2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Cambricon: An Instruction Set Architecture for Neural Networks

Shaoli Liu*§, Zidong Du*§, Jinhua Tao*§, Dong Han*§, Tao Luo*§, Yuan Xie†, Yunji Chen*‡ and Tianshi Chen*‡§
*State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
Email: {liushaoli, duzidong, taojinhua, handong2014, luotao, cyj, chentianshi}@ict.ac.cn
†Department of Electrical and Computer Engineering, UCSB, Santa Barbara, CA, USA
Email: yuanxie@ece.ucsb.edu
‡CAS Center for Excellence in Brain Science and Intelligence Technology
§Cambricon Ltd.

***Abstract*—Neural Networks (NN) are a family of models for a broad range of emerging machine learning and pattern recondition applications. NN techniques are conventionally executed on general-purpose processors (such as CPU and GPGPU), which are usually not energy-efficient since they invest excessive hardware resources to flexibly support various workloads. Consequently, application-specific hardware accelerators for neural networks have been proposed recently to improve the energy-efficiency. However, such accelerators were designed for a small set of NN techniques sharing similar computational patterns, and they adopt complex and informative instructions (control signals) directly corresponding to high-level functional blocks of an NN (such as layers), or even an NN as a whole. Although straightforward and easy-to-implement for a limited set of similar NN techniques, the lack of agility in the instruction set prevents such accelerator designs from supporting a variety of different NN techniques with sufficient flexibility and efficiency.**

In this paper, we propose a novel domain-specific Instruction Set Architecture (ISA) for NN accelerators, called Cambricon, which is a load-store architecture that integrates scalar, vector, matrix, logical, data transfer, and control instructions, based on a comprehensive analysis of existing NN techniques. Our evaluation over a total of ten representative yet distinct NN techniques have demonstrated that Cambricon exhibits strong descriptive capacity over a broad range of NN techniques, and provides higher code density than general-purpose ISAs such as x86, MIPS, and GPGPU. Compared to the latest state-of-the-art NN accelerator design DaDianNao [5] (which can only accommodate 3 types of NN techniques), our Cambricon-based accelerator prototype implemented in TSMC 65nm technology incurs only negligible latency/power/area overheads, with a versatile coverage of 10 different NN benchmarks.

I. Introduction

Artificial Neural Networks (NNs for short) are a large family of machine learning techniques initially inspired by neuroscience, and have been evolving towards deeper and larger structures over the last decade. Though computationally expensive, NN techniques as exemplified by deep learning [22], [25], [26], [27] have become the state-of-the-art across a broad range of applications (such as pattern recognition [8] and web search [17]), some have even achieved human-level performance on specific tasks such as ImageNet recognition [23] and Atari 2600 video games [33].

Traditionally, NN techniques are executed on general-purpose platforms composed of CPUs and GPGPUs, which are usually not energy-efficient because both types of processors invest excessive hardware resources to flexibly support various workloads [7], [10], [45]. Hardware accelerators customized to NNs have been recently investigated as energy-efficient alternatives [3], [5], [11], [29], [32]. These accelerators often adopt high-level and informative instructions (control signals) that directly specify the high-level functional blocks (e.g. layer type: convolutional/ pooling/ classifier) or even an NN as a whole, instead of low-level computational operations (e.g., dot product), and their decoders can be fully optimized to each instruction.

Although straightforward and easy-to-implement for a small set of similar NN techniques (thus a small instruction set), the design/verification complexity and the area/power overhead of the instruction decoder for such accelerators will easily become unacceptably large, when the need of flexibly supporting a variety of different NN techniques results in a significant expansion of instruction set. Consequently, the design of such accelerators can only efficiently support a small subset of NN techniques sharing very similar computational patterns and data locality, but is incapable of handling the significant diversity among existing NN techniques. For example, the state-of-the-art NN accelerator DaDianNao [5] can efficiently support the Multi-Layer Perceptrons (MLPs) [50], but cannot accommodate the Boltzmann Machines (BMs) [39] whose neurons are fully connected to each other. *As a result, the ISA design is still a fundamental yet unresolved challenge that greatly limits both flexibility and efficiency of existing NN accelerators*.

In this paper, we study the design of the ISA for NN accelerators, inspired by the success of RISC ISA design principles [37]: (a) First, decomposing complex and informative instructions describing high-level functional blocks of NNs (e.g., layers) into shorter instructions corresponding to low-level computational operations (e.g., dot product) allows an accelerator to have a broader application scope,

Yunji Chen (cyj@ict.ac.cn) is the corresponding author of this paper.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.42

as users can now use low-level operations to assemble new high-level functional blocks that are indispensable in new NN techniques; (b) Second, simple and short instructions significantly reduce design/verification complexity and power/area of the instruction decoder.

The result of our study is a novel ISA for NN accelerators, called Cambricon. Cambricon is *a load-store architecture* whose instructions are all 64-bit, and contains 64 32-bit General-Purpose Registers (GPRs) for scalars, mainly for control and addressing purposes. To support intensive, contiguous, variable-length accesses to vector/matrix data (which are common in NN techniques) with negligible area/power overhead, Cambricon does not use any vector register file, but keeps data in on-chip scratchpad memory, which is visible to programmers/compilers. There is no need to implement multiple ports in the on-chip memory (as in the register file), as simultaneous accesses to different banks decomposed with addresses' low-order bits are sufficient to supporting NN techniques (Section IV). Unlike an SIMD whose performance is restricted by the limited width of register file, Cambricon efficiently supports larger and variable data width because the banks of on-chip scratchpad memory can easily be made wider than the register file.

We evaluate Cambricon over a total of ten representative yet distinct NN techniques (MLP [2], CNN [28], RNN [15], LSTM [15], Autoencoder [49], Sparse Autoencoder [49], BM [39], RBM [39], SOM [48], HNN [36]), and observe that Cambricon provides higher code density than general-purpose ISAs such as MIPS (13.38 times), x86 (9.86 times), and GPGPU (6.41 times). Compared to the latest state-of-the-art NN accelerator design DaDianNao [5] (which can only accommodate 3 types of NN techniques), our Cambricon-based accelerator prototype implemented in TSMC 65nm technology incurs only negligible latency, power, and area overheads (4.5%/4.4%/1.6%, respectively), with a versatile coverage of 10 different NN benchmarks.

Our key contributions in this work are the following: 1) We propose a novel and lightweight ISA having strong descriptive capacity for NN techniques; 2) We conduct a comprehensive study on the computational patterns of existing NN techniques; 3) We evaluate the effectiveness of Cambricon with an implementation of the first Cambricon-based accelerator using TSMC 65nm technology.

The rest of the paper is organized as follows. Section 2 briefly discusses a few design guidelines followed by Cambricon and presents an overview to Cambricon. Section III introduces computational and logical instructions of Cambricon. Section IV presents a prototype Cambricon accelerator. Section V empirically evaluates Cambricon, and compares it against other ISAs. Section VI discusses the potential extension of Cambricon to broader techniques. Section VII presents the related work. Section VIII concludes the whole paper.

II. Overview of The Proposed ISA

In this section, we first describe the design guideline for our proposed ISA, and then a brief overview of the ISA.

A. Design Guidelines

To design a succinct, flexible, and efficient ISA for NNs, we analyze various NN techniques in terms of their computational operations and memory access patterns, based on which we propose a few design guidelines before make concrete design decisions.

- **Data-level Parallelism.** We observe that in most NN techniques that neuron and synapse data are organized as layers and then manipulated in a uniform/symmetric manner. When accommodating these operations, data-level parallelism enabled by vector/matrix instructions can be more efficient than instruction-level parallelism of traditional scalar instructions, and corresponds to higher code density. Therefore, *the focus of Cambricon would be data-level parallelism*.

