Addressing Sparsity in Deep Neural Networks

Xuda Zhou, Zidong Du, *Member, IEEE*, Shijin Zhang, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen, *Senior Member, IEEE*

***Abstract*—Neural networks (NNs) have been demonstrated to be useful in a broad range of applications, such as image recognition, automatic translation, and advertisement recommendation. State-of-the-art NNs are known to be both computationally and memory intensive, due to the ever-increasing deep structure, i.e., multiple layers with massive neurons and connections (i.e., synapses). Sparse NNs have emerged as an effective solution to reduce the amount of computation and memory required. Though existing NN accelerators are able to efficiently process dense and regular networks, they cannot benefit from the reduction of synaptic weights. In this paper, we propose a novel accelerator, Cambricon-X, to exploit the sparsity and irregularity of NN models for increased efficiency. The proposed accelerator features a processing element (PE)-based architecture consisting of multiple PEs. An *indexing module* efficiently selects and transfers needed neurons to connected PEs with reduced bandwidth requirement, while each PE stores irregular and compressed synapses for local computation in an asynchronous fashion. With 16 PEs, our accelerator is able to achieve at most 544 GOP/s in a small form factor (6.38 mm^2 and 954 mW at 65 nm). Experimental results over a number of representative sparse networks show that our accelerator achieves, on average, 7.23× speedup and 6.43× energy saving against the state-of-the-art NN accelerator. We further investigate possibilities of leveraging activation sparsity and multi-issue controller, which improve the efficiency of Cambricon-X. To ease the burden of programmers, we also propose a high efficient library-based programming environment for our accelerator.**

***Index Terms*—Accelerator, architecture, deep neural networks (DNNs), sparsity.**

Manuscript received December 6, 2017; revised May 2, 2018; accepted June 20, 2018. Date of publication August 8, 2018; date of current version September 18, 2019. This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFA0700902, Grant 2017YFA0700900, and Grant 2017YFB1003101, in part by the NSF of China under Grant 61472396, Grant 61432016, Grant 61473275, Grant 61522211, Grant 61532016, Grant 61521092, Grant 61502446, Grant 61672491, Grant 61602441, Grant 61602446, Grant 61732002, and Grant 61702478, in part by the 973 Program of China under Grant 2015CB358800, in part by the National Science and Technology Major Project under Grant 2018ZX01031102, and in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDBS01050200. This paper was recommended by Associate Editor H. Li. *(Corresponding author: Yunji Chen.)*

X. Zhou is with the School of Computer Science and Technology, University of Science and Technology of China, Hefei 230000, China, also with the Intelligence Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with Cambricon Tech. Ltd., Beijing 100191, China (e-mail: anycall@mail.ustc.edu.cn).

Z. Du, S. Zhang, S. Liu, Q. Guo, and T. Chen are with the Intelligence Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with Cambricon Tech. Ltd., Beijing 100191, China (e-mail: duzidong@ict.ac.cn; zhangshijin@ict.ac.cn; liushaoli@ict.ac.cn; guoqi@ict.ac.cn; chentianshi@ict.ac.cn).

L. Zhang and H. Lan are with the Intelligence Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, also with Cambricon Tech. Ltd., Beijing 100191, China, and also with the State Key Laboratory of Computer Architecture and Intelligence Processor Research Center, University of Chinese Academy of Sciences, Beijing 100190, China (e-mail: zhanglei01@ict.ac.cn; lanhuiying@ict.ac.cn).

L. Li is with the Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: liling@iscas.ac.cn).

Y. Chen is with the Intelligence Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the CAS Center for Excellence in Brain Science, Chinese Academy of Sciences, Beijing 100190, China (e-mail: cyj@ict.ac.cn).

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

Digital Object Identifier 10.1109/TCAD.2018.2864289

I. Introduction

Due to stringent energy constraints, hardware accelerators have emerged as an energy efficient alternative of CPUs and GPUs [1]–[6]. Traditionally, accelerator is thought to target a narrow application scope. However, recent investigations in both academia and industry have shown that a small set of algorithms such as neural networks (NNs) have been state-of-the-art across a broad range of applications including image/video/audio recognition, automatic translation, advertisement recommendation, and so on [7]–[10], which makes the NN accelerators possible to achieve a promising tradeoff between efficiency and practicability. Hence, researchers have proposed a number of NN accelerators [3], [11]–[14].

However, existing accelerators may suffer from extremely large sizes of NNs, especially considering that the sizes of NN continue to increase for better accuracy. For example, AlexNet, proposed by Krizhevsky *et al.* [15] in 2012, has 650 kilo neurons, and the number further increased to ~1 million as reported by Le *et al.* [16], or even several millions reported by Coates *et al.* [17] in 2013. The amounts of synaptic weights are even much higher: 60 million in [15], 1 billion in [16], and 10 billion in [17]. As the large amounts of synaptic weights incur intensive computation and memory accesses, efficiently processing large-scale NNs with existing NN accelerators remains a challenging problem.

To address the challenge of overwhelming neurons and synapses, researchers have proposed a number of effective techniques to make an NN sparse (i.e., reducing the number of neurons and synapses) while maintaining the accuracy of the original NN, including dropout in training [18], sparse representation [19]–[21], and sparsity cost function [19], [20], [22]. Fig. 1 shows the neurons and synapses of a fully connected multilayer perceptron (MLP), as well as its sparse counterpart after pruning. In the sparse MLP, as the values of a number of synapses are zero, such synapses can be removed from the perspective of computation. After synapse pruning, the neurons without input or output connections can be removed as well. Thus, the neurons and synapses in the sparse NN are much fewer than the original dense MLP. Recently, Han *et al.* [23] have proposed a pruning technique to shrink the amount of synaptic weights by about 10× with negligible accuracy loss.

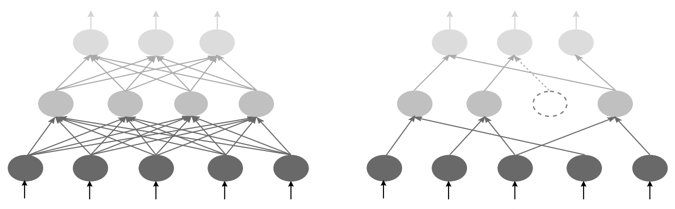


Fig. 1. (a) Fully connected MLP. (b) Sparse MLP.

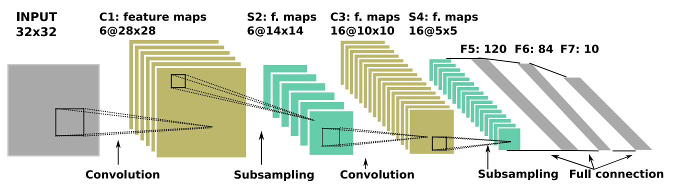


Fig. 2. Typical CNN: LeNet-5.

Interestingly, dramatically reducing the amount of synapses does not necessarily improve the performance and energy efficiency of existing accelerators, which are good at processing regular and dense NNs but lack of dedicated support for irregular and sparse models. For example, by using a state-of-the-art sparse library such as cuSPARSE [24], we found that the GPU can only process a sparse AlexNet with 6.99 million synaptic weights 1.78× faster than the original AlexNet with 59.48 million synaptic weights. A state-of-the-art NN accelerator, DianNao [11], even cannot benefit from the sparsity of NNs at all, since all the pruned synaptic weights still have to be fed into the accelerator with zero values for unnecessary computation.

In this paper, we propose a novel accelerator which can efficiently cope with not only original dense NNs but also heavily pruned sparse ones.[1] The accelerator features a processing element (PE)-based architecture consisting of multiple PEs companied with a buffer controller (BC), so as to exploit the sparsity and irregularity of NN models. Specifically, the BC integrates an efficient indexing module (IM) for selecting only needed neurons from centralized neuron buffers, and then transfers such neurons to connected PEs with reduced bandwidth requirement. After receiving such neurons, the PEs can perform efficient computation with locally stored compressed synapses. Moreover, due to irregular distribution of synapses, multiple PEs can work in an asynchronous fashion to gain increased efficiency. To ease the burden of programmers, we proposed a library-based programming environment for our accelerator with many optimization techniques, including loop tiling and data reuse strategy. Then, we design the neuron encoder/decoder to encode/decode neurons dynamically to leverage activation sparsity, thus reducing the costly DRAM memory accesses. Furthermore, we optimize the PE design to leverage the activation sparsity by skipping the unnecessary computations. Additionally, we propose a multi-issue controller that can execute the data movement instructions and computation instructions in parallel for potential higher performance.

We evaluate our accelerator, Cambricon-X, with a number of representative NNs (including LeNet-5 [25], AlexNet [15], VGG16 [26], etc.) with various sparsity levels. Compared against the state-of-the-art NN accelerator, DianNao, on average, our accelerator achieves 7.23× speedup and 6.43× energy reduction, at the cost of 954 mW power and 6.38 mm^2 area consumption. Moreover, compared against the GPU with the sparse library (i.e., cuSPARSE), on average, our accelerator achieves 10.60× speedup and 29.43× energy reduction. Compared against the CPU with the sparse library (i.e., Sparse BLAS [27]), on average, our accelerator achieves 144.41× speedup. Leveraging activation sparsity can improve the energy efficiency of DRAM memory accesses and PEFUs by 1.28× and 1.81×, respectively. The multi-issue controller achieves additional 1.16× performance gain on convolutional layers by parallelizing the data movement instructions and computation instructions.

II. Background and Motivation

In this section, we first introduce primer on NNs, including state-of-the-art NNs (e.g., CNNs and DNNs) and sparse NNs. Then, we present the motivation of building custom accelerator for sparse NNs.

