, we explain clearly how we store and index the sparse data in accelerator. In order to retain the small size of indexes after coarse-grained pruning, we employ a high efficient indexing method. Particularly, we store the synaptic weights in a compact way. Only the existing synapses (non-pruned) are stored together with their corresponding indexes that indicate the connections between input and output neurons ("synapse indexes" in Figure 10). We store the neurons as they were stored in dense networks, i.e., "0" will be used as neuron value and stored if a neuron does not exist due to pruning or ReLU activation function. This is because neuron values vary with different inputs and neurons usually need a much smaller storage size than synapses. Thus, during sparse neural networks processing, accelerator needs to select neurons/synapses based on neuron values (*dynamic sparsity*) and synapse indexes (*static sparsity*).

NSM. The NSM module processes the *static* sparsity by selecting the needed input neurons, see Figure 12. To more efficiently process shared indexes and inputs from coarse-grained sparsity, we design a central NSM shared by multiple PEs to exploit the regularity-elevated sparsity. For the example in Figure 10 to select from 8 neurons with zero-valued $n4, n6, n8$, NSM produce the final *target string* to find the needed neurons $n1, n7$ and generate *indexing string* for later synapses selection. Specifically, *neuron indexes* will be computed firstly; each index indicates whether the value of the corresponding neuron is "0" ("11101010" in this case). Then an "AND" operation will be applied to *neuron indexes* and *synapse indexes* to form the *neuron flags* ("10000010"). *Neuron flags* will accumulate itself to have the *accumulated neuron flags* and "AND" operation will be applied between them to produce the *target string* ("10000020"). Additionally, *target string* will be used with

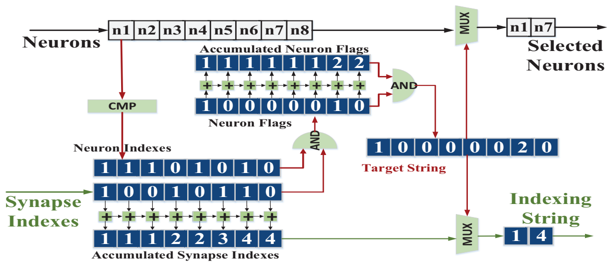


Fig. 12. Neuron selector module(NSM).

the accumulated synapse indexes to produce *indexing string*, which is used to select synapses and passed to the NFU. Note that, the selected neurons and indexing strings are shared by multiple output neurons, thus broadcast to the PEs in NFU.

NFU. The NFU processes all operations in neural networks efficiently as well as *dynamic* sparsity with T_n homogeneous processing elements (PEs). As shown in Figure 13, each PE consists of a local synapse buffer (SB), weight decoder module (WDM), synapse selector module (SSM) and neural function unit (PEFU). Each PE loads the compressed synapses from its local SB as synapses are independent among different output neurons and hence can be stored separately in PEs. A WDM with a Lookup-Table (LUT) is placed next to the SB to extract actual weights from the compressed values after local quantization. The SSM uses the indexing string from NSM and synapses from WDM to select the needed synapses for the PEFU, as shown in Figure 14 (a). Each PEFU consists of T_m multipliers, a T_m-in adder tree and a nonlinear function module, as shown in Figure 14 (b). We use a time-sharing method to map neural networks onto PEs, i.e., each PE works on an output neuron. Ideally, it takes $\lceil M/T_m \rceil$ cycles to compute the output neuron that needs M multiplications as the PEFU can finish T_m multiplications at a cycle. The NFU then collects and assembles T_n outputs from all PEs for later computation.

The SSM selects the needed synapses if *dynamic* sparsity exists. For generality, we store synaptic weights compactly regarding *static* sparsity. As shown in Figure 10, for each output neuron, the first and the fourth synapses ($S_{T_i 1}, S_{T_i 4}$) are the two needed synapses for computation. The SSM selects the needed synapses based on the *indexing string* from the NSM, which contains the locations of needed synapses, see Figure 14 (a). We enforce the "MUX" operation between the synapses and the *indexing string* to finally select the needed synapses. Since each PE works on different output neurons with different synapses, we can implement the SSM and SB locally inside the PE, thus avoiding high bandwidth and long latency.

The indexing part in our design differs from Cambricon-X [16] in three aspects. First, our accelerator contains a shared indexing module (NSM) to leverage the improved regularity of sparse synapses. As PEs share the same indexes of synapses due to coarse-grained pruning, the module for selecting neurons (NSM) is shared among many PEs, thus reducing the indexing module overhead (Section VII-A) and bandwidth requirement between NSM and PEs. Second, our accelerator has a local indexing module (SSM) inside each PE for leveraging neuron sparsity efficiently. Though the regularity

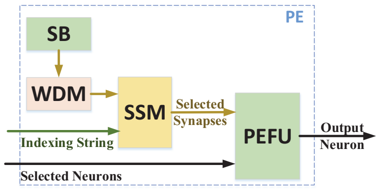


Fig. 13. The architecture of the PE.

of sparse synapses can be improved by coarse-grained pruning, there still exists irregularity of sparse neurons, e.g., ReLU can introduce many zeros to the output values of neurons. A synapse selector module is designed for minimizing the effects of such irregularity. Thus, our indexing module can leverage the neuron and synapse sparsity while IM can only exploit the synapse sparsity. Third, a WDM is introduced in the accelerator to exploit the local quantization that can further reduce the size of weights.