- **Customized Vector/Matrix Instructions.** Although there are many linear algebra libraries (e.g., the BLAS library [9]) successfully covering a broad range of scientific computing applications, for NN techniques, fundamental operations defined in those algebra libraries are not necessarily effective and efficient choices (some are even redundant). More importantly, there are many common operations of NN techniques that are not covered by traditional linear algebra libraries. For example, the BLAS library does not support element-wise exponential computation of a vector, neither does it support random vector generation in synapse initialization, dropout [8] and Restricted Boltzmann Machine (RBM) [39]. Therefore, we must *comprehensively customize a small yet representative set of vector/matrix instructions for existing NN techniques*, instead of simply re-implementing vector/matrix operations from an existing linear algebra library.

- **Using On-chip Scratchpad Memory.** We observe that NN techniques often require intensive, contiguous, and variable-length accesses to vector/matrix data, and therefore using fixed-width power-hungry vector register files is no longer the most cost-effective choice. In our design, *we replace vector register files with on-chip scratchpad memory*, providing flexible width for each data access. This is usually a highly-efficient choice for data-level parallelism in NNs, because synapse data in NNs are often large and rarely reused, diminishing the performance gain brought by vector register files.

B. An Overview to Cambricon

We design the Cambricon following the guidelines p-resented in Section II-A, and provide an overview of the Cambricon in Table I. The Cambricon is a load-store architecture which only allows the main memory to be accessed

Table I. An overview to Cambricon instructions.

Instruction Type		Examples	Operands
Control		jump, conditional branch	register (scalar value), immediate
Data Transfer	Matrix	matrix load/store/move	register (matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register (scalar value), immediate
Computational	Matrix	matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix	register (matrix/vector address/size, s-calar value)
	Vector	vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions (exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector	register (vector address/size, scalar value)
	Scalar	scalar elementary arithmetics, scalar transcendental functions	register (scalar value), immediate
Logical	Vector	vector compare (greater than, equal), vector logical operations (and, or, inverter), vector greater than merge	register (vector address/size, scalar)
	Scalar	scalar compare, scalar logical operations	register (scalar), immediate

with load/store instructions. Cambricon contains 64 32-bit General-Purpose Registers (GPRs) for scalars, which can be used in register-indirect addressing of the on-chip scratchpad memory, as well as temporally keeping scalar data.

Type of Instructions. The Cambricon contains four types of instructions: *computational, logical, control*, and *data transfer* instructions. Although different instructions may differ in their numbers of valid bits, the instruction length is fixed to be 64-bit for the memory alignment and for the design simplicity of the load/store/decoding logic. In this section we only offer a brief introduction to the control and data transfer instructions because they are similar to their corresponding MIPS instructions, though have been adapted to fit NN techniques. For computational instructions (including matrix, vector and scalar instructions) and logical instructions, however, the details will be provided in the next section (Section III).

Control Instructions. The Cambricon has two control instructions, *jump* and *conditional branch*, as illustrated in Fig. 1. The jump instruction specifies the offset via either an immediate or a GPR value, which will be accumulated to the Program Counter (PC). The conditional branch instruction specifies the predictor (stored in a GPR) in addition to the offset, and the branch target (either $PC+\{offset\}$ or $PC+1$) is determined by a comparison between the predictor and zero.

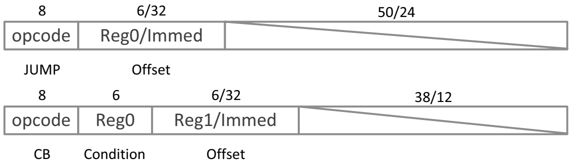

Figure 1. *top:* Jump instruction. *bottom:* Condition Branch (CB) instruction.

Data Transfer Instructions. Data transfer instructions in Cambricon support variable data size in order to flexibly support matrix and vector computational/logical instructions (see Section III for such instructions). Specifically, these instructions can load/store variable-size data blocks (specified by the data-width operand in data transfer instructions) from/to the main memory to/from the on-chip scratchpad memory, or move data between the on-chip scratchpad memory and scalar GPRs. Fig. 2 illustrates the Vector LOAD (VLOAD) instruction, which can load a vector with the size of V_{size} from the main memory to the vector scratchpad memory, where the source address in main memory is the sum of the base address saved in a GPR and an immediate number. The formats of Vector STORE (VSTORE), Matrix LOAD (MLOAD), and Matrix STORE (MSTORE) instructions are similar with that of VLOAD.

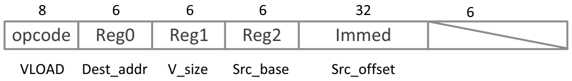

Figure 2. Vector Load (VLOAD) instruction.

On-chip Scratchpad Memory. Cambricon does not use any vector register file, but directly keeps data in on-chip scratchpad memory, which is made visible to programmers/compilers. In other words, the role of on-chip scratchpad memory in Cambricon is similar to that of vector register file in traditional ISAs, and sizes of vector operands are no longer limited by fixed-width vector register files. Therefore, vector/matrix sizes are variable in Cambricon instructions, and the only notable restriction is that the vector/matrix operands in the same instruction cannot exceed the capacity of scratchpad memory. In case they do exceed, the compiler will decompose long vectors/matrices into short pieces/blocks and generate multiple instructions to process them.

Just like the 32x512b vector registers have been baked into Intel AVX-512 [18], capacities of on-chip memories for both vector and matrix instructions must be fixed in Cambricon. More specifically, Cambricon fixes the memory capacity to be 64KB for vector instructions, 768KB for matrix instructions. Yet, Cambricon does not impose specific restriction on bank numbers of scratchpad memory, leaving significant freedom to microarchitecture-level implementations.

III. Computational/Logical Instructions

In neural networks, most arithmetic operations (e.g., additions, multiplications and activation functions) can be aggregated as vector operations [10], [45], and the ratio can be as high as 99.992% according to our quantitative observations on a state-of-the-art Convolutional Neural Network (GoogLeNet) winning the 2014 ImageNet competition (ILSVRC14) [43]. In the meantime, we also discover that 99.791% of the vector operations (such as dot product operation) in the GoogLeNet can be aggregated further as matrix operations (such as vector-matrix multiplication). In a nutshell, NNs can be naturally decomposed into scalar, vector, and matrix operations, and the ISA design must effectively take advantages of the potential data-level parallelism and data locality.

A. *Matrix Instructions*

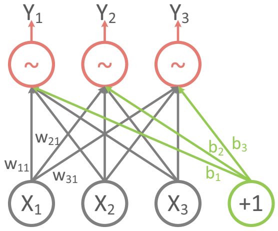

Figure 3. Typical operations in NNs.

We conduct a thorough and comprehensive review to existing NN techniques, and design a total of six matrix instructions for Cambricon. Here we take a Multi-Level Perceptrons (MLP) [50], a well-known and representative NN, as an example, and show how it is supported by the matrix instructions. Technically, an MLP usually has multiple layers, each of which computes values of some neurons (i.e., output neurons) according to some neurons whose values are known (i.e., input neurons). We illustrate the feedforward run of one such layer in Fig. 3. More specifically, the output neuron y_i $(i = 1,2,3)$ in Fig. 3 can be computed as $y_i = f\left(\sum_{j=1}^{3} w_{ij}x_j + b_i\right)$, where x_j is the j-th input neuron, w_{ij} is the weight between the i-th output neuron and the j-th input neuron, b_i is the bias of the i-th output neuron, and f is the activation function. The output neurons can be computed as a vector $\mathbf{y} = (y_1, y_2, y_3)$:

$$\mathbf{y} = \mathbf{f}(W\mathbf{x} + \mathbf{b}), \qquad (1)$$

where $\mathbf{x} = (x_1, x_2, x_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ are vectors of input neurons and biases, respectively, $W = (w_{ij})$ is the weight matrix, and $\mathbf{f}$ is the element-wise version of the activation function f (see Section III-B).

