# Cambricon-X: An Accelerator for Sparse Neural Networks

Shijin Zhang[1,2,4], Zidong Du[1,2], Lei Zhang[1,2,4], Huiying Lan[1,2,4], Shaoli Liu[1,2], Ling Li[3], Qi Guo[1,2], Tianshi Chen[1,2], Yunji Chen[1]

[1] SKL of Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
[2] Cambricon Technologies Co., Ltd., China
[3] Institute of Automation, CAS, Beijing, China
[4] University of Chinese Academy of Sciences, Beijing, China
Email: {zhangshijin, duzidong, zhanglei01, lanhuiying, liushaoli, liling, guoqi, chentianshi, cyj}@ict.ac.cn

***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 neural networks 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 PE-based architecture consisting of multiple *Processing Elements (PE)*. An *Indexing Module (IM)* 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, 7.23x speedup and 6.43x energy saving against the state-of-the-art NN accelerator.**

## 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 trade-off 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 $\sim$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 neural networks 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]. Figure 1 shows the neurons and synapses of a fully-connected MLP (multilayer perceptron), 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 10x with negligible accuracy loss.

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 neural networks 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 1.78x 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

978-1-5090-3508-3/16/$31.00 

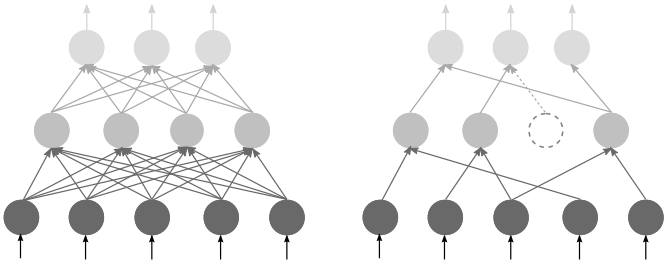

Fig. 1. (a) A fully-connected MLP. (b) A sparse MLP.

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 neural networks but also heavily pruned sparse ones[1]. The accelerator features a PE-based architecture consisting of multiple *processing elements (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 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.

We evaluate our accelerator, Cambricon-X, with a number of representative NNs (including LeNet-5 [25], AlexNet [15], and VGG16 [26], etc.) with various sparsity levels. Compared against the state-of-the-art NN accelerator, DianNao, on average, our accelerator achieves 7.23x speedup and 6.43x 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 10.60x speedup and 29.43x energy reduction. Compared against the CPU with the sparse library (i.e., Sparse BLAS [27]), on average, our accelerator achieves 144.41x speedup.

## II. Background and Motivation

In this section, we first introduce primer on neural networks, 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

**State-of-the-art neural networks.** The state-of-the-art neural network algorithms are Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs). Usually, a CNN or a DNN includes four types of layers, i.e., convolutional layer, pooling layer, classifier layer and normalization layer. The main difference between CNNs and DNNs is the convolution layer, where synapses are *shared* in CNNs but *private* in DNNs. Figure 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.

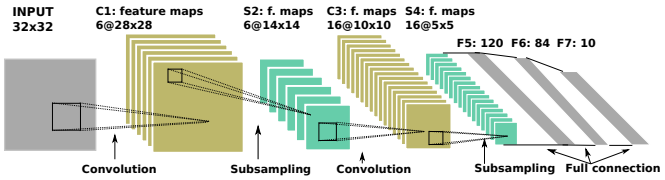


Fig. 2. A typical CNN: LeNet-5 [25].

**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 (DBN) [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 neural network 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 1, 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.

TABLE I
Numbers of neurons and synapses in sparse NNs (C for convolutional layer; F for classifier layer).

| NN | Non-sparse | | Sparse | | | |
|---|---|---|---|---|---|---|
| | # Neurons | # Synapses | # Synapses | Sparsity (%) | # Synapses in C. | # Synapses in F. |
| LeNet-5 [25] | 8.90K | 430.62K | 36.30K | 8.43 | 3.33K | 32.97K |
| AlexNet [15] | 1.28M | 60.95M | 6.80M | 11.15 | 864.86K | 5.93M |
| VGG16 [26] | 14.53M | 138.34M | 10.53M | 7.61 | 4.81M | 5.72M |

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

[2]Two typical types of normalization layers are Local Contrast Normalization Layer (LCN) and Local Response Normalization Layer (LRN), which were proposed in 2009 [28] and 2012 [15] respectively.