B. Storage

As the data processed in our accelerator have different behaviors, we split storage into four parts: an input buffer (NBin), an output buffer (NBout), a synapse index buffer (SIB) and T_m synapse buffers (SBs), as shown in Figure 11. In order to leverage the overlap between computation and DMA memory access, we implement the buffers in a ping-pong manner.

For NBin and NBout, we set the width as $16 \times T_m \times 16$-bit so as to provide $16 \times T_m$ neurons for NSM at a time. The reason is because of the efficiency. While T_n PEs share the same input neurons, PEs needs T_m input neurons at a time to avoid inefficiency caused by stall in computation. With $16 \times T_m$ neurons fed to NSM at a time, NSM is able to filter out T_m inputs for PEs if the sparsity is higher than 1/16. Nevertheless, existing networks have sparsity (including neuron and synapse sparsity) higher than 1/16 (6.25%)—especially in *convolutional* layers, which take high proportion in the total execution time. Thus, the efficiency will not be affected.

For the SB in each PE, we select the width as $T_m \times 64$-bit to provide $4 \times T_m$ different synaptic weights in consideration of *dynamic sparsity*. As the weights stored in SB are in compressed form (pruned and encoded via local quantization), the bit-width for synaptic weights varies across different layers—for example, 8 bits in AlexNet *convolutional* layer, 6 bits in MLP *fully-connected* layer and 4 bits in AlexNet *fully-connected* layer. To better leverage the varied bit-width and avoid drastic hardware overhead of decoding (i.e., WDM), we store the compressed weights aliasing to 4 bits. For example, a row (128 bits) in the SB will contain 32, 16, and 8 data if weights are stored with 4 bits, 8 bits and 16 bits, respectively. Thus, when $T_m \times 64$-bit data are loaded, WDM decodes it into $T_m \times 16$ weights if using less or equal to 4 bits, or $T_m \times 8$ if less than or equal to 8 bits, or $T_m \times 4$ if larger than 8 bits. Compared to a design of WDM that can support any bit-width, despite the complex data addressing patterns, we observe 5.14× area and 4.27× power savings of our implementation of the WDM.

For the SIB, we select the width as $16 \times T_m \times 1$-bit as we use a direct indexing format where each bit, "1" or "0", indicates whether the corresponding synapse exists or not. Thus, SIB

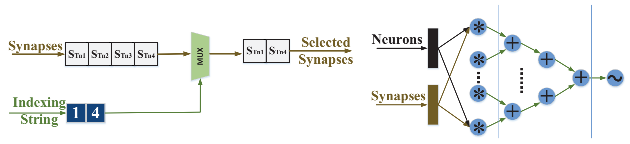


Fig. 14. (a) SSM. (b) PEFU.

provides $16 \times T_m \times 1$-bit indexes for the NSM to select the needed neurons with the corresponding $16 \times T_m$ input neurons.

The sizes of the four buffers are decisive to overall performance and energy consumption. Although previous works proposed offering large enough buffers to hold all synapses of neural network with moderate sizes [6], [18] so as to avoid costly off-chip memory accesses, it incurs considerable delay, area cost, energy penalty and unscalability for the emerging larger and deeper neural networks. Thus, we employ a small buffer size along with a proper data replacement strategy to best trade off among scalability, performance and energy consumption. After exploring different sizes of buffers, we deploy 8KB, 8KB, 32KB and 1KB for NBin, NBout, SBs and SIB, respectively. Only when all the neurons in NBin have been processed, or all the synapses in SB have been reused, or NBout is full, will the main memory be accessed for loading neurons or loading synapses or storing neurons, respectively.

C. Control

The CP is the root of our accelerator and controls the whole execution process. It decodes the instructions from the inner IB efficiently for execution coordination, memory accesses and data organization. The CP monitors the state of every module by setting the corresponding control registers. The highly efficient VLIW-style instructions are generated from our library-based compiler.

VI. Experimental Methodology

In this section, we introduce the experimental methodology. We implement our accelerator in RTL and then synthesize, place and route it with Synopsys toolchain using the TMSC 65nm library. We use CACTI 6.0 to estimate the energy consumption of DRAM accesses [44]. We evaluate the performance and energy costs using PrimeTime PX based on VCD waveform files obtained from backend simulation with typical corner process.

Baselines. We compare our design with three baselines: the CPU, the GPU, and hardware accelerators.

CPU: We use Caffe [45], the popular deep learning framework, and evaluate our benchmarks on a modern CPU which has 6 cores working at 2.1GHz at 22nm technology (denoted as CPU-Caffe). In order to adapt to sparse neural networks, we implement the evaluated benchmarks with the most widely used sparse library, i.e., sparseBLAS [46] (CPU-Sparse).