A critical step in Eq. 1 is to compute $W\mathbf{x}$, which will be performed by the Matrix-Mult-Vector (MMV) instruction in Cambricon. We illustrate this instruction in Fig. 4, where $Reg0$ specifies the base scratchpad memory address of the vector output ($Vout_{addr}$); $Reg1$ specifies the size of the vector output ($Vout_{size}$); $Reg2$, $Reg3$, and $Reg4$ specify the base address of the matrix input (Min_{addr}), the base address of the vector input (Vin_{addr}), and the size of the vector input (Vin_{size}, note that it is variable), respectively. The MMV instruction can support matrix-vector multiplication at arbitrary scales, as long as all the input and output data can be kept simultaneously in the scratchpad memory. We choose to compute $W\mathbf{x}$ with the dedicated MMV instruction instead of decomposing it as multiple vector dot products, because the latter approach requires additional efforts (e.g., explicit synchronization, concurrent read/write requests to the same address) to reuse the input vector $\mathbf{x}$ among different row vectors of M, which is less efficient.

8	6	6	6	6	6	26
opcode	Reg0	Reg1	Reg2	Reg3	Reg4	
MMV	Vout_addr	Vout_size	Min_addr	Vin_addr	Vin_size	

Figure 4. Matrix Mult Vector (MMV) instruction.

Unlike the feedforward case, however, the MMV instruction no longer provides efficient support to the backforward training process of an NN. More specifically, a critical step of the well-known Back-Propagation (BP) algorithm is to compute the gradient vector [20], which can be formulated as a vector multiplied by a matrix. If we implement it with the MMV instruction, we need an additional instruction implementing matrix transpose, which is rather expensive in data movements. To avoid that, Cambricon provides a Vector-Mult-Matrix (VMM) instruction which is directly applicable to the backforward training process. The VMM instruction has the same fields with the MMV instruction, except the opcode.

Moreover, in training an NN, the weight matrix W often needs to be incrementally updated with $W = W + \eta\Delta W$, where η is the learning rate and ΔW is estimated as the outer product of two vectors. Cambricon provides an Outer-Product (OP) instruction (the output is a matrix), a Matrix-Mult-Scalar (MMS) instruction, and a Matrix-Add-Matrix (MAM) instruction to collaboratively perform the weight updating. In addition, Cambricon also provides a Matrix-Subtract-Matrix (MSM) instruction to support the weight updating in Restricted Boltzmann Machine (RBM) [39].

B. *Vector Instructions*

Using Eq. 1 as an example, one can observe that the matrix instructions defined in the prior subsection are still insufficient to perform all the computations. We still need to add up the vector output of $W\mathbf{x}$ and the bias vector $\mathbf{b}$, and then perform an element-wise activation to $W\mathbf{x} + \mathbf{b}$.

While Cambricon directly provides a Vector-Add-Vector (VAV) instruction for vector additions, it requires multiple instructions to support the element-wise activation. Without losing any generality, here we take the widely-used sigmoid activation, $f(a) = e^a/(1+e^a)$, as an example. The element-wise sigmoid activation performed to each element of an input vector (say, **a**) can be decomposed into 3 consecutive steps, and are supported by 3 instructions, respectively:

1. Computing the exponential e^{a_i} for each element ($a_i, i = 1, \dots, n$) in the input vector **a**. Cambricon provides a Vector-Exponential (VEXP) instruction for element-wise exponential of a vector.
2. Adding the constant 1 to each element of the vector $(e^{a_1}, \dots, e^{a_n})$. Cambricon provides a Vector-Add-Scalar (VAS) instruction, where the scalar can be an immediate or specified by a GPR.
3. Dividing e^{a_i} by $1 + e^{a_i}$ for each vector index $i = 1, \dots, n$. Cambricon provides a Vector-Div-Vector (VDV) instruction for element-wise division between vectors.

However, the sigmoid is not the only activation function utilized by the existing NNs. To implement element-wise versions of various activation functions, Cambricon provides a series of vector arithmetic instructions, such as Vector-Mult-Vector (VMV), Vector-Sub-Vector (VSV), and Vector-Logarithm (VLOG). During the design of a hardware accelerator, instructions related to different transcendental functions (e.g. logarithmic, trigonometric and anti-trigonometric functions) can efficiently reuse the same functional block (involving addition, shift, and table-lookup operations), using the CORDIC technique [24]. Moreover, there are activation functions (e.g, $\max(0, a)$ and $|a|$) that partially rely on logical operations (e.g., comparison), and we will present the related Cambricon instructions (e.g., vector compare instructions) in Section III-C.

Furthermore, the random vector generation is an important operation common in many NN techniques (e.g., dropout [8] and random sampling [39]), but is not deemed as a necessity in traditional linear algebra libraries designed for scientific computing (e.g., the BLAS library does not include this operation). Cambricon provides a dedicated instruction (Random-Vector, RV) that generates a vector of random numbers obeying the uniform distribution at the interval $[0, 1]$. Given uniform random vectors, we can further generate random vectors obeying other distributions (e.g., Gaussian distribution) using the Ziggurat algorithm [31], with the help of vector arithmetic instructions and vector compare instructions in Cambricon.

C. Logical Instructions

The state-of-the-art NN techniques leverage a few operations that incorporate comparisons or other logical manipulations. The max-pooling operation is one such operation (see Fig. 5a for an illustration), which seeks the neuron having the largest output among neurons within a pooling window, and repeats this action for corresponding pooling windows in different input feature maps (see Fig. 5b). Cambricon supports the max-pooling operation with a Vector-Greater-Than-Merge (VGTM) instruction, see Fig. 6. The VGTM instruction designates each element of the output vector ($Vout$) by comparing corresponding elements of the input vector-0 ($Vin0$) and input vector-1 ($Vin1$), i.e., $Vout[i] = (Vin0[i] > Vin1[i])?Vin0[i] : Vin1[i]$. We present the Cambricon code of the max-pooling operation in Section III-E, which aggregates neurons at the same position of all input feature maps in the same input vector, iteratively performs VGTM and obtains the final result (see also Fig. 5c for an illustration).

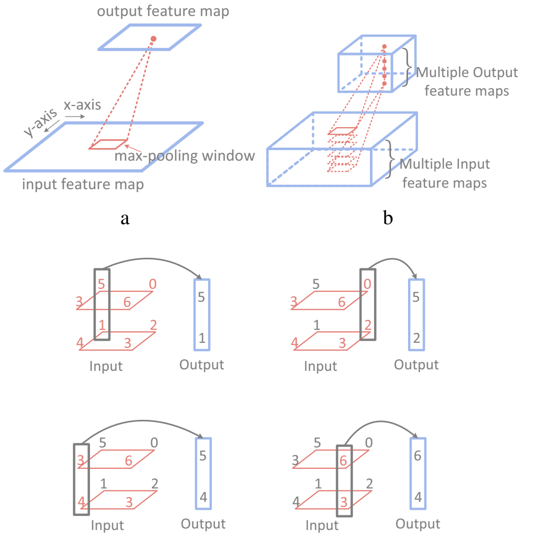

Figure 5. Max-pooling operation.