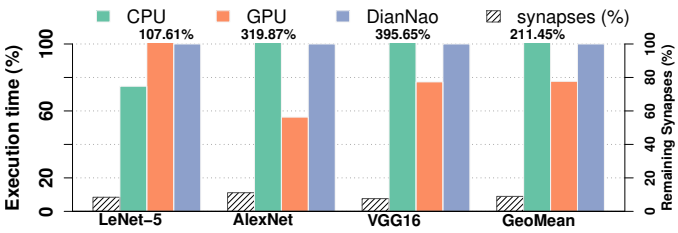

Fig. 3. The speedup of sparse NN vs. dense NN on CPU, GPU and DianNao.

### 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. Figure 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 vs. Caffe [30]) and GPU (cuSPARSE vs. 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 non-computational 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 Figure 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, 2) neither the centralized architecture (e.g., DianNao) nor the symmetric tiled architecture (e.g., DaDianNao) can adapt to the irregularity of sparse NNs.

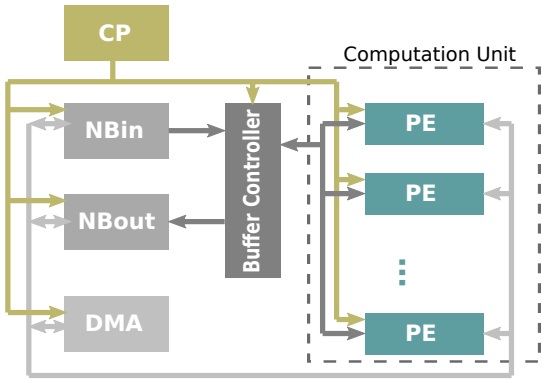

Fig. 4. Accelerator architecture.

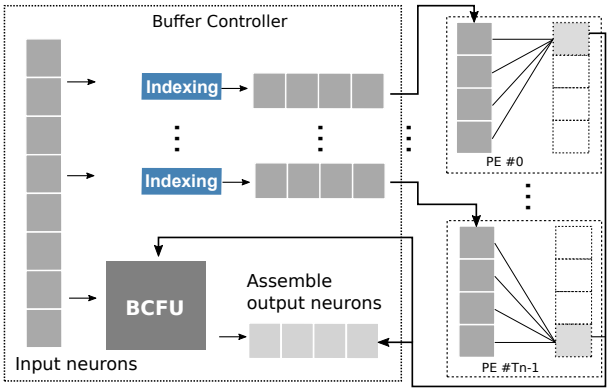

Fig. 5. Buffer controller architecture.

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

## III. Accelerator Design

In this section, we present detailed architecture of our accelerator.

### A. Overview

Figure 4 presents the proposed architecture of our accelerator, which consists of a control processor (CP), a buffer controller (BC), two neural buffers (NBin and NBout), a direct memory access module (DMA) and a computation unit (CU) which contains multiple processing elements (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 Figure 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.

In this design, we use 16-bit fixed-point arithmetic units rather than conventional 32-bit floating-point units. The main

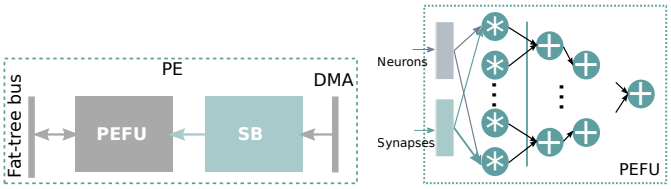


Fig. 6. (a) The architecture of the PE. (b) The architecture of the PEFU.

reason is that, with negligible accuracy loss, the 16-bit fixed-point unit has significantly lower hardware cost than the 32-bit floating-point unit, as validated by previous studies [2], [11], [32]. More specifically, a 16-bit truncated fixed-point multiplier is 6.10x smaller and 7.33x more energy-efficient than a 32-bit floating-point multiplier at TSMC 65nm technology [11]. Furthermore, the width of data buses will be reduced by half by using 16-bit data representation.

### B. Computation Unit

The Computation Unit is designed for efficient computation of the core operation of neural networks, i.e., the vector multiplication-addition operation, with multiple PEs. Figure 6(a) shows the architecture of the PE, consisting of a synapse buffer (SB) and neural network 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.

**PEFU.** The PEFUs are mainly used for multiplication-addition operations in neural networks. A single PEFU consists of several multipliers, say $T_m$, as well as $T_m$-in adder-tree, see Figure 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 frequency, we pipeline the functional units in the PEFU into 2 stages: the multiplication and the addition of all the multiplication results. With $T_m$ inputs, all $T_n$ PEs can produce $T_n$ output neurons simultaneously.

**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 previous work proposed to offer large enough buffers for holding all synapses of neural networks 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., $\sim$7 MB for AlexNet, $\sim$10M for VGG as listed in Table I. Second, our accelerator is designed for supporting neural networks 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 2KB SB in each PE, leading to $2 \times T_n$KB storage in total for synapses.