GPU: We use Caffe to evaluate our benchmarks on a modern GPU card, Nvidia K20M which has a 5 GB GDDR5, 3.52 TFlops peak at 28nm technology (GPU-Caffe). Furthermore, we natively use cuBLAS to implement our benchmarks for fair comparison (GPU-cuBLAS). For the sparse version, we implement the CSR indexing with state-of-the-art cuSparse library [47] (GPU-cuSparse).

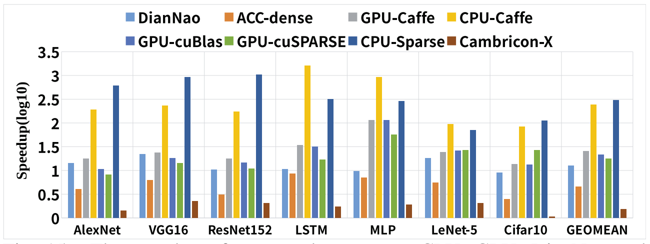


Fig. 15. The speedup of our accelerator over CPU, GPU, DianNao and Cambricon-X

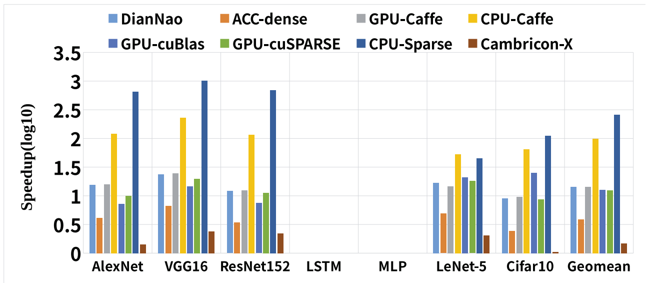


Fig. 16. The speedup of our accelerator over CPU, GPU, DianNao and Cambricon-X in *convolutional* layer

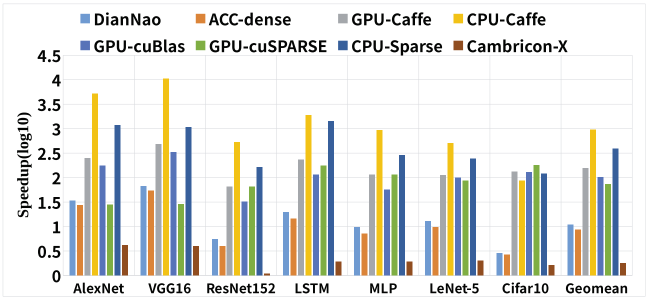


Fig. 17. The speedup of our accelerator over CPU, GPU, DianNao and Cambricon-X in *fully-connected* layer

Hardware accelerators: We compare our accelerator against two state-of-the-art network accelerators, DianNao [5], Cambricon-X [16]. DianNao aims to accelerate most of DNNs and CNNs within a small-footprint with high-throughput. Cambricon-X is an accelerator which aims to exploit sparsity in the sparse neural networks for performance gain and energy reduction. We select Cambrion-X as our baseline accelerator as it achieves the maximum utilization of sparsity and generality. Regarding the utilization of sparsity, averagely, compared to dense versions, Cambrion-X achieves 2.93× speedup; the numbers of Cnvlutin and SCNN are 1.37× and 2.7×, respectively. Regarding the generality, EIE is only capable on FC layers in [18]. We re-implement DianNao and Cambrion-X with 16×16 multipliers and 16 16-in adder-trees in same TSMC 65nm technology. Note that we assume all the accelerators are plugged to a main memory model allowing a bandwidth up to 256 GB/s.

Benchmarks. We use seven representative neural networks listed in Table IV with coarse-grained sparse representation and dense representation. Note that our implementation does not include the entropy encoding.

VII. Experimental Results

A. Hardware Characteristics

In the current implementation, we select $T_m = T_n = 16$ which is compatible with the pruning block size as discussed in Section III-D. We present the layout characteristics including area and power of our accelerator in Table VI. The accelerator has 53KB SRAM in total. It achieves 512 GOP/s working at a frequency of 1GHz in a small area of $6.73mm^2$ with only $798.55mW$ of total power. The area of our accelerator is 1.05× and 2.22× larger than Cambricon-X with $6.38mm^2$ and DianNao with $3.02mm^2$, respectively. The power is $155.45mW$ lower than Cambricon-X with $954mW$ and $313.55mW$ higher than DianNao with $485mW$.

TABLE VI
Hardware characteristics of accelerator.