In addition to the vector computational instruction, Cambricon also provides Vector-Greater-than (VGT), Vector-Equal instruction (VE), Vector AND/OR/NOT instructions (VAND/VOR/VNOT), scalar comparison, and scalar logical instructions to tackle branch conditions, i.e., computing the predictor for the aforementioned Conditional Branch (CB) instruction.

8	6	6	6	6	32
opcode	Reg0	Reg1	Reg2	Reg3	
VGTM	Vout_addr	Vout_size	Vin0_addr	Vin1_addr	

Figure 6. Vector Greater Than Merge (VGTM) instruction.

D. Scalar Instructions

Although we have observed that only 0.008% arithmetic operations of the GoogLeNet [43] cannot be supported with matrix and vector instructions in Cambricon, there are also scalar operations that are indispensable to NNs, such as elementary arithmetic operations and scalar transcendental functions. We summarize them in Table I, which have been formally defined as Cambricon's scalar instructions.

E. Code Examples

To illustrate the usage of our proposed instruction sets, we implement three simple yet representative components of NNs, a MLP feedforward layer [50], a pooling layer [22],

```
MLP code:
    // $0: input size, $1: output size, $2: matrix size
    //  $3: input address, $4: weight address
    //  $5: bias address, $6: output address
    //  $7-$10: temp variable address

  VLOAD    $3, $0, #100          // load input vector from address (100)
  MLOAD    $4, $2, #300          // load weight matrix from address (300)
  MMV      $7, $1, $4, $3, $0    // Wx
  VAV      $8, $1, $7, $5        // tmp=Wx+b
  VEXP     $9, $1, $8            // exp(tmp)
  VAS      $10, $1, $9, #1       // 1+exp(tmp)
  VDV      $6, $1, $9, $10       // y=exp(tmp)/(1+exp(tmp))
  VSTORE   $6, $1, #200          // store output vector to address (200)
Pooling code:
    // $0: feature map size, $1: input data size,
    // $2: output data size, $3: pooling window size  -  1
    // $4: x-axis loop num, $5: y-axis loop num
    // $6: input addr, $7: output addr
    // $8: y-axis stride of input

      VLOAD     $6, $1, #100      //  load input neurons from address (100)
      SMOVE     $5, $3            //  init y
  L0: SMOVE     $4, $3            //  init x
  L1: VGTM      $7, $0, $6, $7
    // ∀ feature map m, output[m]=(input[x][y][m]>output[m])?
    //                            input[x][y][m]:output[m]
      SADD      $6, $6, $0        //  update input address
      SADD      $4, $4, #-1       //  x--
      CB        #L1, $4           //  if(x>0) goto L1
      SADD      $6, $6, $8        //  update input address
      SADD      $5, $5, #-1       //  y--
      CB        #L0, $5           //  if(y>0) goto L0
      VSTORE    $7, $2, #200      //  stroe output neurons to address (200)
BM code:
    // $0: visible vector size, $1: hidden vector size, $2: v-h matrix (W) size
    //  $3: h-h matrix (L) size, $4: visible vector address, $5: W address
    //  $6: L address, $7: bias address, $8: hidden vector address
    //  $9-$17: temp variable address

  VLOAD    $4, $0, #100          // load visible vector from address (100)
  VLOAD    $9, $1, #200          // load hidden vector from address (200)
  MLOAD    $5, $2, #300          // load W matrix from address (300)
  MLOAD    $6, $3, #400          // load L matrix from address (400)
  MMV      $10, $1, $5, $4, $0   // Wv
  MMV      $11, $1, $6, $9, $1   // Lh
  VAV      $12, $1, $10, $11     // Wv+Lh
  VAV      $13, $1, $12, $7      // tmp=Wv+Lh+b
  VEXP     $14, $1, $13          // exp(tmp)
  VAS      $15, $1, $14, #1      // 1+exp(tmp)
  VDV      $16, $1, $14, $15     // y=exp(tmp)/(1+exp(tmp))
  RV       $17, $1               // ∀i, r[i] = random(0,1)
  VGT      $8, $1, $17, $16      // ∀i, h[i] = (r[i]>y[i])?1:0
  VSTORE   $8, $1, #500          // store hidden vector to address (500)
```

Figure 7. Cambricon program fragments of MLP, pooling and BM.

and a Boltzmann Machines (BM) layer [39], using Cambricon instructions. For the sake of brevity, we omit scalar load/store instructions for all three layers, and only show the program fragment of a single pooling window (with multiple input and output feature maps) for the pooling layer. We illustrate the concrete Cambricon program fragments in Fig. 7, and we observe that the code density of Cambricon is significantly higher than that of x86 and MIPS (see Section V for a comprehensive evaluation).

IV. A Prototype Accelerator

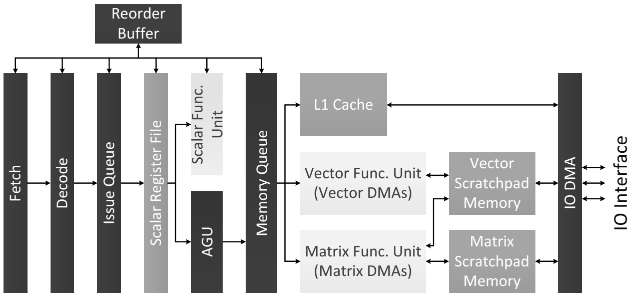

Figure 8. A prototype accelerator based on Cambricon.

In this section, we present a prototype accelerator of Cambricon. We illustrate the design in Fig. 8, which contains seven major instruction pipeline stages: *fetching, decoding, issuing, register reading, execution, writing back, and committing*. We use mature techniques such as scratchpad memory and DMA in this accelerator, since we found that these classic techniques have been sufficient to reflect the flexibility (Section V-B1), conciseness (Section V-B2) and efficiency (Section V-B3) of the ISA. We did not seek to explore the emerging techniques (such as 3D stacking [51] and non-volatile memory [47], [46]) in our prototype design,but left such exploration as future work, because we believe that a promising ISA must be easy to implement and should not be tightly coupled with emerging techniques.

As illustrated in Fig. 8, after the fetching and decoding stages, an instruction is injected into an in-order issue queue. After successfully fetching the operands (scalar data, or address/size of vector/matrix data) from the scalar register file, an instruction will be sent to different units depending on the instruction type. Control instructions and scalar computational/logical instructions will be sent to the scalar functional unit for direct execution. After writing back to the scalar register file, such an instruction can be committed from the reorder buffer[1] as long as it has become the oldest uncommitted yet executed instruction.

Data transfer instructions, vector/matrix computational instructions, and vector logical instructions, which may access the L1 cache or scratchpad memories, will be sent to the Address Generation Unit (AGU). Such an instruction needs to wait in an in-order memory queue to resolve potential memory dependencies[2] with earlier instructions in the memory queue. After that, load/store requests of scalar data transfer instructions will be sent to the L1 cache, data transfer/computational/logical instructions for vectors will be sent to the vector functional unit, data transfer/computational instructions for matrices will be sent to matrix functional unit. After the execution, such an

[1] We need a reorder buffer even though instructions are in-order issued, because the execution stages of different instructions may take significantly different numbers of cycles.

[2] Here we say two instructions are memory dependent if they access an overlapping memory region, and at least one of them needs to write the memory region.

instruction can be retired from the memory queue, and then be committed from the reorder buffer as long as it has become the oldest uncommitted yet executed instruction.