A. Primer on Neural Networks

1) State-of-the-Art Neural Networks: The state-of-the-art NN algorithms are convolutional NNs (CNNs) and deep NNs (DNNs). Fig. 2 shows a representative CNN, LeNet-5 [25]. LeNet-5 consists of two convolutional layers (C1 and C3) to identify certain characteristics of input feature maps (e.g., 6 feature maps of size 28 × 28 in C1) by applying local filters, two pooling layers (S2 and S4) to downscale the feature maps by performing maximum or average subsampling operations on a window (e.g., 28 × 28), and three classifier layers (F5, F6, and F7) to carry out classification according to features extracted from previous layers. Note that LeNet-5 does not include normalization layers, which have been proposed recently.[2] Although in LeNet-5, the number of synapses is about 50 kilos, this number has further increased to 10 billions recently [17], which makes the state-of-the-art NNs notoriously computationally and memory intensive.

2) Sparse Neural Networks: As the large number of neurons and synapses hinder efficient NN processing, researchers proposed a number of training techniques, such as sparse coding [29], auto encoder/decoder [19], [20], and deep belief network [21], to prune redundant synapses and neurons without loss of accuracy. A state-of-the-art pruning technique is proposed by Han *et al.* [23] in 2015. The pruning approach consists of three main steps, i.e., training connectivity, pruning connection (i.e., synapse), and training weight. In the first step, the NN is trained by traditional back propagation algorithm with normal learning rate. Then, connections with weights below a predefined threshold value can be removed. Finally, the pruned network should be retrained with a very small learning rate to obtain the final weights. To achieve a high compression ratio, the above process will be repeated until no synapse can be pruned. As reported in Table I, by using the proposed pruned technique, the average *sparsity* (i.e., the fraction of remaining synapses over the total number of synapses after pruning) is 12% across representative NNs without loss of accuracy.

[1]In this paper, we do not care which techniques are used for pruning the NN.

[2]Two typical types of normalization layers are local contrast normalization layer and local response normalization layer, which were proposed in 2009 [28] and 2012 [15], respectively.

TABLE I
NUMBERS OF NEURONS AND SYNAPSES IN SPARSE NNS (C FOR CONVOLUTIONAL LAYER; F FOR CLASSIFIER LAYER)

NN	Non-sparse # Neurons	Non-sparse # Synapses	Sparse # Synapses	Sparse Sparsity (%)	Sparse # Synapses in C.	Sparse # Synapses in F.
LeNet-5	8.90K	430.62K	36.30K	8.43	3.33K	32.97K
AlexNet	1.28M	60.95M	6.80M	11.15	864.86K	5.93M
VGG16	14.53M	138.34M	10.53M	7.61	4.81M	5.72M

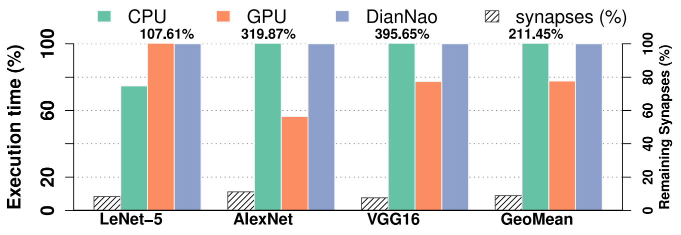


Fig. 3. Speedup of sparse NN versus dense NN on CPU, GPU, and DianNao.

B. Motivation

Though the number of operations (e.g., floating-point operations, flops or fixed-point operations, ops) and memory accesses can be greatly reduced with synapse pruning, existing hardware platforms (including CPUs, GPUs, FPGAs, and custom accelerators) cannot benefit a lot in terms of performance and energy efficiency, due to the lack of dedicated hardware support for irregular and sparse NN models.

On general-purpose platforms such as CPUs and GPUs, the performance gains of sparse NNs are relatively marginal compared to the amount of reduced operations. Fig. 3 shows the performance benefits (in terms of the reduction of execution time) of the sparse networks over the original dense versions on the CPU (Sparse BLAS versus Caffe [30]) and GPU (cuSPARSE versus Caffe) platform. We also present the sparsity of evaluated networks (i.e., LeNet-5 [25], AlexNet [15], and VGG16 [26]). For the CPU platform, except for LeNet-5, the performance of sparse networks is even worse than that of their dense versions, and the average slowdown is 211.45%. For the GPU platform, compared with the average sparsity as 9.06% (i.e., 90.94% synapses have been removed) on evaluated networks, the average performance gain is only 23.34%. Although researchers have proposed optimized approaches to leverage high parallelism of modern CPUs and GPUs for sparse representations, the attainable performance is still far from peak (e.g., only 0.49% for sparse matrix vector multiply on the GPU [31]), because of the inherent bandwidth limitation and noncomputational overhead [31]. The bandwidth problem also limits FPGAs to be in favor of sparse NNs.

The benefit of synapse pruning is even trivial on state-of-the-art NN accelerators, such as DianNao and DaDianNao. Since such custom accelerators cannot directly process sparse formats, sparse NNs have to be mapped to the accelerator the same way as dense NNs by filling the locations of pruned synapses with zero values. In this case, both the number of operations and memory accesses cannot be reduced at all for sparse NNs, yielding no reduction of execution time, as shown in Fig. 3. An intuitive optimization is to directly integrate sparsity encoding/decoding modules into existing DianNao or DaDianNao architecture, so as to reduce off-chip memory accesses as well. However, this solution may not be efficient for two main reasons: 1) the number of total operations remain the same, because the pruned synapses are still filled with zero values, incurring significant waste of computational resource and 2) neither the centralized architecture (e.g., DianNao) nor the symmetric tiled architecture (e.g., DaDianNao) can adapt to the irregularity of sparse NNs.

Therefore, the above observation motivates building a highly efficient architecture to take advantage of the irregularity and sparsity of modern NNs.

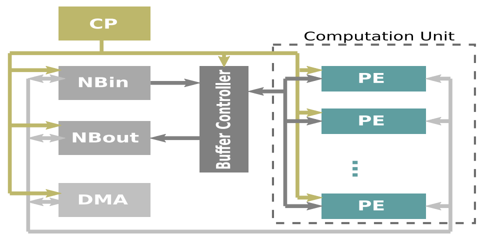


Fig. 4. Accelerator architecture.

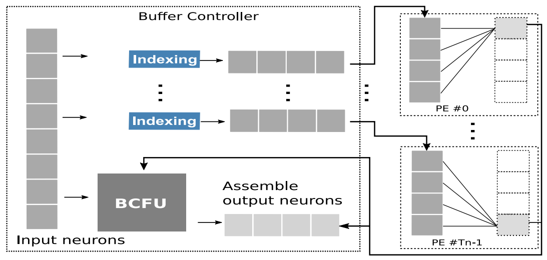


Fig. 5. BC architecture.

III. ACCELERATOR DESIGN

A. Overview

Fig. 4 presents the proposed architecture of our accelerator, which consists of a control processor (CP), a BC, two neural buffers (NBin and NBout), a direct memory access (DMA) module and a computation unit (CU) which contains multiple PEs, say T_n. All the PEs are connected in a topology of Fat-tree in order to avoid wiring congestion. The BC selects needed neurons for each PE from local neuron buffers based on the loaded instructions which are decoded by the CP, and transfers those neurons to PEs for efficient local computation. The logic connection between the BC and multiple PEs is shown in Fig. 5. A key feature of the proposed architecture is the indexing units in the BC. There are T_n indexing units in total, each corresponds to one PE, for selecting its necessary neurons.

B. Computation Unit

The CU is designed for efficient computation of the core operation of NNs, i.e., the vector multiplication–addition operation, with multiple PEs. Fig. 6(a) shows the architecture of the PE, consisting of a synapse buffer (SB) and NN functional units for the PE (PEFU). The PEFUs take synapses from the local SB and neurons from the BC as inputs, producing output neurons that will be sent back to the BC.

1) PEFU: The PEFUs are mainly used for multiplication–addition operations in NNs. A single PEFU consists of several multipliers, say T_m, as well as T_m-in adder-tree, see Fig. 6(b) for detailed architecture of the PEFU. Thus, T_n vector multiplication–addition operations ($T_m \bullet T_m$) can be performed at the same time with T_n PEs. In order to achieve high

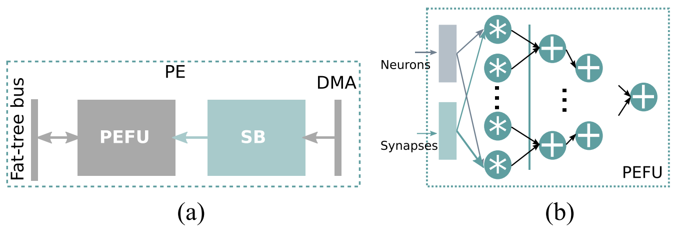
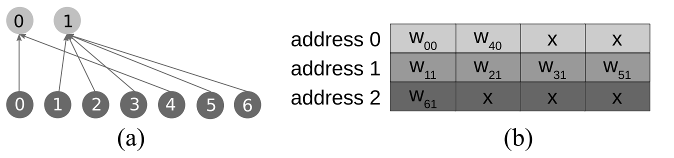


Fig. 6. Architecture of the (a) PE and (b) PEFU.



address 0	W_{00}	W_{40}	x	x
address 1	W_{11}	W_{21}	W_{31}	W_{51}
address 2	W_{61}	x	x	x

Fig. 7. (a) Example of sparse connection. (b) Data organization in the SB.

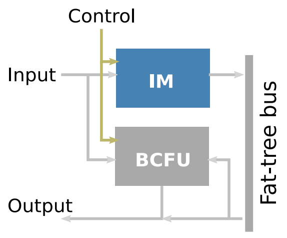


Fig. 8. Architecture of the BC.

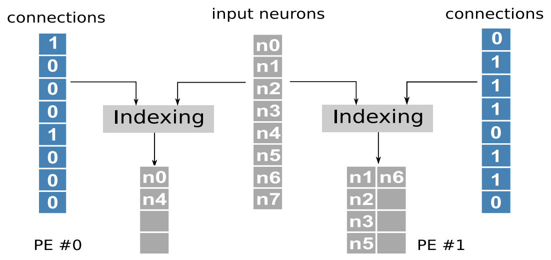


Fig. 9. Functionality of IM module.

frequency, we pipeline the functional units in the PEFU into two stages: 1) the multiplication and 2) the addition of all the multiplication results. With T_m inputs, all T_n PEs can produce T_n output neurons simultaneously.