	Area(mm^2)	%	Power(mW)	%
Total	6.73	100.00%	798.55	100.00%
NBin	0.55	8.17	93.32	11.69
NBout	0.55	8.17	93.32	11.69
SIB	0.05	0.74	6.89	0.86
NSM	0.69	10.26	121.46	15.21
CP	0.16	2.38	75.06	9.40
NFU	4.73	70.28	408.50	51.15
SB	1.05	22.20	151.91	37.19
SSM	0.25	5.29	56.80	13.90
WDM	1.54	32.56	16.25	3.98
PEFU	1.89	33.95	183.54	44.93

Additionally, the sparsity components take $2.48mm^2$ in area (36.85% of the total), 194.51mW in power (24.36% of the total), improving 1.71× performance and 1.37× energy efficiency over Cambricon-X. We observe that even with an additional module of selecting synapses (SSM, for *dynamic neuron sparsity*), the modules for indexing (i.e., NSM+SSM) achieve 1.87× power reduction ($178.26mW$ vs. $332.62mW$) and 2.11× area saving ($0.94mm^2$ vs. $1.98mm^2$), when compared against IM module in Cambricon-X. Note that the shared NSM module costs only 36.52% power and 34.85% area while achieving the same functionality of the IM module (indexing neurons). The WDM extracting weights from local quantization takes $1.54mm^2$ area and $16.25mW$ power.

B. Performance

We compare the performance of our accelerator against CPU, GPU, DianNao and Cambricon-X on the seven neural networks listed in Table IV with sparse representation and dense representation. On the CPU and GPU, we implement evaluated networks with both dense libraries for dense representation(i.e., CPU-Caffe, GPU-Caffe, and GPU-cuBLAS) and sparse libraries for sparse representation(i.e., CPU-Sparse and GPU-cuSparse). For fair comparison, we evaluate the performance of accelerator for dense representation (i.e., ACC-dense).

In Figure 15, we normalize all the performance numbers to that of our accelerator for sparse representation. Regarding dense representation, our accelerator achieves 44.8×, 5.8×, and 5.1× speedup over CPU-Caffe, GPU-Caffe and GPU-cuBLAS, respectively. Regarding sparse representation, our accelerator achieves 331.1× and 19.3× speedup over CPU-Sparse and GPU-cuSparse, respectively. Compared against the state-of-the-art accelerators DianNao and Cambricon-X, our accelerator achieves 13.10× and 1.71× speedup, which shows the high performance of our accelerator. Note that our accelerator can

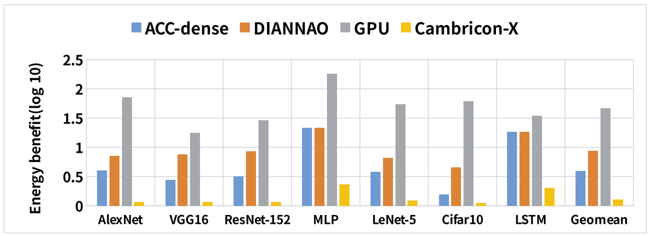


Fig. 18. The energy efficiency of our accelerator over GPU, DianNao and Cambricon-X

efficiently process not only the sparse networks, but also the dense networks.

Our accelerator achieves 4.32× speedup over ACC-dence, which comes from three aspects. First, NSM can fully exploit synapse sparsity (12.01% on average) to reduce the MAC operations, thus achieving 2.06× speedup. Second, SSMs exploit the neuron sparsity (55.41% on average) and achieve 1.44× speedup. Third, the reduced synapses by synapse sparsity (12.01%) and local quantization (2.76× reduction of synapse size) can greatly reduce the memory accesses of synapses, thus obtaining additional 1.46× speedup.

To further explore the performance efficiency, we show the performance comparison of the *convolutional* layer and *fully-connected* layer in Figure 16 and Figure 17. For the *convolutional* layer, our accelerator achieves 283.31×, 12.20×, and 13.94× speedup over CPU-sparse, GPU-cuSparse and DianNao, respectively. For the *fully-connected* layer, our accelerator achieves 531.89×, 79.05× and 13.56× speedup over CPU-sparse, GPU-cuSpare and DianNao, respectively.

It is notable that our accelerator achieves 1.66× and 2.15× speedup over Cambricon-X in the *convolutional* layer and *fully-connected* layer, respectively. The speedup in the *convolutional* layer benefits from *SSMs* exploiting dynamic neuron sparsity because the *convolutional* layer is computation-intensive. The dynamic neuron sparsity can greatly reduce MAC operations (e.g., 29M vs. 52M for conv3 layer in AlexNet with 55% dynamic neuron sparsity), thus greatly reducing execution time. While the speedup in the *fully-connected* layer benefits from the reduced memory accesses of synapses (local quantization) and indexes (index sharing) as the *fully-connected* layer is memory-intensive. The *WDM* can fully support weight quantization which reduces the storage capacity of synapses, thus obtaining 1.77× speedup. The index sharing by coarse-grained pruning can reduce the storage capacity of indexes, thus achieving additional 1.21× speedup.