The accelerator implements both vector and matrix functional units. The vector unit contains 32 16-bit adders, 32 16-bit multipliers, and is equipped with a 64KB scratchpad memory. The matrix unit contains 1024 multipliers and 1024 adders, which has been divided into 32 separate computational blocks to avoid excessive wire congestion and power consumption on long-distance data movements. Each computational block is equipped with a separate 24KB scratchpad. The 32 computational blocks are connected through an h-tree bus that serves to broadcast input values to each block and to collect output values from each block.

A notable Cambricon feature is that it does not use any vector register file, but keeps data in on-chip scratchpad memories. To efficiently access scratchpad memories, the vector/matrix functional unit of the prototype accelerator integrates three DMAs, each of which corresponds to one vector/matrix input/output of an instruction. In addition, the scratchpad memory is equipped with an IO DMA. However, each scratchpad memory itself only provides a single port for each bank, but may need to address up to four concurrent read/write requests. We design a specific structure for the scratchpad memory to tackle this issue (see Fig. 9). Concretely, we decompose the memory into four banks according to addresses' low-order two bits, connect them with four read/write ports via a crossbar guaranteeing that no bank will be simultaneously accessed. Thanks to the dedicated hardware support, Cambricon does not need expensive multi-port vector register file, and can flexibly and efficiently support different data widths using the on-chip scratchpad memory.

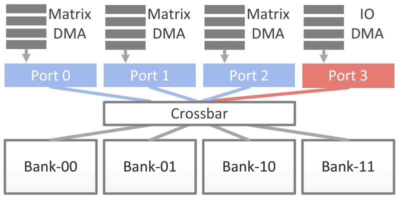

Figure 9. Structure of matrix scratchpad memory.

V. Experimental Evaluation

In this section, we first describe the evaluation methodology, and then present the experimental results.

A. Methodology

Design evaluation. We synthesize the prototype accelerator of Cambricon (Cambricon-ACC, see Section IV) with Synopsys Design Compiler using TSMC 65nm GP standard VT library, place and route the synthesized design with the Synopsys ICC compiler, simulate and verify it with Synopsys VCS, and estimate the power consumption with Synopsys Prime-Time PX according to the simulated Value Change Dump (VCD) file. We are planning an MPW tape-out of the prototype accelerator, with a small area budget of 60 mm^2 at a 65nm process with targeted operating frequency of 1 Ghz. Therefore, we adopt moderate functional unit sizes and scratchpad memory capacities in order to fit the area budget. II shows the details of design parameters.

Table II. Parameters of our prototype accelerator.

issue width	2
depth of issue queue	24
depth of memory queue	32
depth of reorder buffer	64
capacity of vector scratchpad memory	64KB
capacity of matrix scratchpad memory	768KB (24KB x 32)
bank width of scratchpad memory	512 bits (32 x 16-bit fixed point)
operators in matrix function unit	1024 (32x32) multipliers & adders
operators in vector function unit	32 multipliers & dividers & adders & transcendental function operators

Baselines. We compare the Cambricon-ACC with three baselines. The first two are based on general-purpose CPU and GPU, and the last one is a state-of-the-art NN hardware accelerator:

- *CPU.* The CPU baseline is an x86-CPU with 256-bit SIMD support (Intel Xeon E5-2620, 2.10GHz, 64 GB memory). We use the Intel MKL library [19] to implement vector and matrix primitives for the CPU baseline, and GCC v4.7.2 to compile all benchmarks with options "-O2 -lm -march=native" to enable SIMD instructions.
- *GPU.* The GPU baseline is a modern GPU card (NVIDIA K40M, 12GB GDDR5, 4.29 TFlops peak at a 28nm process); we implement all benchmarks (see below) with the NVIDIA cuBLAS library [35], a state-of-the-art linear algebra library for GPU.
- *NN Accelerator.* The baseline accelerator is DaDianNao, a state-of-the-art NN accelerator exhibiting remarkable energy-efficiency improvement over a GPU [5]. We re-implement the DaDianNao architecture at a 65nm process, but replace all eDRAMs with SRAMs because we do not have a 65nm eDRAM library. In addition, we re-size DaDianNao such that it has a comparable amount of arithmetic operators and on-chip SRAM capacity as our design, which enables a fair comparison of two accelerators under our area budget (<60 mm^2) mentioned in the previous paragraph. The re-implemented version of DaDianNao has a single central tile and a total of 32 leaf tiles. The central tile has 64KB SRAM, 32 16-bit adders and 32 16-bit multipliers; Each leaf tile has 24KB SRAM, 32 16-bit adders and 32 16-bit multipliers. In other words, the total numbers of adders and multipliers, as well as the total SRAM capacity in the re-implemented DaDianNao, are the same with our prototype accelerator. Although we are constrained to give up eDRAMs in both accelerators, this is still a fair and

reasonable experimental setting, because the flexibility of an accelerator is mainly determined by its ISA, not concrete devices it integrates. In this sense, the flexibility gained from Cambricon will still be there even when we resort to large eDRAMs to remove main memory accesses and improve the performance for both accelerators.

Benchmarks. We take 10 representative NN techniques as our benchmarks, see Table III. Each benchmark is translated manually into assemblers to execute on Cambricon-ACC and DaDianNao. We evaluate their cycle-level performance with Synopsys VCS.

B. Experimental Results

We compare Cambricon and Cambricon-ACC with the baselines in terms of metrics such as performance and energy. We also provide the detailed layout characteristics of the prototype accelerator.

1) Flexibility: In view of the apparent flexibility provided by general-purpose ISAs (e.g., x86, MIPS and GPU-ISA), here we restrict our discussions to ISAs of NN accelerators. DaDianNao [5] and DianNao [3] are the two unique NN accelerators that have explicit ISAs (other ones are often hardwired). They share similar ISAs, and our discussion is exemplified by DaDianNao, the one with better performance and multicore scaling. To be specific, the ISA of this accelerator only contains four 512-bit VLIW instructions corresponding to four popular layer types of neural networks (fully-connected classifier layer, convolutional layer, pooling layer, and local response normalization layer), rendering it a rather incomplete ISA for the NN domain. Among 10 representative benchmark networks listed in Table III, the DaDianNao ISA is only capable of expressing MLP, CNN, and RBM, but fails to implement the rest 7 benchmarks (RNN, LSTM, AutoEncoder, Sparse AutoEncoder, BM, SOM and HNN). An observation well explaining the failure of DaDianNao on the 7 representative networks is that they cannot be characterized as aggregations of the four types of layers (thus aggregations of DaDianNao instructions). In contrast, Cambricon defines a total of 43 64-bit scalar/control/vector/matrix instructions, and is sufficiently flexible to express all 10 networks.

2) Code Density: Code density is a meaningful ISA metric only when the ISA is flexible enough to cover a broad range of applications in the target domain. Therefore, we only compare the code density of Cambricon with GPU, MIPS, and x86, with 10 benchmarks implemented with Cambricon, CUDA-C, and C, respectively. We manually write the Cambricon program; We compile the CUDA-C programs with nvcc, and count the length of the generated ptx files after removing initialization and system-call instructions; We compile the C programs with x86 and MIPS compilers, respectively (with the option -O2). We then count the lengths of two kinds of assemblers. We illustrate in Fig. 10 Cambricon's reduction on code length over other ISAs. On average, the code length of Cambricon is about 6.41x, 9.86x, and 13.38x shorter than GPU, x86, and MIPS, respectively. The observations are not surprising, because Cambricon aggregates many scalar operations into vector instructions, and further aggregates vector operations into matrix instructions, which significantly reduces the code length.