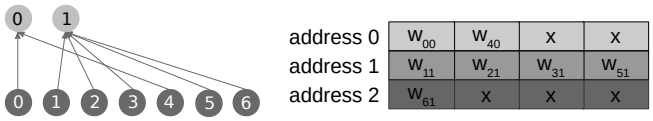


Fig. 7. (a) An example of sparse connection. (b) The data organization in the SB.

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 7 input neurons and 2 output neurons, as shown in Figure 7(a). We assume that this network is mapped to only one PE with parameter $T_m = 4$, and we use $w_{i,j}$ to denote the synaptic weight between input neuron $\#i$ and output neuron $\#j$. The weights of synapses connected to different output neurons are stored in the SB compactly by aligning to $T_m$ (i.e., 4). As shown in Figure 7(b), the two weights of output neuron 0 (i.e., $w_{00}$ and $w_{40}$) are stored in address 0 one by one while the five weights of output neuron 1 (e.g., $w_{11}$ and $w_{61}$) are stored in subsequent addresses: addresses 1 and 2. Hence, when computing output neuron 0, the SB only needs to be read once, while for output neuron 1, it needs to be read twice. 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 buffer controller is designed for transferring necessary neurons to PEs, orchestrating computations on PEs, and performing less computation-intensive operations. Figure 8 illustrates the architecture of the BC, which consists of a module used to index data for computation based on connections (Indexing Module, IM), and the specialized function units for the BC (BCFU). At first, inputs are fetched from NBin based on the control signals decoded from instructions (e.g., memory access instruction). Then, either the needed neurons are selected from the inputs and transferred to each PE or the inputs are directed fed into the BCFU. After completing computation in PEs, the results are collected for further processing on BCFU or directly wrote back to NBout. Note that, the accelerator can bypass the IM when processing dense NNs to avoid the potential slowdown due to IM.

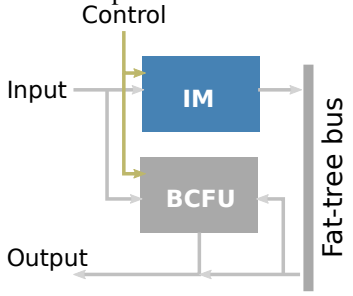


Fig. 8. The architecture of the buffer controller.

**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.

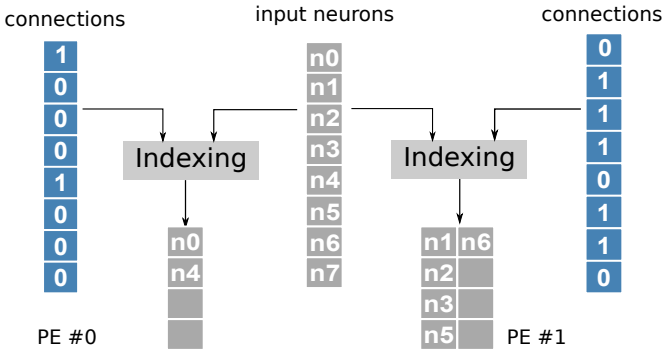


Fig. 9. The functionality of IM module.

**IM.** The IM is the key component of our accelerator, and it is used for indexing needed neurons of sparse neural networks with different levels of sparsity. Instead of distributing an indexing module to each PE, we design a centralized indexing module 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 Figure 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 8 neurons for computation on PEs.

To implement the indexing module, 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 ("1"s 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 (COO), 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 one-dimensionally in parallel. Thus we investigate *direct indexing* and *step indexing* for implementing high efficient indexing module.

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 is shown in Figure 10(a). We also present the potential hardware implementation in Figure 10(b). The indexing process can be elaborated as follows. First we add each bit in the original binary string to obtain an accumulated string, and each element in the accumulated string indicates the location of corresponding connection. After enforcing the "AND" operation between

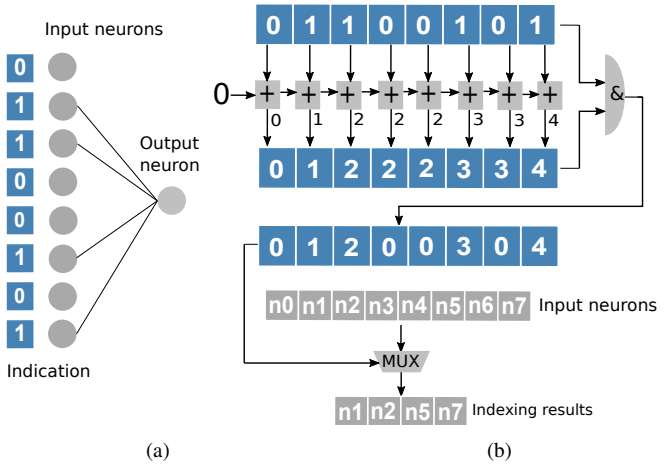


Fig. 10. (a) A sparse network example with the direct indexing. (b) Hardware implementation of direct indexing.

the accumulated string and the original string, the indexes of each connected neuron can be obtained.