2) SB: The SB is used for storing distributed synapses, and there are two key issues during the design of the SB. The first is to determine an appropriate size of the SB, and the second is to organize synapses in the SB.

Though the previous work proposed to offer large enough buffers for holding all synapses of NNs with moderate sizes [12], [13], so as to avoid costly off-chip memory accesses, the SB in our accelerator is not designed for holding all synapses. The reason is twofold. First, even with sparsity, the total size of synapses is more than several megabytes, e.g., ~7 MB for AlexNet and ~10 M for VGG as listed in Table I. Second, our accelerator is designed for supporting NNs with different sparsity levels, including dense networks that have much large sizes of synapses. Thus, designing a large SB for storing all synapses would incur considerable delay, area, and energy penalty. Actually, the SB with optimal size should be able to hide the latency of memory accesses, in order to keep the PEFU busy without waiting for input data. In our current implementation, we deploy 2 KB SB in each PE, leading to $2 \times T_n$ KB storage in total for synapses. Thus, each SB can offer T_m data to the PEFU every cycle, i.e., a $T_m \times 16$-bit width SRAM is provided.

To illustrate the synapse organization in the SB, we use a sparse network example consisting of seven input neurons and two output neurons, as shown in Fig. 7(a). The weights of synapses connected to different output neurons are stored in the SB compactly by aligning to T_m (i.e., 4). Hence, when computing output neuron 0, the SB only needs to be read once but twice for output neuron 1. As the number of synapses of different neurons may significantly differ from each other, we allow SBs in different PEs to load new data from the memory asynchronously to improve overall efficiency.

C. Buffer Controller

The BC is designed for transferring necessary neurons to PEs, orchestrating computations on PEs, and performing less computation-intensive operations. Fig. 8 illustrates the architecture of the BC, which consists of a module used to index data for computation based on connections (IM), and the specialized function units for the BC (BCFU).

1) BCFU: The BCFU is mainly used for storing neurons to be selected by IM. Note that there are T_m such units, thus it can store T_m neurons simultaneously.

2) IM: The IM is the key component of our accelerator, and it is used for indexing needed neurons of sparse NNs with different levels of sparsity. Instead of distributing an IM to each PE, we design a centralized IM in the BC and only transfer the indexed neurons to PEs, which can significantly reduce the bandwidth requirement between the neural buffer and PEs because the number of data after indexing is much smaller in sparse networks. In Fig. 9, different input neurons are selected for different PEs based on stored connections. For PE #0, only two neurons, i.e., n_0 and n_4, are selected from all eight neurons for computation on PEs.

To implement the IM, we investigate two commonly used indexing options, i.e., *direct indexing* and *step indexing*. The direct indexing approach uses a binary string with one bit per synapse, indicating whether the corresponding synapse exists, i.e., "1" for existence and "0" for absence. The step indexing approach further indexes the binary string of direct indexing by using distances between existed synapses (1s in the binary string), i.e., each element in the index table indicates the distance between two existed synapses.

Although there exists other indexing methods, such as compressed sparse row (CSR), coordinate list, and compressed sparse column (CSC), direct indexing and step indexing are relatively easy to implement from the perspective of hardware design. For example, well used CSR/CSC need two arrays to store indexes for sparse matrix which will be costly for storage in the context of sparse NNs whose sparsity are usually larger than 5% (see Table I). Besides, CSR/CSR are indexing row and column of matrix while our deliberated design scheduling in accelerator is indexing multiple neurons and synapses 1-D in parallel. Thus, we investigate *direct indexing* and *step indexing* for implementing high efficient IM.

In direct indexing, neurons are selected from all input neurons directly based on existed connections (i.e., 1s) in the binary string. The binary string of a sparse network example

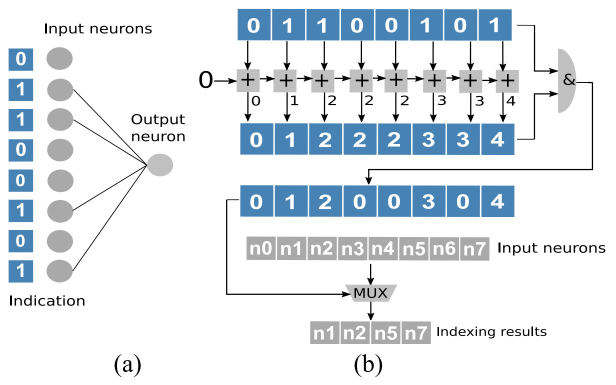


Fig. 10. (a) Sparse network example with the direct indexing. (b) Hardware implementation of direct indexing.

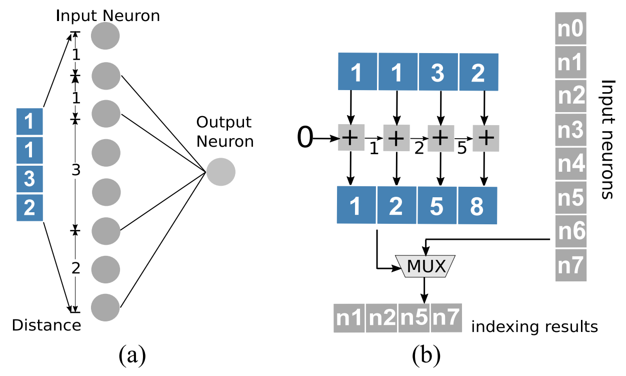


Fig. 11. (a) Sparse network example with the step indexing. (b) Hardware implementation of the step indexing.

is shown in Fig. 10(a). We also present the potential hardware implementation in Fig. 10(b). In step indexing, neurons are selected based on the distances between input neurons with existed synapses. We present the same network example with step indexing in Fig. 11(a) and the potential hardware implementation in Fig. 11(b).

We implement the above two indexing approaches in RTL and compare corresponding hardware costs in terms of area and power with synthesized results in Fig. 12. Note that indexes are computed in parallel for both implementations. By selecting 16 data from an array with a length varying from 32 to 512 (i.e., sparsity varying from 50% to 3.12%) in one cycle, we observe that the costs are increasing with the sparsity. Moreover, the costs of the step indexing are always smaller than that of the direct indexing on all evaluated datasets. For example, when the size of data array is 256 (this size is also used in our current implementation), the area and power of step indexing are about 10% and 40%, respectively, less than that of the direct indexing.

Based on the above investigation, we select and apply the step indexing to implementing IM. In the current design, IM is able to read $T_m \times T_m$ data every cycle for selecting input neurons of each PE.

D. CP

The CP is designed for efficiently and flexibly controlling the execution with various instructions. The instructions are used for data organization, execution coordination, memory accesses, etc., and they are stored in a small instruction buffer.

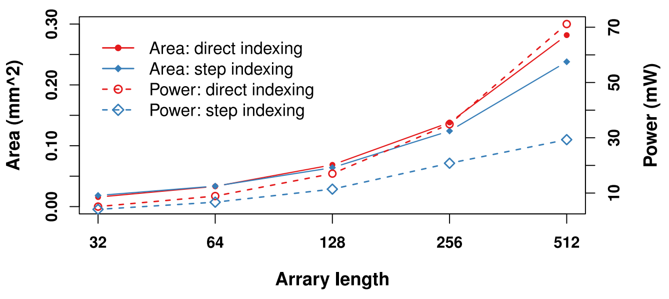


Fig. 12. Area and power costs of selecting from different length of array at a time.

To ease the programming burden of end users, we provide a compiler in C++ to generate highly efficient instructions, which will be elaborated later.

E. NB

The NB includes NBin and NBout, for storing input and output neurons, respectively: the input neurons are selected from NBin then sent to all the PEs for computation, and the output neurons are collected to NBout after computation. The neurons stored in the NBs are arranged orderly disregarding various connection patterns of the sparse networks. In the current implementation, we set the width of the data bus between the IM and NBin as $T_m \times T_m \times 16$ bits. Thus, at most T_m data can be selected out for each PE per cycle.

The sizes of NBin and NBout are decisive to the overall performance and energy consumption. After studying different sizes of NBs, we found that 8 KB is the optimal tradeoff between the achieved performance and related energy consumption. Therefore, in our accelerator implementation, we use 8 KB for both NBin and NBout. Note that, the size of NBin is larger than that of SB (for storing synapses, 2 KB) since the accelerator needs to store more input neurons before selecting necessary neurons for PEs.

Apparently, 8 KB NBs cannot hold all neurons of large-scale NNs, and thus a proper data replacement strategy should be employed to reduce costly off-chip memory accesses. Only when all the neurons in NBin have been processed or NBout is full, the main memory will be accessed for loading new input neurons or storing computed output neurons, respectively.

IV. Programming Model

In this section, we introduce the programming model proposed for our accelerator.

A. Library-Based Programming

To ease the programming burden, we propose a library-based programming model for our accelerator. The basic idea is to provide a set of high-level (e.g., C/C++ level) library functions, each corresponding to a basic NN operation, so that users can invoke our accelerator directly with high-level languages. Listing 1 shows the function declaration of the convolution operation in our library. In addition to NN operations, we also provide relatively low-level primitives, such as matrix/vector multiply/add. Thus, users can leverage such primitives, together with typical language constructs (e.g., loop and condition) to implement more complicated operations. Eventually, the original C/C++ code will be compiled and

```
ConvolutionForward(
  TensorDescriptor_t inputDesc,  // input descriptor
  void* input,                   // input data
  TensorDescriptor_t filterDesc, // filter descriptor
  void* filter,                  // filter
  TensorDescriptpr_t outputDesc, // output descriptor
  void* output);                 // output data
```

Listing 1. Function declaration of the convolution operation.