Specifically, on MLP, Cambricon can improve the code density by 13.62x, 22.62x, and 32.92x against GPU, x86, and MIPS, respectively. The main reason is that there are very few scalar instructions in the Cambricon code of MLP. However, on CNN, Cambricon achieves only 1.09x, 5.90x, and 8.27x reduction of code length against GPU, x86, and MIPS, respectively. It is because that the main body of CNN is a deeply nested loop requiring many individual scalar operations to manipulate the loop variable. Hence, the advantage of aggregating scalar operations into vector operations has a small gain on code density.

Moreover, we collect the percentage breakdown of Cambricon instruction types in the 10 benchmarks. On average, 38.0% instructions are data transfer instructions, 4.8% instructions are control instructions, 12.6% instructions are matrix instructions, 33.8% instructions are vector instructions, and 10.9 % instructions are scalar instructions. This observation clearly shows that vector/matrix instructions play a critical role in NN techniques, thus efficient implementations of these instructions are essential to the performance of an Cambricon-based accelerator.

3) Performance: We compare Cambricon-ACC against x86-CPU and GPU on all 10 benchmarks listed in Table III. Fig. 12 illustrates the speedup of Cambricon-ACC against x86-CPU, GPU, and DaDianNao. On average, Cambricon-ACC is about 91.72x and 3.09x faster than of x86-CPU and GPU, respectively. This is not surprising because Cambricon-ACC integrates dedicated functional units and scratchpad memory optimized for NN techniques.

On the other hand, due to the incomplete and restricted ISA, DaDianNao can only accommodate 3 out of 10 benchmarks (i.e., MLP, CNN and RBM), thus its flexibility is significantly worse than that of Cambricon-ACC. In the meantime, the better flexibility of Cambricon-ACC does not lead to significant performance loss. We compare Cambricon-ACC against DaDianNao on the three benchmarks that DaDianNao can support, and observe that Cambricon-ACC is only 4.5% slower than DaDianNao on average. The reason for a small performance loss of Cambricon-ACC over DaDianNao is that, Cambricon decomposes complex high-level functional instructions of DaDianNao (e.g., an instruction for a convolutional layer) into shorter and low-level computational instructions (e.g., MMV and dot product), which may bring in additional pipeline bubbles between instructions. With the high code density provided by Cambricon, however, the amount of additional bubbles is moderate, the corresponding performance loss is therefore negligible.

Table III. Benchmarks (H stands for hidden layer, C stands for convolutional layer, K stands for kernel, P stands for pooling layer, F stands for classifier layer, V stands for visible layer).

Technique	Network Structure	Description
MLP	input(64) - H1(150) - H2(150) - Output(14)	Using Multi-Layer Perceptron (MLP) to perform anchorperson detection. [2]
CNN	input(1@32x32) - C1(6@28x28, K: 6@5x5) - S1(6@14x14, K: 2x2) - C2(16@10x10, K: 16@5x5) - S2(16@5x5, K: 2x2) - F(120) - F(84) - output(10)	Convolutional neural network (LeNet-5) for hand-written character recognition. [28]
RNN	input(26) - H(93) - output(61)	Recurrent neural network (RNN) on TIMIT database. [15]
LSTM	input(26) - H(93) - output(61)	Long-short-time-memory (LSTM) neural network on TIMIT database. [15]
Autoencoder	input(320) - H1(200) - H2(100) - H3(50) - Output(10)	A neural network pretrained by auto-encoder on MNIST data set. [49]
Sparse Autoencoder	input(320) - H1(200) - H2(100) - H3(50) - Output(10)	A neural network pretrained by sparse auto-encoder on MNIST data set. [49]
BM	V(500) - H(500)	Boltzmann machines (BM) on MINST data set. [39]
RBM	V(500) - H(500)	Restricted boltzmann machine (RBM) on MINST data set. [39]
SOM	input data(64) - neurons(36)	Self-organizing maps (SOM) based data mining of seasonal flu. [48]
HNN	vector (5), vector component(100)	Hopfield neural network (HNN) on hand-written digits data set. [36]

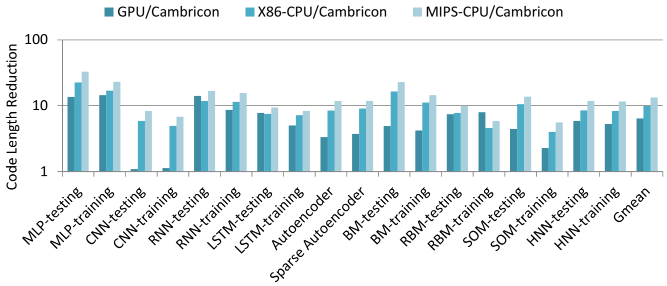

Figure 10. The reduction of code length against GPU, x86-CPU, and MIPS-CPU.

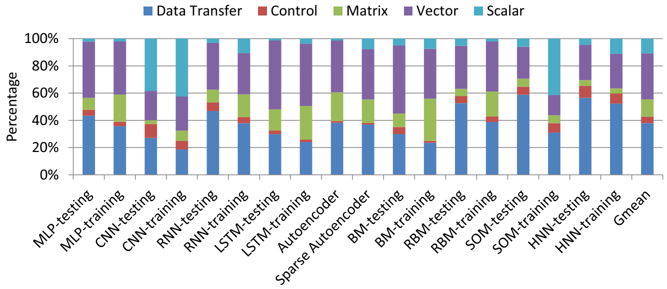

Figure 11. The percentages of instruction types among all benchmarks.

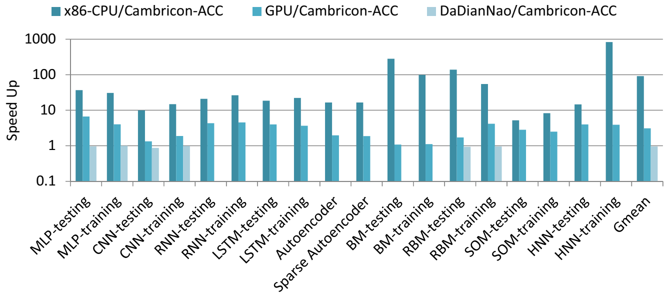

Figure 12. The speedup of Cambricon-ACC against x86-CPU, GPU, and DaDianNao.

4) Energy Consumption: We also compare the energy consumptions of Cambricon-ACC, GPU and DaDianNao, which can be estimated as products of power consumptions (in Watt) and the execution times (in Second). The power consumption of GPU is reported by the NVPROF, and the power consumptions of DaDianNao and Cambricon-ACC are estimated with Synopsys Prime-Tame PX according to the simulated Value Change Dump (VCD) file. We do not have the energy comparison against CPU baseline, because of the lack of hardware support for the estimation of the actual power of the CPU. Yet, recently it has been reported that an SIMD-CPU is an order-of-magnitude less energy-efficient than a GPU (NVIDIA K20M) on neural network applications [4], which well complements our experiments.

As shown in Fig. 13, the energy consumptions of GPU and DaDianNao are 130.53x and 0.916x that of Cambricon-ACC, respectively, where the energy of DaDianNao is averaged over 3 benchmarks because it can only accommodate 3 out of 10 benchmarks. Compared with Cambricon-ACC, the power consumption of GPU is much higher, as the GPU spends excessive hardware resources to flexibly support various workloads. On the other hand, the energy consumption of Cambricon-ACC is only slightly higher than of DaDianNao, because both accelerators integrate the same sizes of functional units and on-chip storage, and work at the same frequency. The additional energy consumed by Cambricon-ACC mainly comes from instruction pipeline logic, memory queue, as well as the vector transcendental functional unit. In contrast, DaDianNao uses a low-precision but lightweight lookup table instead of using transcendental functional units.