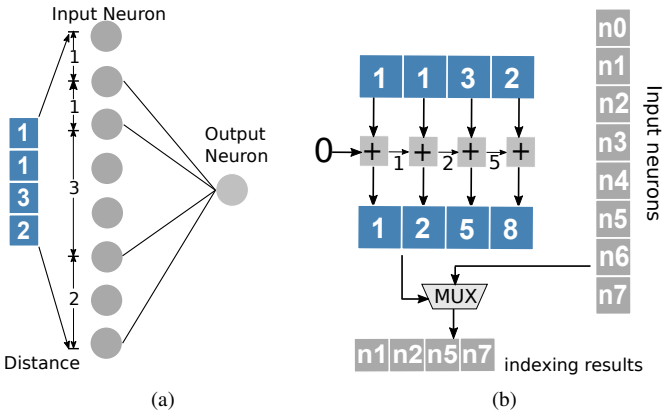


Fig. 11. (a) A sparse network example with the step indexing. (b) Hardware implementation of the step indexing.

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 Figure 11(a) and the potential hardware implementation in Figure 11(b). The indexing process can be detailed as follow. First, we add the numbers in the index table (e.g., "1132" in Figure 11(b)) sequentially to get the indexes of inputs neurons which have connections with the current output neuron. Then, such indexes are used for addressing the corresponding input neurons. Compared against the direct indexing, the indexes in step indexing are integer numbers whose widths depend on the sparsity of NNs.

We implement the above two indexing approaches in RTL and compare corresponding hardware costs in terms of area and power with synthesized results in Figure 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.

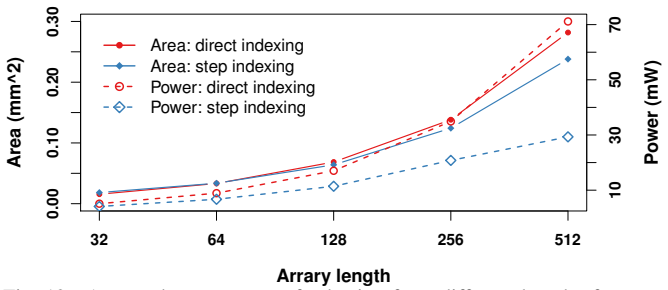


Fig. 12. Area and power costs of selecting from different length of array at a time.

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, and memory accesses, etc., and they are stored in a small instruction buffer. 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 8KB is the optimal tradeoff between the achieved performance and related energy consumption. Therefore, in our accelerator implementation, we use 8KB for both NBin and NBout. Note that, the size of NBin is larger that that of SB (for storing synapses, 2KB) since the accelerator needs to store more input neurons before selecting necessary neurons for PEs.

Apparently, 8KB NBs cannot hold all neurons of large-scale neural networks, 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.

### F. Interconnect and communication

**Interconnect.** We employ the Fat-tree [33] interconnect topology, which features that more data links are provided near the top of the interconnect hierachy, to connect the BC and all the PEs, in order to improve the efficiency of data movement between them. There are two reasons to use the Fat-tree interconnect: 1) compared to other non-tree interconnect topologies, using Fat-tree can avoid the long critical path caused by unbalanced delays between the BC and PEs, and 2) compared to other tree-like interconnect topologies, Fat-tree can provide private connection to alleviate network congestion, as data sent to different PEs are independent.

**Communication.** The data communication between off-chip memory and on-chip buffers (including NBin, NBout, and SB) are implemented through Direct Memory Access (DMA). To balance the execution of different PEs and avoid the congestion of memory access, we first split the required synapses into chunks. Then, the memory access port will be assigned to only one PE at a time for a short period, thus each PE will be able to load only several chunks during that period. In this case, each PE would have some synapses, if not all, to perform the corresponding computation at different cycles. Such asynchronous computation pattern can reduce memory congestion by allowing different PEs to compute at different cycles.

## IV. Mapping

The representative types of layers include the convolutional, pooling, classifier, and normalization layer. A convolutional layer constructs the output feature maps through convolving multiple input feature maps with shared or private kernels, and occupies about 85% computational time of the entire network processing. The data of feature maps are stored in the on-chip buffer in the order of map id. During the computing process of a convolutional layer, all output neurons will complete the computation related to the data in NBin of the BC before replacing with new input neurons, in order to maximize the reuse of input neurons in NBin to reduce off-chip memory accesses.