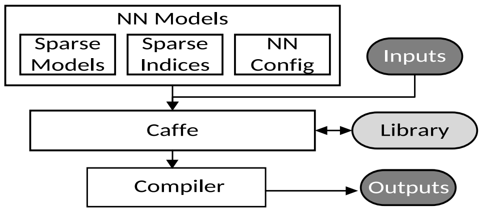


Fig. 13. Programming process of our accelerator.

optimized by our in-house compiler, so as to generate highly efficient binary instructions.

B. Programming Framework

To gain performance portability, we also integrate the implemented programming library into widely used deep learning frameworks such as Caffe [30]. Therefore, end users can directly leverage the interface of Caffe (i.e., network configuration file) without any modification of their codes. Fig. 13 shows the programming process of our accelerator. Initially, we use the sparse NN model obtained from the training phase, and the corresponding sparse representation to create a compact NN model file. Then, the compact model file, the NN configuration, and the input data, are sent to Caffe. In Caffe, our library functions are invoked to our compiler to generate outputs. For the dense NN, the model file only contains the dense NN model gained from the training phase. In this case, the underlying network format is entirely transparent to the user.

C. Compiling Process

A huge gap exists between high-level deep learning frameworks such as Caffe, and our accelerator. The major challenge of compiling from high-level code to the accelerator is related to the on-chip data management. Specifically, first, data allocations and movements are complicated as data are using various types and new features such as sparsity are introduced in our accelerator; therefore, appropriate strategies for different data are needed for high efficiency. Second, the limited size of on-chip memory brings in complicated scheduling. Considering the contradiction of large amount of neurons/synapses and limited on-chip memory size, techniques such as loop tiling are inevitably. Particularly, different loop tiling strategies will affect the data reuse, memory access, scheduling, and thus the hardware efficiency. We need to explore different loop tiling strategies and choose the one with the maximal data reuse during compiling.

We design a sophisticated compiler to bridge the gap and we show the process of compiling a source file written by our library into executable code for Cambricon-X in Fig. 14. The compiling process contains two primary parts in general, i.e., data placement and instructions generation. In data placement,

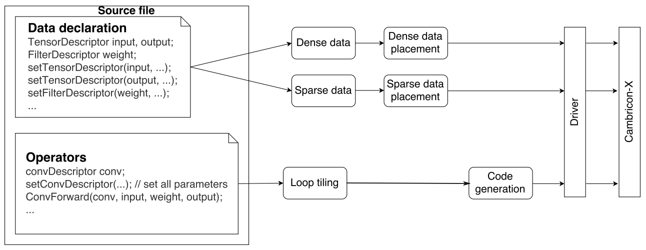

Source file
Data declaration
TensorDescriptor input, output;
FilterDescriptor weight;
setTensorDescriptor(input, ...);
setTensorDescriptor(output, ...);
setFilterDescriptor(weight, ...);
...
Operators
convDescriptor conv;
setConvDescriptor(...); // set all parameters
ConvForward(conv, input, weight, output);
...
Dense data
Dense data placement
Sparse data
Sparse data placement
Loop tiling
Code generation
Driver
Cambricon-X

Fig. 14. Compiling process.

data structures in source codes are allocated and transferred to the accelerator. We use *tensor* to encapsulate input and output neurons, and *filter* to encapsulate weights of convolutional and fully connected layers, both of which serve as 4-D tensors. We use separate data types to represent these data as they will be loaded into different on-chip memories (i.e., NBin/Nout and SBs) while executing. Sparsity is specified as an attribute of enum type passed to the filter descriptor. In instructions generation, different operators are managed with different data to be executed on accelerator, mainly the accelerator specified operations, such as convolutional operations. Then we apply loop tiling to decompose the operator into suboperators to make sure that the data (i.e., inputs, weights, and outputs) associated with the suboperator can be filled into the limited on-chip memory. Meanwhile, data movement operators are generated and integrated with the computational operators. Finally, the code generator is followed to transform the operators into the specific instructions that can be decoded and executed by Cambricon-X.

D. Loop Tiling and Optimizations

Loop tiling is to reasonably partition the computations so that data required by each piece of computation could fit in the on-chip memory. Different tiling strategies will affect executing behaviors such as data reuse and scheduling, leading to different efficiency. As memory accesses are costly in Cambricon-X (see Section VI-C), the priority of our tiling strategy is to improve the on-chip data reuse, thus reducing memory accesses and improving hardware energy efficiency. As reported in Eyeriss [32], different data reuse strategies can be applied for energy efficiency, including convolutional reuse, filter reuse, input feature map reuse, and partial sums reuse. Inline with Eyeriss, we classify the data reuse strategies into three types: 1) output-reuse (i.e., partial sums reuse); 2) synapse-reuse (i.e., convolutional reuse and filter reuse); and 3) input-reuse (i.e., input feature map reuse), which indicate to maximize the reuse of output data, synapse data, and input data, respectively. For output-reuse strategy, a segment of input neurons are loaded to the NBin buffer and a segment of synapses are loaded to the SB buffer, to compute a segment of output neurons, which are stored in NBout buffer. Then the second segment of input neurons and synapses will be loaded to compute the second partial sum. After input segment times of replacement, a segment of output data is accomplished and will be stored back to the main memory. As to the synapse-reuse strategy, input data and output data will be repeatedly accessed. And for input-reuse strategy, synapse and output data will be repeatedly accessed. While selecting the reuse strategy for a convolutional layer, we calculate the workloads reduced by the three reuse strategies, respectively, and use the one that maximizes the data reuse.

TABLE II
BENCHMARKS WITH REMAINING SYNAPSES IN EACH LAYER (C FOR CONVOLUTIONAL LAYERS; F FOR CLASSIFIER LAYERS; TOTAL FOR ALL LAYERS)

—	LeNet-5		AlexNet		VGG16	
	—Synapses	Sparsity	Synapses	Sparsity	Synapses	Sparsity
Total	—36.30K	8.43%	6.80M	11.15%	10.53M	7.61%
C	—3.33K	13.06%	864.86K	37.08%	4.81M	32.69%
F	—32.97K	8.14%	5.93M	10.12%	5.72M	4.63%
—	Dropout NN1		Dropout NN2		Cifar10	
	—Synapses	Sparsity	Synapses	Sparsity	Synapses	Sparsity
Total	—44.38K	6.99%	5.89M	8.00%	6.15K	5.02%
C	—-	-	-	-	4.62K	5.84%
F	—44.38K	6.99%	5.89M	8.00%	1.53K	4.07%

Without loss of generality, we use a convolutional layer (*conv4_2*) extracted from existing network VGG16 to clearly show the date reuse strategy selection. For any given segment number (*SegN*) and segment size (*SegSize*), the estimated memory access workloads for inputs, outputs, and synapses can be computed with simple formulas. Precisely, *conv4_2* has input size of $30 \times 30 \times 512$ (with padding zero in height and width dimensions), output size of $28 \times 28 \times 512$, and synapse size of $3 \times 3 \times 512 \times 512$. Hence, for Cambricon-X implemented in this paper with parameters shown in Section V, the channel and height dimensions of the output data is sliced into 4×128 (SegN $\times$ *SegSize*) and 28×1, respectively. The channel of input data is sliced into 16×32. The segment size of input, output, and synapse are $SegSize_{\text{in}} = 30 \times 3 \times 32$, $SegSize_{\text{out}} = 28 \times 1 \times 128$, and $SegSize_{\text{syn}} = 3 \times 3 \times 128 \times 32 \times 0.27$. Note that the synapse segment size is multiplied by the sparsity (0.27) to estimate the workloads. Thus, the estimated memory access workloads for input, output, and synapse reuse strategy are 60.98 MB ($16 \times 1 \times 28 \times (SegSize_{\text{in}} + SegSize_{\text{syn}} \times 4 + SegSize_{\text{out}} \times 4 \times 2)$), 44.63 MB ($4 \times 28 \times 1 \times (SegSize_{\text{out}} + SegSize_{\text{in}} \times 16 + SegSize_{\text{syn}} \times 16)$), and 35.56 MB ($16 \times 4 \times (SegSize_{\text{syn}} + SegSize_{\text{in}} \times 28 \times 1 + SegSize_{\text{out}} \times 28 \times 2)$), respectively. Therefore, we select the synapse-reuse strategy for *conv4_2*.

V. EXPERIMENTAL METHODOLOGY

In this section, we introduce the experimental methodology.

A. Benchmarks

We use six representative NNs, i.e., LeNet-5 [25], AlexNet [15], VGG16 [26], Dropout NN1 [18] (2-layer MLP, 800 hidden neurons), Dropout NN2 [18] (3-layer MLP, 8192×8192 hidden neurons), and Cifar10 quick model [33] as our benchmarks. Table II lists the characteristics of those networks, including the number of synapses, and the corresponding sparsity of different kinds of layers, so as to demonstrate the flexibility of our accelerator.

B. Measurements

We implement our accelerator with RTL description in Verilog, synthesize it with Synopsys Design Compiler, then place and route it with Synopsys IC Compiler using the TSMC 65 nm Gplus High VT library. We use CACTI 6.0 to estimate the energy consumption of DRAM accesses [34].

TABLE III
HARDWARE PARAMETERS OF ACCELERATORS

	Cambricon-X	DianNao
# BC	1	-
# PE	16	-
# multiplier	256	256
# 16-in adder tree	16	16
# ALU	16	16

C. Baselines

We compare our design with three baselines, i.e., the CPU, the GPU, and DianNao.

1) *CPU:* We use Caffe [30], the most popular deep learning framework, to evaluate our benchmarks on a modern CPU, Intel Xeon CPU E5-2620 v2 (denoted as CPU-Caffe). Also, in order to adapt to sparse NNs, we implement the evaluated benchmarks with the most widely used sparse library, i.e., sparseBLAS [27] (CPU-Sparse), on the CPU.
2) *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 28 nm 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 on the GPU with state-of-the-art cuSparse library [24] (GPU-cuSparse).
3) *Accelerator:* We also compare our accelerator against the state-of-the-art NN accelerator, DianNao [11]. With the help from the authors of DianNao, we reimplement DianNao with the same technology process as well as other details in their paper, in order to have a fair comparison. More specifically, we implement DianNao with 16×16 multipliers, 16 16-in adder trees and 16 nonlinear modules. It is notable that we apply the optimal loop tiling and data reuse strategy for both DianNao and Cambricon-X for the optimal performance gain and energy efficiency.