C. Energy

We report the energy comparsion of GPU, DianNao, Cambricon-X, and our accelerator across the seven neural networks as shown in Figure 18, where we include the energy of off-chip memory accesses. Our accelerator achieves 49.60×, 9.16× and 1.37× better energy efficiency compared to GPU, DianNao and Cambricon-X, respectively. Moreover, we compare the energy costs of off-chip memory for our accelerator with and without local quantization and we observe 1.24× energy reduction on average by reduced data amount of local quantization. Regarding the accelerator energy costs without off-chip memory accesses, our accelerator achieves

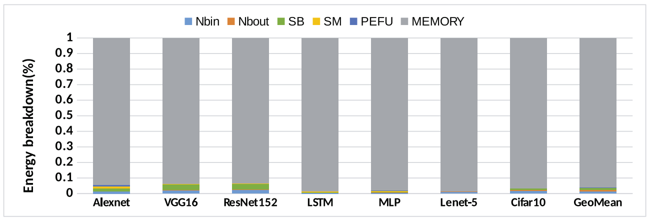


Fig. 19. The energy breakdown of our accelerator with off-chip memory accesses.

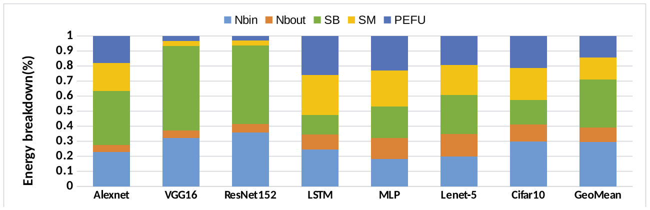


Fig. 20. The energy breakdown of our accelerator without off-chip memory accesses.

1169.51×, 12.30× and 1.75× better energy efficiency, respectively. The results demonstrate the high energy efficiency of our accelerator.

We further show the energy breakdown of our accelerator for all the evaluated networks as shown in Figure 19. We can observe that the main memory accesses consume more than 90% of the total energy. For LSTM and MLP network, the main memory accesses consume more than 98% of the total energy, much higher than other neural networks. It demonstrates these two kinds of networks are more memory intensive. This informs us that through sparsity and weight quantization, the off-chip memory access energy can be drastically reduced for LSTM and MLP. Moreover, the off-chip memory access energy is also reduced due to the sparsity, i.e., 72.6% reduction when compared to dense networks.

We also show the energy breakdown of our accelerator without off-chip memory accesses in Figure 20. The on-chip memory accesses (including SB, NBin and NBout) consume about 70% of the total energy, which shows that the on-chip memory accesses energy still dominates the on-chip energy consumption. For LSTM and MLP, the on-chip memory accesses consume less than 60% of the total on-chip energy as the synapses are not reused in the two networks. For very deep convolutional networks, such as VGG16 and ResNet152, the ratio of on-chip memory accesses energy is more than 90% due to the complicated loop tiling and data reuse strategy [48] in convolutional layers. It is notable that that the on-chip SB energy reduces by 2.48× with 2.76× reduction of synapse size on average when compared with Cambricon-X.

D. Discussion

Entropy coding. We also investigate the accelerator supporting the entropy coding. The implementation of an entropy decoding module at TSMC 65nm reports $6.781 * 10^{-3} mm^2$ area cost with a throughput of one datum per cycle. As the variable-length coding nature of entropy coding, the decoding module has to process decoding sequentially. Even if we can divide the data into a number of data streams and apply entropy decoding in parallel, it will introduce tremendous area and power costs in the accelerator. For example, for each SB which provides $T_m \times 4$

data at a cycle in each PE, it needs $T_m \times 4$ entropy decoding modules to avoid performance loss. Thus, in total 1024 entropy decoding modules are needed with $6.94mm^2$ additional area cost and $971.37mW$ additional power cost for $T_n = T_m = 16$ (same as the implementation in this paper). The total area and power of the accelerator is $13.67mm^2$ and $1769.92mW$, respectively, which is $2.03\times$ larger and $2.22\times$ higher when compared to the design without entropy coding. However, we observe no performance gain in the convolutional layer and only $1.18\times$ performance gain in the fully-connected layer, which is very limited compared to the additional costs of large area and power. Thus, we do not include entropy coding in our accelerator.

Sparsity sensitivity. We investigate the sparsity sensitivity of our accelerator as shown in Figure 21 where we vary the neuron sparsity and coarse-grained synapse sparsity separately. We made several interesting observations. 1) The maximum speedup of the accelerator on convolutional layer is approaching the ideal speedup ($15.5\times$ vs. $16\times$). Compatible with most of neural networks (Table III), the NSM is designed to select 16 out of 256 neurons, thus achieving $16\times$ speedup at most. Our accelerator achieves nearly the ideal speedup by hiding the DMA memory access behind the computation with the help of ping-pong buffering. It is notable that our accelerator outperforms Cambricon-X at the same synapse sparsity in the convolutional layer, as our accelerator can fully exploit the data reuse and load balance from coarse-grained sparsity. 2) Our accelerator can easily achieve performance gain even when the sparsity is only 95% on the fully-connected layer, and the performance can be greatly improved with the decreasing sparsity ($59.59\times$ given the sparsity as 1%). The compressed synapses greatly reduce the memory access time, thus greatly reducing the total execution time in the fully-connected layer. 3) With dense synapses (100% sparsity), our accelerator achieves at most $3.9\times$ speedup by exploiting neuron sparsity, approaching the ideal speedup. As the neuron sparsity is beyong 25% for current convolutional layers (Table III), the SSM is designed to select 16 out of 64 synapses (25% neuron sparsity), thus $4\times$ speedup at most, which is sufficient. 4) The neuron sparsity does not bring performance gain in the fully-connected layer, as the memory access time for the synapses dominates the total execution time and the neuron sparsity does not reduce the memory accesses. The above observations further validate that our accelerator can well exploit both neuron sparsity and synapse sparsity of modern neural networks.