The pooling layer downsamples input feature maps to construct output feature maps, by performing either *maximum* or *average* operation on a 2D pooling window. For pooling layer, the data could be efficiently mapped to the BC, and is processed similarly to that in a convolutional layer.

The classifier layer performs classification according to the features extracted from previous layers. In contrast to the convolutional layer, there is no sharing of synaptic weights between input-output neuron pairs in the classifier layer. Hence, the mapping of synaptic weights onto the on-chip buffer aims at maximizing the reuse of input and output neurons.

The normalization layer performs normalization over local input regions, and it can be decomposed into a number of sub-layers and fundamental computational primitives such as

element-wise square, matrix addition, and divisions, etc. [13]. Thus, we can leverage previous methods to process the normalization layer.

The computing processes for the convolutional layer, the pooling layer, the classifier layer and the normalization layer are similar to that in DaDianNao [12].

## V. Programming Model

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

```
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.

### B. Programing 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. Figure 13 shows the programming process of our accelerator. Initially, we use the sparse neural network model obtained from the training phase, and the corresponding sparse representation to create a compact neural network model file. Then, the compact model file, the neural network configuration, and the input data, are sent to Caffe. In Caffe, our library functions are invoked to generate outputs. For the dense neural network, the model file only contains the dense neural network model gained from the training phase. In this case, the underlying network format is entirely transparent to the user.

## VI. Experimental Methodology

In this section, we introduce the experimental methodology.

**Bechmarks.** We use 6 representative neural networks, i.e., LeNet-5, AlexNet, VGG, Dropout NN1 (2-layer MLP, 800 hidden neurons), Dropout NN2 (3-layer MLP, 8192×8192 hidden neurons), and Cifar10 quick model [34] 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.

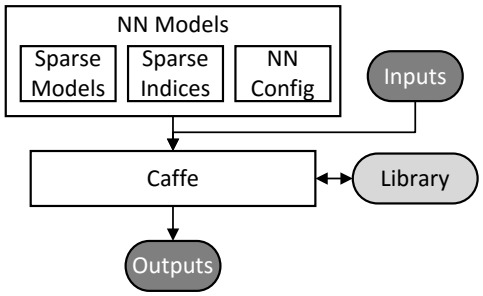


Fig. 13. Programming process of our accelerator.

**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 65nm Gplus High VT library. We use CACTI 6.0 to estimate the energy consumption of DRAM accesses [35].

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

**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 neural networks, we implement the evaluated benchmarks with the most widely used sparse library, i.e., sparseBLAS [27] (CPU-Sparse), on the CPU.

**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 on the GPU with state-of-the-art cuSparse library [24] (GPU-cuSparse).

**Accelerator:** We also compare our accelerator against the state-of-the-art neural network 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 $16 \times 16$ multipliers, 16 16-in adder trees and 16

TABLE II
Benchmarks with remaining synapses in each layer (C for convolutional layers; F for classifier layers; Total for all layers).

| — | LeNet-5 [25] | | AlexNet [15] | | VGG16 [26] | |
|---|---|---|---|---|---|---|
| | —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 [18] | | Dropout NN2 [18] | | Cifar10 [34] | |
| | —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% |

non-linear modules.

## VII. 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 non-linear 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 2.11x larger than DianNao with 6.38 $mm^2$ vs. 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 III
Hardware parameters of accelerators.

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

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 |

### 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 Figure 14, we normalize all the performance numbers of the above implementations to that of our accelerator for

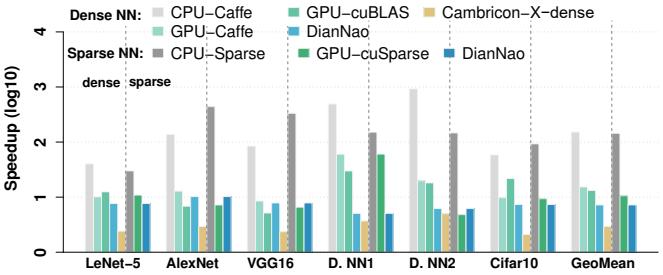


Fig. 14. The speedup of our accelerator over CPU, GPU and DianNao for all evaluated NNs, including both the dense and sparse implementations. All the performance numbers of different implementations are normalized to that of our accelerator with sparse networks.