VI. EXPERIMENTAL RESULTS

A. Hardware Characteristics

In the current implementation, we select $T_m = T_n = 16$ as discussed in Section III-B, thus the accelerator consists of 16 PEs, each of which has 16 multipliers and one 16-in adder tree. We report the characteristics of our accelerator as well as our reimplemented DianNao in Table III. Note that the ALU in Table III refers to the modules used in the last stage in DianNao for nonlinear functions, achieving the same functionality as CTFU. With such design, our accelerator is able to achieve the peak performance as DianNao, i.e., 528 fixed-point operations every cycle.

We present in Table IV the layout characteristics (including area and power consumption) of our accelerator. The accelerator, which has 56 KB on-chip SRAM and 528 operators in total, is 2.11× larger than DianNao with 6.38 mm^2 versus 3.02 mm^2. The total power of our accelerator is only 954 mW, which is 469 mW higher than DianNao with 485 mW. Additionally, we achieve a frequency of 1 GHz which is a little bit higher than 0.98 GHz in DianNao.

TABLE IV
HARDWARE CHARACTERISTICS OF ACCELERATORS

accelerator	Area (mm^2)	%	Power (mW)	%
Total	6.38	100	954	100
BC				
NBin	0.55	8.66	93.32	9.78
NBout	0.55	8.66	93.32	9.78
CTFU	0.11	1.72	31.63	13.31
IM	1.98	31.07	332.62	34.83
CP	0.16	2.54	75.06	7.86
PEs				
LTFU	1.78	27.94	153.01	16.02
SB	1.05	16.51	151.91	15.91

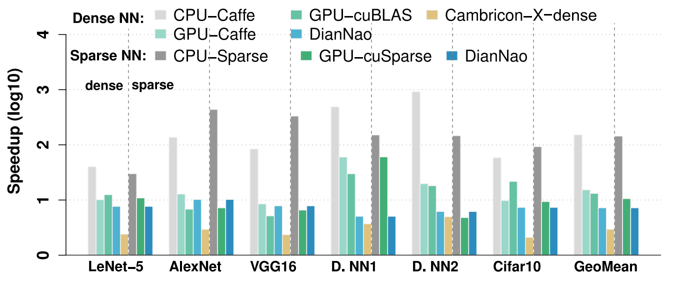


Fig. 15. Speedup of our accelerator over CPU, GPU, and DianNao for all evaluated NNs, including both the dense and sparse implementations. All the performance numbers of different implementations are normalized to that of our accelerator with sparse networks.

B. Performance

We compare our accelerator against CPU, GPU, and DianNao on all evaluated networks listed in Table II with different implementations. On the CPU and the GPU, in addition to implementations with dense libraries for dense representation (i.e., CPU-Caffe, GPU-Caffe, and GPU-cuBLAS), we also implement evaluated networks with sparse libraries for sparse representation (i.e., CPU-Sparse and GPU-cuSparse). For fair comparison, we evaluate the performance of our accelerator for dense representation (i.e., Cambricon-X-dense) as well. In Fig. 15, we normalize all the performance numbers of the above implementations to that of our accelerator for sparse representation. Regarding implementations for dense representation, on average, our accelerator is 51.55×, 5.20×, and 4.94× faster than CPU-Caffe, GPU-Caffe, and GPU-cuBLAS, respectively, on the evaluated benchmarks (our accelerator for sparse representation achieves 151.82×, 15.32×, and 13.18×, respectively). Regarding the sparse representation, on average, our accelerator is 144.41× and 10.60× faster than CPU-Sparse and GPU-Sparse, respectively. Compared against DianNao, our accelerator still achieves 7.23× speedup, which well demonstrates the efficiency of our accelerator. Note that our accelerator can efficiently process not only the sparse networks but also the dense networks, as demonstrated by the observation that Cambricon-X-dense achieves 2.46× speedup over DianNao.

To gain more insights of the above performance benefit, we further show the performance comparison of the convolutional layers and classifier layers, in Figs. 16 and 17, respectively, where all the performance numbers are normalized to that of our accelerator with sparse representation. In Fig. 16, for the convolutional layer, on average, our accelerator is 8.90×, 7.67×, and 8.89× faster than GPU-cuBLAS, GPU-cuSparse, and DianNao, respectively. In Fig. 17, for

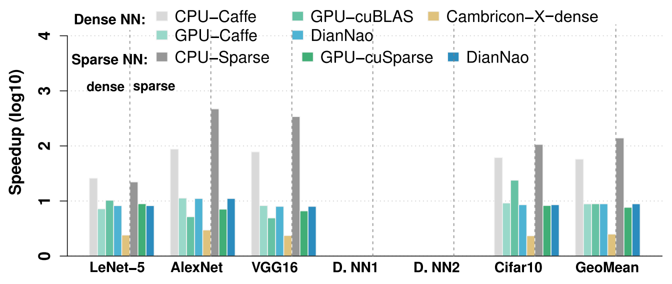


Fig. 16. Speedup of our accelerator over CPU, GPU, and DianNao for convolutional layers. Note that there is no convolutional layer for Dropout NN1 and Dropout NN2.

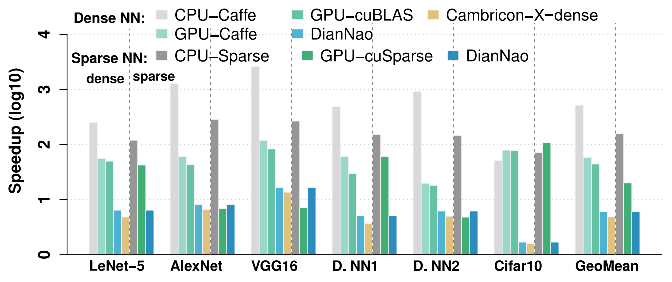


Fig. 17. Speedup of our accelerator over CPU, GPU, and DianNao for classifier layers.

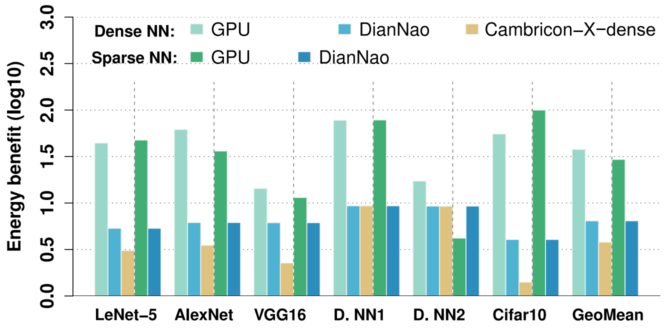


Fig. 18. Energy benefit of our accelerator over GPU (GPU-Caffe and GPU-sparse for the dense and sparse representation, respectively) and DianNao, where all the results are normalized to that of our accelerator with sparse networks.

the classifier layer, on average, our accelerator is 44.28×, 20.07×, and 5.99× faster than GPU-cuBLAS, GPU-cuSparse, and DianNao, respectively. Generally, the speedup on the classifier layers is much larger than that on the convolutional layers, because the classifier layers can achieve higher sparsity than the convolutional layer (5.23% versus 22.65%) on the evaluated benchmarks. This is also validated by the observation that our accelerator achieves 2.51× and 4.84× speedup over Cambricon-X-dense for the convolutional and classifier layers, respectively.

C. Energy

In Fig. 18, we report the energy comparison of the GPU, DianNao, and our accelerator across all the benchmarks, where the energy of off-chip memory access is also included. Compared to the GPU platform, on average, our accelerator achieves 37.79× and 29.43× better energy efficiency for the dense and sparse networks, respectively. Compared to DianNao, on average, our accelerator achieves 6.43× better

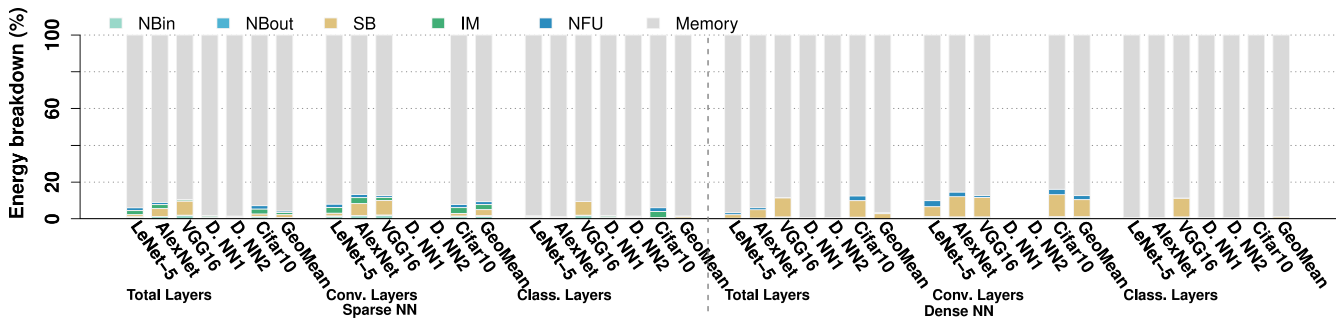


Fig. 19. Energy breakdown with memory accesses.

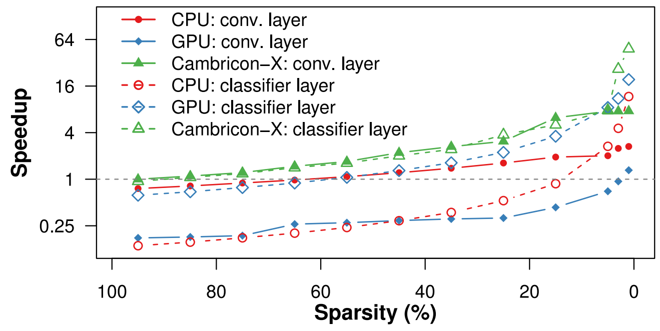


Fig. 20. Speedup of sparse layer over dense layer.

energy efficiency for both the dense and sparse networks. We made an interesting observation that the best energy efficiency achieved by our accelerator over both the GPU and DianNao is from *AlexNet*. The main reason is that kernel sizes in convolutional layers of AlexNet are larger than that of other networks, which elevate the memory efficiency drastically. Moreover, our accelerator for dense representation reduces energy by 1.70× over DianNao, which demonstrates that our accelerator is also energy efficient for processing dense networks.