Reduced irregularity. The reduced irregularity can benefit the accelerator in two aspects. First, the reduced irregularity can simplify the accelerator design. Instead of multiple distributed NSMs, one shared NSM is sufficient, saving $10.35mm^2$ area and $1821.9mW$ power consumption. Similarly, a shared SIB, instead of 16 independent SIBs, reduces $15KB$ SRAM size. Second, the reduced irregularity reduces the memory accesses of synapse indexes greatly—$26.83\times$ reduction of synapse indexes—thus achieving $1.06\times$ speedup and $1.11\times$ better energy efficiency.

Input sensitivity. We investigate the input sensitivity across different datasets and believe that pruned network is partially

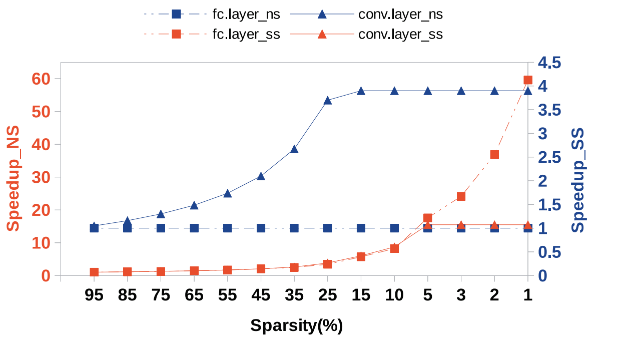


Fig. 21. Speedup of sparse version over dense version (NS: neural sparsity; SS: synapse sparsity)

sensitive to input datasets. To begin with, we immigrate a trained pruned VGG16 to Faster-RCNN on PASCAL-VOL-2007 dataset and observe a 63% mAP (vs. 73% mAP). However, by only immigrating first 8 layers, Faster-RCNN recovers to 73% mAP. Without loss of generality, with immigrating the first half layers to a different dataset, the network will not suffer from accuracy loss. The observation is consistent with ResNet152 immigrated to Faster-RCNN on PASCAL-VOL-2007 dataset and VGG16 immigrated to the Deep-Face-Recog on LFW dataset. On the other hand, the coarse-grained pruning/training procedure only need to be done off-line periodically for many scenarios. Thus, input sensitivity will not be critical as the costs for retraining for new data is acceptable.

Long-tuning process. In order to obtain the optimal compression ratio, we have to explore the parameters (block size, weight thresholds, quantization bits, etc.) for different networks. Thus it is a design space exploration (DSE) problem which may need a long-tuning process. But the training/tuning would only need to be performed periodically. Moreover, without loss of generality, we observe that block size of (1,16,1,1) and 8-bit local quantization are sufficient for most convolutional layers with reasonable tradeoff between compression ratio and accuracy.

Similar ideas as coarse-grained sparsity. Recent researchers also work on similar ideas as coarse-grained sparsity, such as SIMD-aware weight pruning [49], synapse vector elimination [50], channel reduction and filter reduction [51], [52]. Although those techniques can benefit the accelerator design by reducing the irregularity, it usually causes notable accuracy loss, as indicated in [53]. [54] explored a range of structured sparsities, including fine-grained sparsity, vector-level sparsity, kernel-level sparsity and filter-level sparsity, and evaluated the trade-off between the model's regularity and accuracy. Note that our proposed pruning approach does not limit the scale of pruning block, thus a generalized version of previous works. By specifying the parameters, coarse-grained pruning can achieve the same effect as those forms of sparsity. Despite that, to the best of our knowledge, our accelerator is the first accelerator that is able to leverage the coarse-grained sparsity.