5) Chip Layout: We show the layout of Cambricon-ACC in Fig. 14, and list the area and power breakdowns in Table IV. The overall area of Cambricon-ACC is 56.24 mm^2, which is about 1.6% larger than of DaDianNao (55.34 mm^2, re-implemented version). The combinational logic (mainly vector and matrix functional units) consumes 32.15% area of Cambricon-ACC, and the on-chip memory (mainly vector and matrix scratchpad memories) consumes about 15.05% area.

The matrix part (including the matrix function unit and the matrix scratchpad memory) accounts for 62.69% area of Cambricon-ACC, while the core part (including the instruction pipeline logic, scalar function unit, memory queue, and so on) and the vector part (including the vector function unit and the vector scratchpad memory) only account for 9.00 % area. The remaining 28.31% area is consumed by the channel part, including wires connecting the core & vector part and the matrix part, and wires connecting together different blocks of the matrix part.

We also estimate the power consumption of the prototype design with Synopsys PrimePower. The peak power consumption is 1.695 W (under 100% toggle rate), which is only about one percentage of the K40M GPU. More specifically, the core & vector part and matrix part consume 8.20%, and 59.26% power, respectively. Moreover, data movements in the channel part consume 32.54% power, which is several times higher than the power of the core & vector part. It can be expected that the power consumption of the channel part can be much higher if we do not divide the matrix part into multiple blocks.

Table IV. Layout characteristics of Cambricon-ACC (1 GHz), implemented in TSMC 65nm technology.

Component	**Area**(μm^2)	**(%)**	**Power**(mW)	**(%)**
Whole Chip	56241000	100%	1695.60	100%
Core & Vector	5062500	9.00%	139.04	8.20%
Matrix	35259840	62.69%	1004.81	59.26%
Chanel	15918660	28.31%	551.75	32.54%
Combinational	18081482	32.15%	476.97	28.13%
Memory	8461445	15.05%	174.14	10.27%
Registers	5612851	9.98%	300.29	17.71%
Clock network	877360	1.56%	744.20	43.89%
Filler Cell	23207862	41.26%		

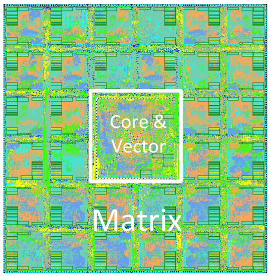

Figure 14. The layout of Cambricon-ACC, implemented in TSMC 65nm technology.

VI. Potential Extension to Broader Techniques

Although Cambricon is designed for existing neural network techniques, it can also support future neural network techniques or even some classic statistical techniques, as long as they can be decomposed into scalar/ector/matrix instructions in Cambricon. Here we take *logistic regression* [21] as an example, and illustrate how it can be supported by Cambricon. Technically, logistic regression contains two phases, training phase, and prediction phase. The training phase employs a gradient descent algorithm similar to the training phase of MLP technique, which can be supported by Cambricon. In the prediction phase, the output can be computed as $y = sigmoid\left(\sum_{i=0}^{d}\theta_i x_i\right)$ (where $x = (x_0, x_1 ... x_n)^T$ is the input vector, x_0 always equals to 1, $\theta = (\theta_0, \theta_1 ... \theta_n)^T$ is the model parameters). We can leverage the dot product instruction, scalar elementary arithmetic instructions, and scalar exponential instruction of Cambricon to perform the prediction phase of logistic regression. Moreover, given a batch of n different input vectors, the MMV instruction, vector elementary arithmetic instructions and vector exponential instruction in Cambricon collaboratively allow prediction phases of n inputs to be computed in parallel.

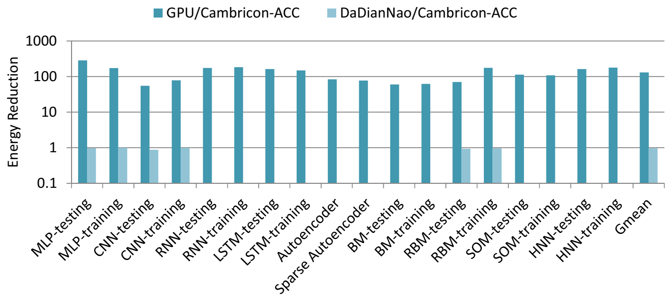

Figure 13. The energy reduction of Cambricon-ACC over GPU and DaDianNao.

VII. Related Work

In this section, we summarize prior work on NN techniques and NN accelerator designs.

Neural Networks. Existing NN techniques have exhibited significant diversity in their network topologies and learning algorithms. For example, Deep Belief Networks (DBNs) [41] consist of a sequence of layers, each of which is fully connected to its adjacent layers. In contrast, Convolutional Neural Networks (CNNs) [25] use convolutional/pooling windows to specify connections between neurons, thus the connection density is much lower than in DBNs. Interestingly, connection densities of DBNs and CNNs are both lower than the Boltzmann Machines (BMs) [39] that fully connect all neurons with each other. Learning algorithms for different NNs may also differ from each other, as exemplified by the remarkable discrepancy among the back-propagation algorithm for training Multi-Level Perceptrons (MLPs) [50], the Gibbs sampling algorithm for training Restricted Boltzmann Machines (RBMs) [39], and the unsupervised learning algorithm for training Self-Organizing Map (SOM) [34].

In a nutshell, while adopting high-level, complex, and informative instructions could be a feasible choice for accelerators supporting a small set of similar NN techniques, the significant diversity and the large number of existing NN techniques make it unfeasible to build a single accelerator that uses a considerable number of high-level instructions to cover a broad range of NNs. Moreover, without a certain degree of generality, even an exisiting successful accelerator design may easily become inapplicable simply because of the evolution of NN techniques.

NN Accelerators. NN techniques are computationally intensive, and are traditionally executed on general-purpose platforms composed of CPUs and GPGPUs, which are usually not energy-efficient for NN techniques [3], because they invest excessive hardware resources to flexibly support various workloads. Over the past decade, there have been many hardware accelerators customized to NNs, implemented on FPGAs [13], [38], [40], [42] or as ASICs [3], [12], [14], [44]. Farabet *et al.* proposed an accelerator named Neuflow with systolic architecture [12], for the feed-forward paths of CNNs. Maashri *et al.* implemented another NN accelerator, which arranges several customized accelerators around a switch fabric [30]. Esmaeilzadeh *et al.* proposed a SIMD-like architecture (NnSP) for Multi-Layer Perceptrons (MLPs) [10]. Chakradhar *et al.* mapped the CNN to reconfigurable circuits [1]. Chi *et al.* proposed PRIME [6], a novel process-in-memory architecture that implements reconfigurable NN accelerator in ReRAM-based main memory. Hashmi *et al.* proposed the Aivo framework to characterize their specific cortical network model and learning algorithms, which can generate execution code of their network model for general-purpose CPUs and GPUs rather than hardware accelerators [16]. The above designs were customized for one specific NN technique (e.g., MLP or CNN), whose application scopes are limited. Chen *et al.* proposed a small-footprint NN accelerator called DianNao, whose instructions directly correspond to different layer types in CNN [3]. DaDianNao adopts a similar instruction set, but achieves even higher performance and energy-efficiency via keeping all network parameters on-chip, which is a piece of innovation on accelerator architecture instead of ISA [5]. Therefore, the application scope of DaDianNao is still limited by its ISA, which is similar to the case of DianNao. Liu *et al.* designed the PuDianNao accelerator that accommodates seven classic machine learning techniques, whose control module only provides seven different opcodes (each corresponds to a specific machine learning technique) [29]. Therefore, PuDianNao only allows minor changes to the seven machine learning techniques. In summary, the lack of agility in instruction sets prevents previous accelerators from flexibly and efficiently supporting a variety of different NN techniques.