We further show the energy breakdown of our accelerator for all layers, the convolutional layers, and the classifier layers in Fig. 19, where the results for both the dense and sparse networks are also presented. We can see that the main memory accesses consume more than 80% of the total energy across all layers, which is consistent with the results reported by Chen *et al.* [11]. It is clear that the ratio of memory access energy of the classifier layers is much higher than that of the convolutional layers (i.e., 98.39% versus 90.63% on average) due to the high sparsity in the classifier layers. Also, by comparing breakdown results of the sparse and dense networks, we can observe that, for most networks, the ratios of memory access energy of sparse networks are generally higher than that of the dense networks (e.g., 90.63% versus 87.28% on the convolutional layers averagely). In other words, the energy problem of off-chip memory access is more severe for the sparse networks due to their relatively low computational intensity compared with the dense networks.

D. Sensitivity to Network Sparsity

We investigate the sensitivity of hardware platforms to different levels of network sparsity. Fig. 20 shows the speedup of sparse network layers (i.e., the convolutional layer and the classifier layer) with varying sparsity (ranging from 1% to 95%) over the dense ones on CPU, GPU, and our accelerator.[3] We made several interesting observations.

1) When the sparsity is relatively high (e.g., more than 70%), both the CPU and GPU platforms cannot benefit from sparse networks due to nontrivial cost of sparse data processing. For example, the performance of the classifier layer with 85% sparsity is even 85% and 31% worse than that of the dense layer on the CPU and GPU, respectively.
2) On the GPU platform, the classifier layer and the convolutional layer have significantly different behaviors. More specifically, for the classifier layer, when the sparsity is less than 55%, the sparse network can outperform the dense version, and the speedup can be continuously improved with decreasing sparsity. While for the convolutional layer, the sparse network can only outperform the dense version when the sparsity is less than 3%. The underlying reason is that the speedup of sparse matrix–vector multiplication (SpMV) employed by the classifier layer is much more sensitivity to the decreasing sparsity than that of sparse matrix–matrix multiplication employed by the convolutional layer.
3) On the CPU platform, one observation is completely different from that of the GPU platform, that is, the speedup of the convolutional layer is generally better than that of the classifier layer. Nevertheless, the speedup of the classifier layer is improved much more drastically than that of the convolutional layer. This observation complies with that of the GPU platform, that is, the relatively bandwidth-limit SpMV is much more sensitive to the decreasing network sparsity.
4) On our accelerator, the sparse networks can easily outperform the dense versions even when the sparsity is only 95%, and the performance gain can be improved greatly with the decreasing sparsity. For example, given the sparsity as 1%, our accelerator can achieve 48.53× speedup while the GPU and CPU can only achieve 19.42× and 11.72× speedup, respectively, on the classifier layer. In addition, for both the convolutional layer and classifier layer, the sparse networks have consistent speedup over the dense versions.

The above observations further validate that our accelerator can well exploit the sparsity of modern NN models. Also, it is clear that our accelerator are more adaptive to future NNs with far lower sparsity, i.e., <1%, compared against to existing hardware platforms.

[3]We do not present the results of DianNao, since the speedup is always 1 for networks with different sparsity.

TABLE V
ACTIVATION SPARSITY AVERAGING ON ENTIRE DATASET

	Conv.	FC	Total
LeNet-5 (Sigmoid)	100%	88.50%	99.99%
LeNet-5 (ReLU)	51.52%	61.46%	51.80%
AlexNet	62.37%	60.73%	62.30%
VGG16	40.52%	56.97%	40.53%
Dropout NN1	–	33.69%	33.69%
Dropout NN2	–	66.10%	66.10%
Cifar10	69.39%	80.07%	69.43%
GeoMean	54.82%	57.96%	52.20%

VII. Cambricon-X Optimization

In this section, we further improve the efficiency of Cambricon-X by investigating two possibilities, i.e., activation sparsity and multi-issue controller.

A. Activation Sparsity

Activation sparsity occurs when the output of a neuron (after activation function) is zero. Especially when using ReLU [35] as the activation function, neurons having negative results before ReLU will have output value of zero. Formally, neurons with ReLU compute the output with

$$y = \text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{1}$$

where x is the input for ReLU activation function and y is the output of ReLU. Therefore, activation functions such as ReLU can introduce many zeros in the networks. Different from neuron sparsity and synapse sparsity illustrated in Section II, activation sparsity does not change the network topology but varies with inputs. It is tightly related to inputs as different inputs will lead to different computation results.

In Table V, we report the average activation sparsity of six NN benchmarks, including in the whole networks, convolutional layers, and the classifier layers. Note that the original LeNet-5 uses Sigmoid as the activation function, thus very limited activation sparsity. Despite that, it can be observed that roughly half of the neurons in these networks with ReLU have zero-valued outputs. The total activation sparsity varies from 33.69% for Dropout NN1, to up to 69.43% for Cifar10 quick model and the average across all networks is 52.20%. The average activation sparsity in the convolution layers is slightly lower than that in the classifier layers (54.82% versus 57.96%). Thus, ideally, leveraging the activation sparsity can further improve the efficiency of accelerator by 2×. We leverage the activation sparsity from two aspects. First, we design a neuron encoder/decoder module to encode/decode the neurons dynamically, thus reducing the costly DRAM memory accesses. Second, we design and optimize the PEs to disable unnecessary computations to reduce the processing energy.

1) Neuron Encoder/Decoder: Figs. 21 and 22 present the proposed architecture of neuron encoder and neuron decoder, respectively. The neuron encoder is placed near the NBout to encode the neurons by eliminating neurons with zero output. The location of nonzero neurons are indexed with directing indexing. The encoding process is as follows. Based on the neuron values, the neuron indexes are generated and each index indicates whether the corresponding neuron value is zero. Then we enforce the "AND" operation between the neuron indexes and the accumulated neuron indexes to produce

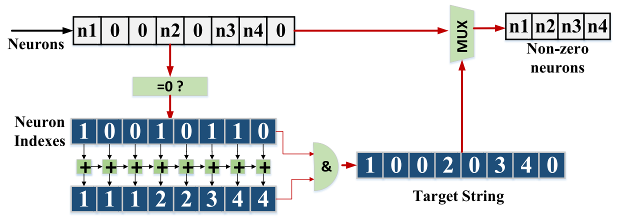


Fig. 21. Neuron encoder.

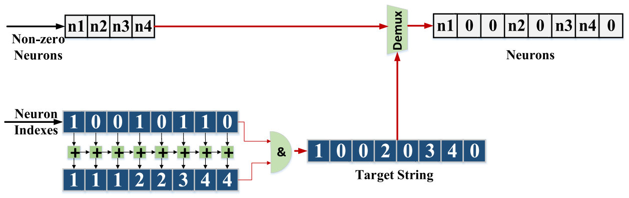


Fig. 22. Neuron decoder.

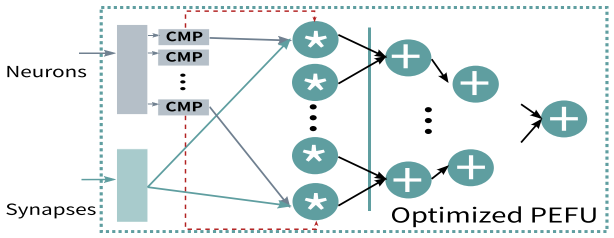


Fig. 23. Optimized PEFU.

the target string, which is used for selecting the nonzero neurons. Finally, the original neurons are encoded into nonzero neurons and the neuron indexes. The neuron decoder is placed near the NBin to decode the neurons. The decoding process is a reverse process of the encoding process, which decodes the neurons based on the nonzero neurons and the corresponding indexes. The accelerator loads the encoded input neurons from the DRAM, decodes them through the neuron decoder, and writes them to the NBin. The computed output neurons are read from the NBout, encoded through the neuron encoder, and stored to the DRAM. The process can greatly reduce the DRAM accesses, thus reducing DRAM access energy.

2) PE Optimization: In Fig. 23, we show the design of the optimized PEFU in PE. The optimized PEFU has the ability to judge whether each input neuron is zero or not. If the input neuron is zero, the computation can be skipped, thus the following operators including multiplier will be disabled. Note that for such optimization, the activation sparsity can only be utilized for better energy efficiency but not the performance improvement as we only gate the operators for unnecessary computations in accelerator.

3) Evaluation: In Fig. 24, we report the DRAM energy benefit of neuron encoder/decoder over the original design. We further breakdown the energy benefit for convolutional layers and classifier layers. The neuron encoder/decoder achieves 1.28× better energy efficiency on average. To be more specific, it achieves 1.68× and 1.02× better energy efficiency for convolutional layers and classifier layers, respectively. The DRAM energy benefit of classifier layers is very limited as the DRAM accesses for synapses dominates the classifier layers, thus very limited improvement when encoding/decoding neurons. In our current implement, the neuron encoder is

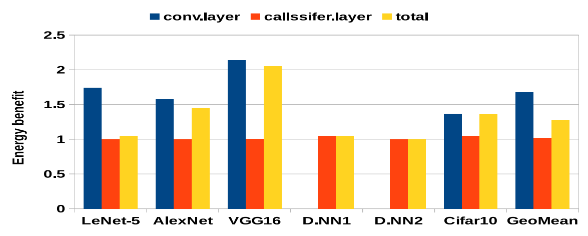


Fig. 24. DRAM energy benefit of the neuron encoder/decoder.