sparse representation. Regarding implementations for dense representation, on average, our accelerator is 51.55x, 5.20x and 4.94x faster than CPU-Caffe, GPU-Caffe and GPU-cuBLAS, respectively, on the evaluated benchmarks (Our accelerator for sparse representation achieves 151.82x, 15.32x and 13.18x respectively). Regarding the sparse representation, on average, our accelerator is 144.41x and 10.60x faster than CPU-Sparse and GPU-Sparse, respectively. Compared against DianNao, our accelerator still achieves 7.23x 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 2.46x 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 Figure 15 and Figure 16, respectively, where all the performance numbers are normalized to that of our accelerator with sparse representation. In Figure 15, for the convolutional layer, on average, our accelerator is 8.90x, 7.67x and 8.89x faster than GPU-cuBLAS, GPU-cuSparse, and DianNao, respectively. In Figure 16, for the classifier layer, on average, our accelerator is 44.28x, 20.07x, and 5.99x 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% vs. 22.65%) on the evaluated benchmarks. This is also validated by the observation that our accelerator achieves 2.51x and 4.84x speedup over Cambricon-X-dense for the convolutional and classifier layers, respectively.

### C. Energy

In Figure 17, 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 37.79x and 29.43x better energy efficiency for the dense and sparse networks, respectively. Compared to DianNao, on average, our accelerator achieves 6.43x better energy efficiency for both the dense and sparse networks. We made an interesting observation that the best energy efficiency achieved

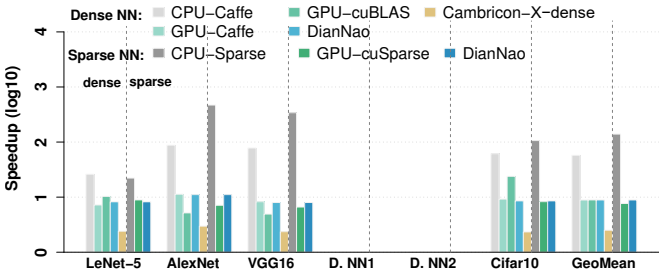


Fig. 15. The speedup of our accelerator over CPU, GPU and DianNao for convolutional layers. Note that there is no convolutional layer for Dropout NN1 and Dropout NN2.

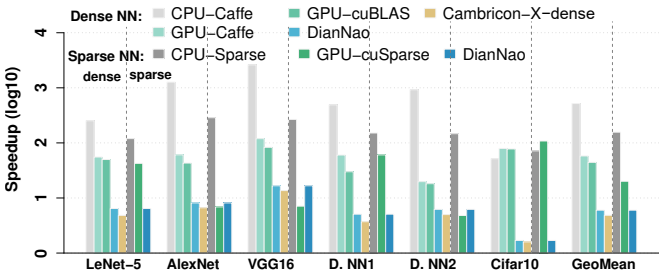


Fig. 16. The speedup of our accelerator over CPU, GPU and DianNao for classifier layers.

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 1.70x 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 Figure 18, 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% vs. 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% vs. 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.

## VIII. Discussion

### A. Sensitivity to network sparsity

We investigate the sensitivity of hardware platforms to different levels of network sparsity. Figure 19 shows the speedup of sparse network layers (i.e., the convolutional layer

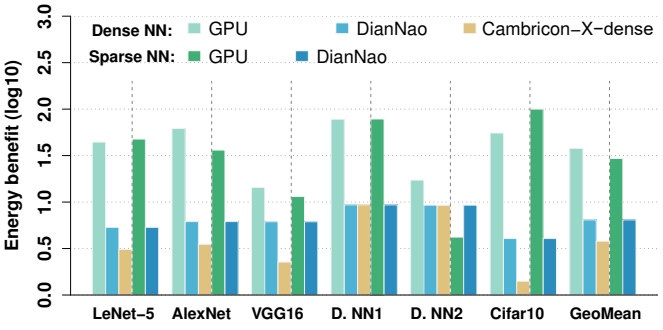


Fig. 17. Energy benefit of our accelerator over GPU (GPU-Caffe and GPU-sparse for the dense and sparse representation, respectively) and DianNao, where all the results are normalized to that of our accelerator with sparse networks.

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 non-trivial 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 (SpMM) 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, and 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 neural network models.

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

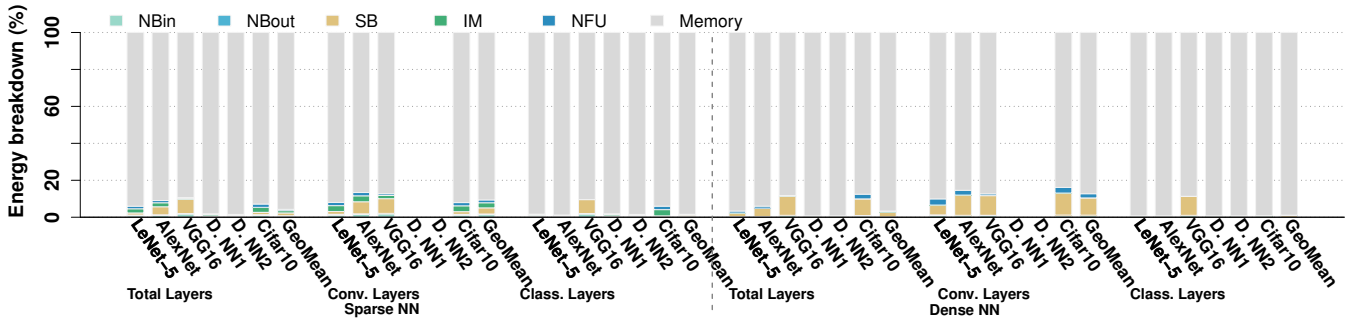


Fig. 18. Energy breakdown with memory accesses.

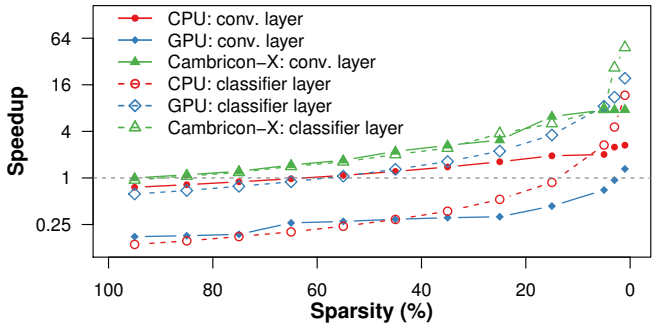


Fig. 19. Speedup of sparse layer over dense layer.

Also, it is clear that our accelerator are more adaptive to future neural networks with far lower sparsity, i.e., $< 1\%$, compared against to existing hardware platforms.

### B. 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. We discuss the underlying reason 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 $1 \times 1$, 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.

### C. DaDianNao

We also investigate the DaDianNao architecture, a large-scale neural network accelerator containing 16 tiles (each has the same computational ability as *all* PEs in our accelerator), a 36MB 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 neural networks. 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 neural network with the sparsity of 90%, in DaDianNao, each tile consumes 10 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 neural networks, the bandwidth requirement of DaDianNao is 10x 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 neural networks 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 neural networks.