TABLE VII
PERFORMANCE COMPARISON AGAINST EIE *(microsecond)*.

layer	AlexNet			VGG16			Geomean
	fc6	fc7	fc8	fc6	fc7	fc8	
EIE	30.30	12.20	9.90	34.40	8.70	7.50	–
ACC	18.43	8.19	5.13	25.23	4.12	5.09	–
Speedup	1.64×	1.49×	1.93×	1.36×	2.11×	1.49×	1.65×

Other accelerators. EIE [18] leverages the sparsity of *fully-connected* layers in neural networks with CSC sparse representation scheme. It uses very large SRAM to store all the synapses of the *fully-connected* layer on SRAM, thus eliminating off-chip memory accesses to save power. The *fully-connected* layers of AlexNet fit into EIE consumes total of $40.8mm^2$ which is $5.07\times$ larger than our accelerator. Instead of fixed quantization in EIE, we implemented WDM in accelerator, which allows self-defined bit-width for data representations of NNs for accuracy trade-off. For fair comparison of performance to EIE, we assume that our accelerator can store all the synapses of the *fully-connected* layer and only focus on the computation time. As shown in Table VII, our accelerator on average achieves $1.65\times$ speedup over EIE in *fully-connected* layers. SCNN [20] only improves the performance and energy efficiency by $2.7\times$ and $2.3\times$, respectively, over the dense accelerator. While our accelerator achieves $4.32\times$ and $5.10\times$ better in terms of performance and energy efficiency, respectively, over the dense version, which shows the high performance and efficiency of our accelerator.

VIII. Related work

Large neural networks are typically over-parameterized and the large number of neurons and synapses severely hinder efficient NN processing. Some effective algorithms have been developed to tackle the challenging problem. [55] utilizes sparse decomposition to reduce the redundancy of weights and computational complexity of CNNs. [15], [56] proposed BinaryNet whose weights were -1/1 or -1/0/1. [57] proposed a network architecture using the ”hashing trick”. [58] showed that weight matrices can be reduced by low-rank decomposition approaches. Works in [59], [60], [61], [62] directly pruned neurons based on some specific criteria, i.e., static neuron sparsity. However, it usually causes noticeable accuracy drops compared with synapse pruning at the same sparsity, therefore can not achieve high sparsity. For example, Network Trimming [61] achieves a sparsity ratio of 37%/26% for VGG16/LeNet-5 (ours 8.07%/8.60%). Data-free Parameter Pruning [60] achieves a sparsity ratio of 65.11%/16.5% for AlexNet/LeNet-5 (ours 10.03%/8.60%). Diversity Networks [62] did not provide detailed sparsity and only evaluated on small networks with dataset MNSIT/Cifar10. Optimal Brain Damage [63] and Optimal Brain Surgeon [64] prune networks to reduce the number of connections *(static synapse sparsity)* based on the Hessian of the loss function. Deep Compression [13] applies pruning *(static sparsity)*, quantization and Huffman encoding to obtain a greater than $35\times$ compression ratio on AlexNet. CNNpack [37] prunes synapses in the frequency domain with the help of DCT, obtaining a $39\times$ compression ratio, thereby producing state-of-the-art CNNs compression ratio. Different from the above methods, our compression methods fully exploit local convergence in neural networks, thus obtaining a much higher compression ratio.

There are many ASIC-implemented accelerators for neural networks. DianNao [5] accelerates various CNNs and DNNs in a small foot-print with high-throughput. DaDianNao [6] supplies sufficient on-chip memory for efficiently processing large-scale neural networks. ShiDianNao [7] fully exploits neuron reuse and synapse reuse in the *convolutional* layer, thus completely eliminating off-chip memory accesses. Farabet *et al.* [65] proposed a systolic architecture called NeuFlow architecture and Chakradhar *et al.* [66] designed a systolic-like coprocessor to handle 2D convolution efficiently. Although the aforementioned accelerators can achieve high throughput with low energy, they cannot exploit the sparsity and irregularity of modern compressed neural networks. Cambricon-X exploits synapse sparsity with IM selecting and transferring needed neurons, but it fails to exploit dynamic neuron sparsity. Cnvlutin [17] aims to exploit dynamic neuron sparsity, but it cannot exploit synapse sparsity. Above works can either leverage *static sparsity* or *dynamic sparsity*, but fail to benefit from both. EIE [18] can leverage the neuron sparsity and synapse sparsity at the same time, but it only aimed for the *fully-connected* layer. Recent accelerator SCNN [20] exploits both synapse sparsity and neuron sparsity but has limited improvements. Overall, our accelerator is better in terms of performance and energy efficiency, thus a more efficient design.

IX. Conclusions

In this paper, we propose a generalized coarse-grained pruning technique which can exploit local convergence in neural networks and reduce the irregularity drastically ($20.13\times$ on average). The coarse-grained pruning together with weight encoding obtains a greater than $79\times$ compression ratio on AlexNet and $98\times$ on VGG16. We further design a hardware accelerator, Cambricon-S, to well leverage the benefits of our compression method and exploit both dynamic neuron sparsity and synapse sparsity. The key feature of the accelerator is the NSM and SSMs for selecting neurons and synapses. Benefit from the coarse-grained pruning, the selected neurons are shared by PEs, thus reducing inner bandwidth. With a footprint of $6.73mm^2$ with $798.55mW$, our accelerator can achieve peak performance 512GOP/s. Compared with Cambricon-X, our accelerator achieves $1.71\times$ and $1.37\times$ better performance and energy efficiency, respectively.