Comparison. Compared to prior work, we decompose traditional high-level and complex instructions describing high-level functional blocks of NNs (e.g., layers) into shorter instructions corresponding to low-level computational operations (e.g., scalar/vector/matrix operations), which allows a hardware accelerator to have a broader application scope. Furthermore, simple and short instructions may reduce the design and verification complexity of the accelerators.

VIII. Conclusion and Future Work

In this paper, we propose a novel ISA for neural networks called Cambricon, which allows NN accelerators to flexibly support a broad range of different NN techniques. We compare Cambricon with x86 and MIPS across ten diverse yet representative NNs, and observe that the code density of Cambricon is significantly higher than that of x86 and MIPS. We implement a Cambricon-based prototype accelerator in TSMC 65nm technology, and the area is 56.24 mm^2, the power consumption is only 1.695 W. Thanks to Cambricon, this prototype accelerator can accommodate all ten benchmark NNs, while the state-of-the-art NN accelerator, DaDianNao, can only support 3 of them. Even when executing the 3 benchmark NNs, our prototype accelerator still achieves comparable performance/energy-efficiency with the state-of-the-art accelerator with negligible overheads. Our future work includes the final chip tape-out of the prototype accelerator, an attempt to integrate Cambricon into a general-purpose processor, as well as an in-depth study that extends Cambricon to support broader applications.

Acknowledgment

This work is partially supported by the NSF of China (under Grants 61133004, 61303158, 61432016, 61472396, 61473275, 61522211, 61532016, 61521092, 61502446), the 973 Program of China (under Grant 2015CB358800), the S-trategic Priority Research Program of the CAS (under Grants XDA06010403, XDB02040009), the International Collaboration Key Program of the CAS (under Grant 171111KYS-B20130002), and the 10000 talent program. Xie is supported in part by NSF 1461698, 1500848, and 1533933.

References

[1] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A Dynamically Configurable Coprocessor for Convolutional Neural Networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[2] Yun-Fan Chang, P. Lin, Shao-Hua Cheng, Kai-Hsuan Chan, Yi-Chong Zeng, Chia-Wei Liao, Wen-Tsung Chang, Yu-Chiang Wang, and Yu Tsao. Robust anchorperson detection based on audio streams using a hybrid I-vector and DNN system. In *Proceedings of the 2014 Annual Summit and Conference on Asia-Pacific Signal and Information Processing Association*, 2014.

[3] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. A High-Throughput Neural Network Accelerator. *IEEE Micro*, 2015.

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[7] 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*, 2013.

[8] G.E. Dahl, T.N. Sainath, and G.E. Hinton. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[9] V. Eijkhout. Introduction to High Performance Scientific Computing. In *www.lulu.com*, 2011.

[10] H. Esmaeilzadeh, P. Saeedi, B.N. Araabi, C. Lucas, and Sied Mehdi Fakhraie. Neural network stream processing core (NnSP) for embedded systems. In *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems*, 2006.

[11] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 IEEE/ACM International Symposium on Microarchitecture*, 2012.

[12] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the 2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2011.

[13] C. Farabet, C. Poulet, J.Y. Han, and Y. LeCun. CNP: An FPGA-based processor for Convolutional Networks. In *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*, 2009.

[14] V. Gokhale, Jonghoon Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014.

[15] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks. In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, 2005.

[16] Atif Hashmi, Andrew Nere, James Jamal Thomas, and Mikko Lipasti. A Case for Neuromorphic ISAs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[17] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data. In *Proceedings of the 22Nd ACM International Conference on Conference on Information &#38; Knowledge Management*, 2013.

[18] INTEL. AVX-512. https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[19] INTEL. MKL. https://software.intel.com/en-us/intel-mkl.

[20] Pineda Fernando J. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.*, 1987.

[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. 2013.

[22] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *Proceedings of the 12th IEEE International Conference on Computer Vision*, 2009.

[23] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *arXiv:1502.01852*, 2015.

[24] V. Kantabutra. On hardware for computing exponential and trigonometric functions. *Computers, IEEE Transactions on*, 1996.

[25] Alex Krizhevsky, Sutskever Ilya, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*. 2012.

[26] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation. In *Proceedings of the 24th International Conference on Machine Learning*, 2007.

[27] Q.V. Le. Building high-level features using large scale unsupervised learning. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[28] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[29] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A Polyvalent Machine Learning Accelerator. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[30] Maashri, A.A. and DeBole, M. and Cotter, M. and Chandramoorthy, N. and Yang Xiao and Narayanan, V. and Chakrabarti, C. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*, 2012.

[31] G Marsaglia and W W. Tsang. The ziggurat method for generating random variables. *Journal of statistical software*, 2000.

[32] Paul A Merolla, John V Arthur, Rodrigo Alvarez-icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D Flickner, William P Risk, Rajit Manohar, and Dharmendra S Modha. A million spiling-neuron interated circuit with a scalable communication network and interface. *Science*, 2014.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. In *Nature*, 2015.

[34] M.A. Motter. Control of the NASA Langley 16-foot transonic tunnel with the self-organizing map. In *Proceedings of the 1999 American Control Conference*, 1999.

[35] NVIDIA. CUBLAS. https://developer.nvidia.com/cublas.

[36] C.S. Oliveira and E. Del Hernandez. Forms of adapting patterns to Hopfield neural networks with larger number of nodes and higher storage capacity. In *Proceedings of the 2004 IEEE International Joint Conference on Neural Networks*, 2004.

[37] David A. Patterson and Carlo H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981.

[38] M. Peemen, A.A.A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for Convolutional Neural Networks. In *Proceedings of the 31st IEEE International Conference on Computer Design*, 2013.

[39] R Salakhutdinov and G Hinton. An Efficient Learning Procedure for Deep Boltzmann Machines. *Neural Computation*, 2012.

[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 the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009.

[41] R. Sarikaya, G.E. Hinton, and A. Deoras. Application of Deep Belief Networks for Natural Language Understanding. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 2014.

[42] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale Convolutional Networks. In *Proceedings of the 2011 International Joint Conference on Neural Networks*, 2011.

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *arXiv:1409.4842*, 2014.

[44] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

[45] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on CPUs. In *In Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[46] Yu Wang, Tianqi Tang, Lixue Xia, Boxun Li, Peng Gu, Huazhong Yang, Hai Li, and Yuan Xie. Energy Efficient RRAM Spiking Neural Network for Real Time Classification. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, 2015.

[47] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the Challenges of Cross-Point Resistive Memory Architectures. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, 2015.

[48] Tao Xu, Jieping Zhou, Jianhua Gong, Wenyi Sun, Liqun Fang, and Yanli Li. Improved SOM based data mining of seasonal flu in mainland China. In *Proceedings of the 2012 Eighth International Conference on Natural Computation*, 2012.

[49] Xian-Hua Zeng, Si-Wei Luo, and Jiao Wang. Auto-Associative Neural Network System for Recognition. In *Proceedings of the 2007 International Conference on Machine Learning and Cybernetics*, 2007.

[50] Zhengyou Zhang, M. Lyons, M. Schuster, and S. Akamatsu. Comparison between geometry-based and Gabor-wavelets-based facial expression recognition using multi-layer perceptron. In *Proceedings of the Third IEEE International Conference on Automatic Face and Gesture Recognition*, 1998.

[51] Jishen Zhao, Guangyu Sun, Gabriel H. Loh, and Yuan Xie. Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface. *ACM Transactions on Architecture and Code Optimization*, 2013.