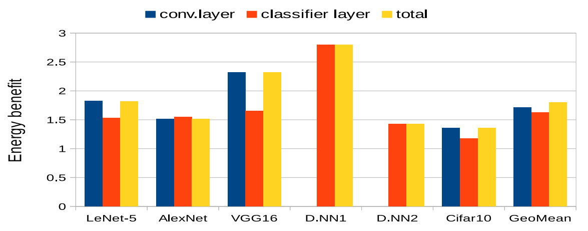


Fig. 25. Energy benefit of the optimized PEFU.

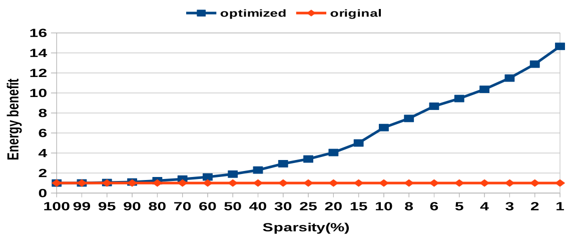


Fig. 26. Energy benefit of the optimized PEFU with varying sparsity.

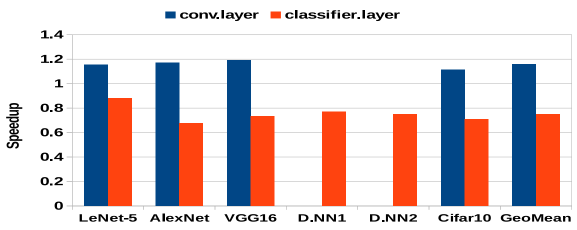


Fig. 27. Speedup of using multi-issue controller.

designed to select 16 out of 64 neurons as the activation sparsity is beyond 25% in most of NNs. Although the neuron encoder/decoder introduces additional 0.08 mm^2 area cost (1.25%), it achieves 1.28× better energy efficiency, which validates the efficiency of the neuron encoder/decoder.

As reported in Table V, roughly half of the multiplication operations in the PEFU can be skipped, which can greatly reduce the energy consumption of the PEFU. We show the energy benefit of the optimized PEFU over the original PEFU in Fig. 25. To gain more insight, we breakdown the energy benefit with the convolutional layers and classifier layers. With additional 0.03 mm^2 (0.49%) area cost, our optimized PEFU achieves 1.81× better energy efficiency (1.73× and 1.65× in convolutional layers and classifier layers, respectively). For large NNs, i.e., AlexNet and VGG, our optimized PEFU achieves 1.55× and 2.27× better energy efficiency, respectively. For small NNs, on average, our optimized PEFU can still achieve 1.78× better energy efficiency. The results demonstrate that the optimized PEFU is energy efficient for activation sparsity.

4) Scalability: Different from synapse sparsity that will change the network topology and affect the accuracy, leveraging activation sparsity through skipping the unnecessary computations does not change the network topology and accuracy. We further investigate the sensitivity of optimized PEFU to different levels of activation sparsity. Fig. 26 shows the energy efficiency of optimized PEFU with varying activation sparsity over the dense version. We observe that energy efficiency of optimized PEFU can easily outperforms the dense versions even when the activation sparsity is only 99%(1.01×). What is more, the energy efficiency can be improved greatly with the decreasing activation sparsity. The energy efficiency increases from 1.00× to 14.65× as the activation sparsity decreases from 100% to 1%. While the original PEFU cannot exploit the activation sparsity. The above observation further validate that our optimized PEFU can well exploit the activation sparsity for better energy efficiency.

B. Multi-Issue Cambricon-X

As Cambricon-X is equipped with an in-order control unit with issue width of one, it may ignore some potential savings for performance improvement. Therefore, when executing a layer of DNNs, the data movement instructions (*IO instructions*, i.e., *load/store*) and computation instructions are executed sequentially. Even Cambricon-X has been implemented with mechanic of double buffering where it can execute a computation instruction once sufficient data have been loaded, it has to wait the computation instruction be finished before it can issue another instruction. A simple but effective solution is to have the accelerator predecode next instruction and start to execute if it does not conflict with the previous instruction. Thus, we investigate the possibility of a multi-issue controller to leverage such potential for higher performance.

The key feature of the multi-issue controller is to issue several independent instructions in a cycle. In general, the data movement instructions and the computation instructions are independent and can be executed at the same time. The data movement instructions include the load instructions issued to the NBin/SB and the store instructions issued to the NBout. In order to hide the DMA behind the computation, we leverage the NBin, SB, and NBout in the double buffering manner. We search the independent instructions in the compiling process in a static way and issue them together when executing. As the multi-issue controller needs to fetch more instructions in a cycle, we optimize the instruction buffer with higher bandwidth to fetch several instructions in a cycle.

The performance gain of the multi-issue controller over the original controller is shown in Fig. 27, where we breakdown the performance gain in the convolutional layers and classifier layers. With additional 0.05 mm^2 area cost (0.78%), the multi-issue controller can further achieve 1.16× speedup on the convolutional layers, which comes from the computations overlap the memory accesses. However, for the classifier layers, the performance is even worse, and the average slowdown is 24.8%. This is because classifier layers are memory-intensive where the bandwidth is the bottle-neck for performance. While double buffering partitions the data in a more fine-grained way with half-size available on-chip storage, thus bringing in additional load/store instructions and longer latency, which overweights the performance gain from the overlaps between computations and memory accesses.

Thus, we still apply issue width as 1 (i.e., same as original single issue controller) for classifier layers. To gain more insight, we normalize the performance (1.16×) to the area (6.38 mm^2 versus 6.43 mm^2) and observe 1.15× gain in terms of performance/area for the convolutional layer, which further validates the efficiency of our multi-issue controller.

C. Put All Together: Optimized Cambricon-X

We optimized Cambricon-X with a neuron encoder/decoder, optimized PEFUs and a multi-issue controller for potential efficiency. The neuron encoder/decoder can leverage activation sparsity by dynamically encoding/decoding neurons to reduce DRAM memory accesses. The optimized PEFU can leverage activation sparsity by gating the operations for unnecessary computations. The multi-issue controller can issue independent instructions in a cycle, thus parallelizing the data movement instructions and the computation instructions for potential performance gain. We observe that with additional 0.16 mm^2 area cost (2.51%), the optimized Cambricon-X can achieve 1.28× energy efficiency for DRAM memory accesses, 1.81× energy efficiency in PEFUs and 1.16× performance gain in the convolutional layers compared with the original Cambricon-X.

VIII. Discussion

A. Pruning Neurons

Although a neuron can be pruned when all its input synapses have been removed, in our current implementation, the convolutional layer cannot greatly benefit from such neuron pruning. The reasons are as follows. For simplicity, we only assume that there is only one pruned neuron in the convolutional layer. As the pruned neuron will not consume computational resource in the mapped PE, a new neuron should be assigned to the PE to avoid pipeline stall. In this case, the PE will process a new neuron on another output feature map with the same location (as the pruned neuron), in order to maximize the reuse of input neurons. Thus, the addresses of all output neurons that are assembled by PEs at different cycles will not be aligned to T_n. In fact, the arrangement of such unaligned data in NBout would incur considerable hardware cost for the convolutional layer, because the neurons of the same output feature maps are stored orderly. On contrary, the outputs of the classifier layer simply constitute a vector, i.e., a feature map with size of 1×1, and thus it does not require to align different neurons on the same feature map. In other words, only the classifier layer can benefit from the neuron pruning.

B. DaDianNao

We also investigate the DaDianNao architecture, a large-scale NN accelerator containing 16 tiles (each has the same computational ability as *all* PEs in our accelerator), a 36 MB on-chip eDRAM, and the eDRAM router between them [12]. The eDRAM and tiles are connected with a shared data bus, and thus all tiles receive the same inputs broadcasted from the eDRAM. There are two intuitive options to extend the above DaDianNao architecture for efficiently supporting sparse NNs. The first option is to integrate an indexing unit into each tile, so as to select needed neurons from received data within the tile. However, this solution advances high bandwidth requirement between the eDRAM and all tiles, as the size of received data is several time higher than that of our accelerator. For example, given a NN with the sparsity of 90%, in DaDianNao, each tile consumes ten cycles with the original 256-bit data bus to fetch the original 160 16-bit data. Since only 16 data are useful for computation, in our accelerator, each PE only consumes 1 cycle on average to fetch data. In other words, for sparse NNs, the bandwidth requirement of DaDianNao is 10× higher than our accelerator. The second option is to offer a central indexing unit to select needed data and then send selected data for each tile sequentially. This solution is also inefficient due to the contention of the shared data bus. In summary, the DaDianNao architecture cannot be extended in a straightforward fashion to exploit the sparsity and irregularity of modern NNs to achieve high efficiency as the accelerator proposed in this paper.

IX. Related Work

In this section, we mainly introduce related work on accelerators for efficient processing of NNs.

GPUs with mature libraries (e.g., cuBLAS) and frameworks (e.g., Caffe [30] and Tensorflow [36]) are the most widely used platform for NNs in both academic research and industrial practice. To adapt to the sparsity of modern NNs, sparse libraries such as cuSparse are also used for accelerating NN processing on GPUs.

FPGAs are also deployed for processing NNs, such MLP [37]–[39] or CNN/DNN [40]–[42]. However, the relatively low operation frequency limits broad applications of FPGAs for accelerating NNs.