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

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

There exists many ASIC-implemented accelerators for neural networks. Chen et al. proposed DianNao [11], [43] to accelerate various neural networks in a flexible fashion. DaDianNao is proposed for efficiently processing large-scale neural networks with sufficient on-chip memory as eDRAM [12]. Based on the observation that the DRAM access is the main

bottleneck of neural network processing, Du et al. [13] proposed ShiDianNao to completely eliminate off-chip memory accesses in embedded systems. Although the above accelerators can achieve high energy efficiency for servers or embedded systems, they cannot exploit the sparsity and irregularity of modern neural networks. As convolutional neural networks recently gain more attentions, Farabet et al. proposed a systolic architecture called NeuFlow architecture [44] and Chakradhar et al. designed a systolic-like coprocessor [45] to handle 2D convolution efficiently in a CNN where the convolutional layer occupies about 85% computational time of the entire network processing. Gokhale et al. [46] designed a mobile coprocessor for visual processing at mobile devices, which supports both CNNs and DNNs. Han *et al* proposed EIE [47] for leveraging the sparsity of full-connected layers in neural networks with CSC sparse representation scheme, which is a two-dimensional indexing method thus not as efficient as our accelerator.

Substantially different from above designs, our accelerator aims at high efficiency for both dense and sparse neural networks.

## X. Conclusions

In this paper, we propose a novel accelerator (Cambricon-X) which can effectively cope with not only the traditional dense neural networks but also the pruned sparse neural networks. The accelerator features a PE-based architecture consisting of the BC and multiple PEs. The BC integrates an indexing module 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 neural network accelerator, DianNao, our accelerator achieves 7.23x and 6.43x better performance and energy efficiency respectively.

## Acknowledgment

This work is partially supported by the NSF of China (under Grants 61133004, 61303158, 61432016, 61472396, 61473275, 61522211, 61532016, 61521092), the 973 Program of China (under Grant 2015CB358800), the Strategic Priority Research Program of the CAS (under Grants XDA06010403, XDB02040009), the International Collaboration Key Program of the CAS (under Grant 171111KYSB20130002), and the 10000 talent program. Yunji Chen is the corresponding author.

## References

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, pp. 37–47, 2010.

[2] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *Proceedings of Annual International Symposium on Computer Architecture (ISCA)*, pp. 356–367, 2012.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 449–460, 2012.

[4] K. Fan, S. Mahlke, and A. Arbor, "Bridging the Computation Gap Between Programmable Processors and Hardwired Accelerators," in *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 313–322, 2009.

[5] G. Venkatesh, J. Sampson, N. Goulding-hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "QSCORES : Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores Categories and Subject Descriptors," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 163–174, 2011.

[6] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 277–288, 2009.

[7] G. Dahl, T. Sainath, and G. Hinton, "Improving deep neural networks for LVCSR using rectified linear units and dropout," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8609–8613, 2013.

[8] V. Mnih and G. Hinton, "Learning to Label Aerial Images from Noisy Data," in *Proceedings of the 29th International Conference on Machine Learning (ICML)*, pp. 567–574, 2012.

[9] P. S. Huang, X. He, J. Gao, and L. Deng, "Learning deep structured semantic models for web search using clickthrough data," in *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pp. 2333–2338, 2013.

[10] B. Liang and P. Dubey, "Recognition, Mining and Synthesis," *Intel Technology Journal*, vol. 09, no. 02, 2005.

[11] T. Chen, Z. Du, N. Sun, J. Wang, and C. Wu, "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)*, pp. 269–284, 2014.

[12] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 609–622, 2015.