X. Acknowledgments

This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA07009022017YFB1003101), the NSF of China (under Grants 61472396, 61432016, 61473275, 61522211, 61532016, 61521092, 61502446, 61672491, 61602441, 61602446, 61732002and 61702478), the 973 Program of China (under Grant 2015CB358800), National Science and Technology Major Project (2018ZX01031102) and Strategic Priority Research Program of Chinese Academy of Sciences (XDBS01050200).

References

[1] V. Mnih and G. E. Hinton, “Learning to label aerial images from noisy data,” in *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, 2012.

[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos *et al.*, “Deep speech 2: End-to-end speech recognition in english and mandarin,” *arXiv preprint arXiv:1512.02595*, 2015.

[3] H. Ze, A. Senior, and M. Schuster, "Statistical parametric speech synthesis using deep neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013.

[4] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for natural language processing," *arXiv preprint arXiv:1606.01781*, 2016.

[5] 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 international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2541967

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

[7] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[8] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "Dlau: A scalable deep learning accelerator unit on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36.

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

[10] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013.

[11] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of The 30th International Conference on Machine Learning*, 2013.

[12] 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*, 2015.

[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[14] J. L. Holi and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Transactions on Computers*, vol. 42, 1993.

[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016.

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

[17] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016.

[19] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017.

[20] P. Angshuman, R. Minsoo, M. Anurag, P. Antonio, V. Rangharajan, K. Brucek, E. Joe, K. Stephen, and J. D. William, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *In 44th International Symposium on Computer Architecture*, 2017.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, 1998.

[22] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007.

[23] R. Salakhutdinov and G. E. Hinton, "Learning a nonlinear embedding by preserving class neighbourhood structure." in *AISTATS*, vol. 11, 2007.

[24] B. A. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, vol. 1, 1989.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, 1997.

[26] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, 2014.

[27] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014.

[28] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001.

[29] L. Y. Pratt, *Comparing biases for minimal network construction with back-propagation*. Morgan Kaufmann Pub, 1989, vol. 1.

[30] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, 1996.

[31] Y.-l. Boureau, Y. L. Cun *et al.*, "Sparse feature learning for deep belief networks," in *Advances in neural information processing systems*, 2008.

[32] H. Lee, C. Ekanadham, and A. Y. Ng, "Sparse deep belief net model for visual area v2," in *Advances in neural information processing systems*, 2008.

[33] H. Lee, A. Battle, R. Raina, and A. Y. Ng, "Efficient sparse coding algorithms," *Advances in neural information processing systems*, vol. 19, 2007.

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

[35] M. Henneaux and C. Teitelboim, *Quantization of gauge systems*. Princeton university press, 1992.

[36] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[37] Y. Wang, C. Xu, S. You, D. Tao, and C. Xu, "Cnnpack: Packing convolutional neural networks in the frequency domain," in *Advances In Neural Information Processing Systems*, 2016.

[38] "Coded representation of picture and audio information progressive bi-level image compression," in *ISO/IEC International Standard 11544:ITU-T Rec.T.82*, 1993.

[39] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, 1952.

[40] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, 1987.

[41] A. Krizhevsky, "cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks," 2012.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[43] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling." in *Interspeech*, 2014.

[44] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007.

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

[46] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, 2002.

[47] NVIDIA, "The nvidia cuda sparse matrix library (cusoarse)."

[48] 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*, vol. 52, 2017.

[49] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[50] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[51] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in *Advances in Neural Information Processing Systems*, 2016.

[52] V. Lebedev and V. Lempitsky, “Fast convnets using group-wise brain damage,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[53] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning filters for efficient convnets,” *arXiv preprint arXiv:1608.08710*, 2016.

[54] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, “Exploring the regularity of sparse structure in convolutional neural networks,” *arXiv preprint arXiv:1705.08922*, 2017.

[55] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” *arXiv preprint arXiv:1412.6115*, 2014.

[56] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.

[57] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, “Compressing neural networks with the hashing trick.” in *ICML*, 2015.

[58] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting linear structure within convolutional networks for efficient evaluation,” in *Advances in Neural Information Processing Systems*, 2014.

[59] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu, “Reshaping deep neural network for fast decoding by node-pruning,” in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014.

[60] S. Srinivas and R. V. Babu, “Data-free parameter pruning for deep neural networks,” *arXiv preprint arXiv:1507.06149*, 2015.

[61] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, “Network trimming: A data-driven neuron pruning approach towards efficient deep architectures,” *arXiv preprint arXiv:1607.03250*, 2016.

[62] Z. Mariet and S. Sra, “Diversity networks,” *arXiv preprint arXiv:1511.05077*, 2015.

[63] Y. LeCun, J. S. Denker, S. A. Solla, R. E. Howard, and L. D. Jackel, “Optimal brain damage.” in *NIPs*, vol. 2, 1989.

[64] B. Hassibi, D. G. Stork, and G. J. Wolff, “Optimal brain surgeon and general network pruning,” in *Neural Networks, 1993., IEEE International Conference on*. IEEE, 1993.

[65] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, “Neuflow: A runtime reconfigurable dataflow processor for vision,” in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011.

[66] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010.