There exists many ASIC-implemented accelerators for NNs [11]–[13], [43]–[46], but they cannot exploit the sparsity and irregularity of modern NNs. Recently, several researchers have been rising to such challenges. Pragmatic [47] eliminates most of the ineffectual computations (zero bits in activation values) on-the-fly, thus improving performance and energy efficiency compared to state-of-the-art high-performance accelerators. Eyeriss [48] applies run-length compression and PE data gating to exploit zeros in the feature maps (activation sparsity), thus saving DRAM bandwidth and PE energy consumption. However, it does not bring in performance gain. Cnvlutin [49] aims to exploit activation sparsity and improves performance by 1.37× with an area overhead of 4.49%. EIE [50] supports activation sparsity and synapse sparsity. It achieves 2.9×, 19×, and 3× better throughput, energy efficiency, and area efficiency, respectively, when compared with DaDianNao. However, it only targets the fully connected layer of DNNs, which only takes a small ratio of computation time of CNNs. ESE [51] is implemented on FPGA for sparse LSTM model. It only aims at the LSTM layer and is not suitable for the conventional layer. Recent accelerator SCNN [52] can exploit both synapse sparsity and activation sparsity. But it introduces in extra costs for storing and computing the coordinates, thus hindering the efficiency. SCNN only achieves 79% of the performance, but consumes 33% more energy when processing dense networks. It only achieves 2.7× and 2.3× improvement in performance and energy efficiency. Substantially, our accelerator 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. Eyeriss cannot exploit sparsity for performance gain. Regarding the generality, EIE and ESE are only capable on FC layers and LSTM layers, respectively.

X. Conclusion

In this paper, we propose a novel accelerator (Cambricon-X) which can effectively cope with not only the traditional dense NNs but also the pruned sparse NNs. The accelerator features a PE-based architecture consisting of the BC and multiple PEs. The BC integrates an IM for selecting necessary neurons for PEs. Each PE stores irregular and compressed synapses for local computation, and all of them work in an asynchronous manner. With a footprint of 6.38 mm^2 and 954 mW, our accelerator is able to perform 16 output neurons with sparse connections simultaneously, yielding 544 GOP/s at most. Compared with a state-of-the-art NNs accelerator, DianNao, our accelerator achieves 7.23× and 6.43× better performance and energy efficiency, respectively. We optimize Cambricon-X by leveraging activation sparsity, which achieves 1.28× and 1.81× better energy efficiency for DRAM memory accesses and PEFUs, respectively. Further with multi-issue controller, we observe additional 1.16× performance gain on the convolutional layers. To ease the burden of programmers, we also propose a library-based programming environment for our accelerator.

References

[1] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," in *Proc. Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 37–47.

[2] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proc. Annu. Int. Symp. Comput. Archit. (ISCA)*, Portland, OR, USA, 2012, pp. 356–367.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Vancouver, BC, Canada, 2012, pp. 449–460.

[4] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke, "Bridging the computation gap between programmable processors and hardwired accelerators," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Raleigh, NC, USA, 2009, pp. 313–322.

[5] G. Venkatesh *et al.*, "QSCORES: Trading dark silicon for scalable energy efficiency with quasi-specific cores categories and subject descriptors," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2011, pp. 163–174.

[6] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Raleigh, NC, USA, 2009, pp. 277–288.

[7] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Vancouver, BC, Canada, 2013, pp. 8609–8613.

[8] V. Mnih and G. Hinton, "Learning to label aerial images from noisy data," in *Proc. 29th Int. Conf. Mach. Learn. (ICML)*, Edinburgh, U.K., 2012, pp. 567–574.

[9] P.-S. Huang *et al.*, "Learning deep structured semantic models for Web search using clickthrough data," in *Proc. Int. Conf. Inf. Knowl. Manag. (CIKM)*, 2013, pp. 2333–2338.

[10] B. Liang and P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technol. Intel Mag.*, vol. 9, no. 2, pp. 1–10, 2005. [Online]. Available: http://www.intel.com/technology/itj/2005/volume09issue02/foreword.htm

[11] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Salt Lake City, UT, USA, 2014, pp. 269–284.

[12] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Cambridge, U.K., 2015, pp. 609–622.

[13] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Portland, OR, USA, 2015, pp. 92–104.

[14] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie, "Neural network stream processing core (NnSP) for embedded systems," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCS)*, 2006, pp. 2773–2776.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.

[16] Q. V. Le *et al.*, "Building high-level features using large scale unsupervised learning," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Edinburgh, U.K., 2012, pp. 8595–8598.

[17] A. Coates *et al.*, "Deep learning with COTS HPC systems," in *Proc. 30th Int. Conf. Mach. Learn. (ICML)*, Atlanta, GA, USA, 2013, pp. 1337–1345.

[18] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, Jun. 2014.

[19] M. A. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Efficient learning of sparse representations with an energy-based model," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2006, pp. 1137–1144.

[20] M. A. Ranzato, Y.-L. Boureau, and Y. LeCun, "Sparse feature learning for deep belief networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Vancouver, BC, Canada, 2007, pp. 1185–1192.

[21] H. Lee, C. Ekanadham, and A. Y. Ng, "Sparse deep belief net model for visual area V2," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2008, pp. 873–880.

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

[23] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 1135–1143.

[24] NVIDIA. *The NVIDIA CUDA Sparse Matrix Library (CUSPARSE)*. Accessed: Aug. 23, 2018. [Online]. Available: https://developer.nvidia.com/cusparse

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

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

[27] 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 Trans. Math. Softw.*, vol. 28, no. 2, pp. 239–267, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/567806.567810

[28] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" in *Proc. IEEE 12th Int. Conf. Comput. Vis. (ICCV)*, Kyoto, Japan, 2009, pp. 2146–2153.

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

[30] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[31] A. Rafique, G. A. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix–vector multiply on GPUs and FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 24–34, Jan. 2015.

[32] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, 2016.

[33] A. Krizhevsky. *Cuda-Convnet: High-Performance C++/Cuda Implementation of Convolutional Neural Networks*. Accessed: Aug. 23, 2018. [Online]. Available: https://code.google.com/p/cuda-convnet/

[34] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Chicago, IL, USA, 2007, pp. 3–14.

[35] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Stat.*, 2011, pp. 315–323.

[36] *TensorFlow: An Open Source Software Library for Machine Intelligence*, Google Inc., Mountain View, CA, USA, 2015. [Online]. Available: http://www.tensorflow.org/

[37] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.

[38] R. G. Girones *et al.*, "FPGA implementation of a pipelined on-line backpropagation," *J. VLSI Signal Process. Syst. Signal Image Video Technol.*, vol. 40, no. 2, pp. 189–213, 2005.

[39] E. Ordoñez-Cardenas and R. de J. Romero-Troncoso, "MLP neural network and on-line backpropagation learning implementation in a low-cost FPGA," in *Proc. 18th ACM Great Lakes Symp. VLSI*, Orlando, FL, USA, 2008, pp. 333–338.

[40] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl. Spec. Syst. Archit. Process. (ASAP)*, Boston, MA, USA, 2009, pp. 53–60.

[41] C. Farabet, C. Poulet, J. Y. Han, and Y. Lecun, "CNP: An FPGA-based processor for convolutional networks," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, 2009, pp. 32–37.

[42] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale convolutional networks," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, San Jose, CA, USA, 2011, pp. 2809–2813.

[43] T. Chen *et al.*, "A small-footprint accelerator for large-scale neural networks," *ACM Trans. Comput. Syst.*, vol. 33, no. 2, 2015, Art. no. 6.

[44] C. Farabet *et al.*, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, 2011, pp. 109–116.

[45] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 247–257.

[46] V. Gokhale, J. Jin, and A. Dundar, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Columbus, OH, USA, 2014, pp. 682–687. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6910056

[47] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *Proc. ACM 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, Cambridge, MA, USA, 2017, pp. 382–394.

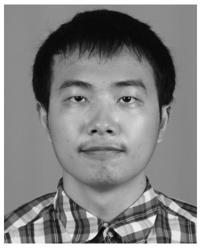
[48] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[49] J. Albericio *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, 2016, pp. 1–13.

[50] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, vol. 16, 2016, pp. 243–254. [Online]. Available: http://arxiv.org/abs/1602.01528

[51] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Monterey, CA, USA, 2017, pp. 75–84.

[52] P. Angshuman *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Int. Symp. Comput. Archit.*, 2017, pp. 27–40.



Xuda Zhou is currently pursuing the Ph.D. degree with the University of Science and Technology of China, Hefei, China, under the guidance of Prof. X. Zhou.

He is also a visiting student with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, under the guidance of Prof. Y. Chen. His current research interests include computer architecture, hardware neural networks, and neural network compression.

Zidong Du (M'16) received the Bachelor of Engineering degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2011.

He is currently an Assistant Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interests include computer architecture, inexact computing, and hardware neural networks.

Shijin Zhang received the Ph.D. degree from the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2017.

He is currently an Assistant Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences.

Lei Zhang is currently pursuing the Ph.D. degree with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, under the guidance of Prof. Y. Chen.

Huiying Lan is currently pursuing the Ph.D. degree with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, under the guidance of Prof. Y. Chen.

Shaoli Liu received the bachelor's degree from Nankai University, Tianjin, China, in 2009 and the Ph.D. degree from the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2014.

He is currently an Associate Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences.

Ling Li received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2009.

She is a Professor with the Institute of Software, Chinese Academy of Sciences. She joined the Godson (Loongson) Project in 2009. She is the Chief Architect of Godson video decoding IP. She has authored or co-authored papers on journals (including the IEEE TRANSACTIONS ON IMAGE PROCESSING, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *IET Image Processing*) and conferences (including DCC, SPAA, and ICASSP).

Qi Guo received the bachelor's degree from Tongji University, Shanghai, China, in 2007 and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2012.

He is an Associate Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include computer architecture and performance analysis.

Tianshi Chen received the bachelor's degree in mathematics (Special Class for the Gifted Young) from the University of Science and Technology of China (USTC), Hefei, China, in 2005 and the Ph.D. degree from the School of Computer Science and Technology, USTC in 2010, under the supervision of Prof. G. Chen and Prof. X. Yao.

He is currently the CEO of Cambricon Tech. Ltd., Beijing, China. He was a Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interests include computer architecture and computational intelligence.

Yunji Chen (SM'15) received the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2007.

He is a Professor with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include parallel computing, microarchitecture, and computational intelligence. He has authored or co-authored one book and over 60 papers in the above areas.

Dr. Chen was a recipient of the Best Paper Awards from ASPLOS14 and MICRO14 for the investigations in neural network accelerators.