[13] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, 2015.

[14] H. Esmaeilzadeh, P. Saeedi, B. Araabi, C. Lucas, and S. Fakhraie, "Neural Network Stream Processing Core (NnSP) for Embedded Systems," in *Proceedings of 2006 IEEE International Symposium on Circuits and Systems (ISCS)*, pp. 2773–2776, 2006.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[16] Q. V. Le, M. A. Ranzato, M. Devin, G. S. Corrado, and A. Y. Ng, "Building High-level Features Using Large Scale Unsupervised Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 8595 – 8598, 2012.

[17] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pp. 1337–1345, 2013.

[18] 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 (JMLR)*, vol. 15, pp. 1929–1958, 2014.

[19] M. A. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Efficient Learning of Sparse Representations with an Energy-Based Model," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pp. 1137–1144, 2006.

[20] M. A. Ranzato, Y.-L. Boureau, and Y. LeCun, "Sparse feature learning for deep belief networks," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pp. 1185–1192, 2007.

[21] H. Lee, C. Ekanadham, and A. Y. Ng, "Sparse deep belief net model for visual area V2," pp. 873–880, 2008.

[22] A. Y. N. Honglak Lee, Alexis Battle, Rajat Raina, "Efficient Sparse coding algorithms," in *Proceedings of Advances in Nerual Infromation Processing Systems (NIPS)*, pp. 801–808, 2006.

[23] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015.

[24] NVIDIA, "The nvidia cuda sparse matrix library (cusparse)," .

[25] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278– 2324, 1998.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/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, pp. 239–267, June 2002.

[28] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *Proceedings of 2009 IEEE 12th International Conference on Computer Vision (ICCV)*, pp. 2146–2153, 2009.

[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, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

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

[32] Z. Du, A. Lingamneni, Y. Chen, K. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators," in *Proceedings of 2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 201–206, 2014.

[33] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, pp. 892–901, Oct 1985.

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

[35] 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 (MICRO)*, pp. 3–14, 2007.

[36] Google Inc., "Tensorflow: an open source software library for machine intelligence," 2015.

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

[38] R. G. Girones, R. C. Palero, J. C. Boluda, and A. S. Cortes, "FPGA implementation of a pipelined on-line backpropagation," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 40, no. 2, pp. 189–213, 2005.

[39] F. D. Ingenieria, E. Ordoñez cardenas, and R. D. J. Romero-troncoso, "MLP Neural Network And On-Line Backpropagation Learning Implementation In A Low-Cost FPGA," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pp. 333–338, 2008.

[40] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A Massively Parallel Coprocessor for Convolutional Neural Networks," in *Proceedings of 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 53–60, 2009.

[41] C. Poulet, J. Y. Han, and Y. Lecun, "CNP: An FPGA-based processor for Convolutional Networks," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pp. 32–37, 2009.

[42] P. Sermanet and Y. LeCun, "Traffic sign recognition with multi-scale Convolutional Networks," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pp. 2809–2813, 2011.

[43] T. Chen, S. Zhang, S. Liu, Z. Du, T. Luo, Y. Gao, J. Liu, D. Wang, C. Wu, N. Sun, Y. Chen, and O. Temam, "A small-footprint accelerator for large-scale neural networks," *ACM Transactions on Computer Systems*, vol. 33, no. 2, 2015.

[44] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A runtime reconfigurable dataflow processor for vision," in *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 109–116, 2011.

[45] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, pp. 247–257, 2010.

[46] V. Gokhale, J. Jin, and A. Dundar, "A 240 G-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 682–687, 2014.

[47] 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 43th Annual International Symposium on Computer Architecture (ISCA'16)*, vol. 16, 2016.