An Instruction Set Architecture for Machine Learning

YUNJI CHEN, SKL of Computer Architecture, Institute of Computing Technology, CAS; University of Chinese Academy of Sciences; Institute of BrainIntelligence Technology, Zhangjiang Laboratory (BIT, ZJLab); Shanghai Research Center for Brain Science and Brain-Inspired Intelligence (Shanghai Brain/AI)
HUIYING LAN, ZIDONG DU, SHAOLI LIU, JINHUA TAO, DONG HAN, TAO LUO, and QI GUO, SKL of Computer Architecture, Institute of Computing Technology, CAS
LING LI, Institute of Software, Chinese Academy of Sciences, CAS; University of Chinese Academy of Sciences
YUAN XIE, Department of Electrical and Computer Engineering, UCSB, USA
TIANSHI CHEN, SKL of Computer Architecture, Institute of Computing Technology, CAS

Machine Learning (ML) are a family of models for learning from the data to improve performance on a certain task. ML techniques, especially recent renewed neural networks (deep neural networks), have proven to be efficient for a broad range of applications. ML techniques are conventionally executed on general-purpose processors (such as CPU and GPGPU), which usually are not energy efficient, since they invest excessive hardware resources to flexibly support various workloads. Consequently, application-specific hardware accelerators have been proposed recently to improve energy efficiency. However, such accelerators were designed for a small set of ML techniques sharing similar computational patterns, and they adopt complex and

A preliminary version of this study was presented in ISCA 2017: Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*, Seoul, South Korea, June 18–22, 2016, 393–405. In this article, we have made the following new contributions: extend the scope of applications of Cambricon to ML techniques by adding eight new instructions that capture the key computational patterns in classic ML techniques; propose an entire programming framework for Cambricon, including an assembly language, an assembler and runtime; explore the opportunities to improving parallelism and data reuse strategies especially for large networks, and propose a solution to maximize the data reuse.
This work is partially supported by the National Key Research and Development Program of China (under Grants 2017YFA0700900, 2017YFA0700902, 2017YFA0700901, 2017YFB1003101), the NSF of China (under Grants 61432016, 61532016, 61672491, 61602441, 61602446, 61732002, 61702478, 61732007, and 61732020), Beijing Natural Science Foundation (JQ18013), the 973 Program of China (under Grant 2015CB358800), National Science and Technology Major Project (2018ZX01031102), the Transformation and Transfer of Scientific and Technological Achievements of Chinese Academy of Sciences (KFJ-HGZX-013), Key Research Projects in Frontier Science of Chinese Academy of Sciences (QYZDB-SSW-JSC001), Strategic Priority Research Program of Chinese Academy of Science (XDB32050200, XDC01020000), and CAS Center for Excellence in Brain Science and Intelligence Technology (CEBSIT).
Authors' addresses: Y. Chen, SKL of Computer Architecture, Institute of Computing Technology, CAS; University of Chinese Academy of Sciences; Institute of BrainIntelligence Technology, Zhangjiang Laboratory (BIT, ZJLab); Shanghai Research Center for Brain Science and Brain-Inspired Intelligence (Shanghai Brain/AI); email: cyj@ict.ac.cn; H. Lan, Z. Du (Corresponding author), S. Liu, J. Tao, D. Han, T. Luo, Q. Guo, T. Chen, SKL of Computer Architecture, Institute of Computing Technology, CAS; emails: {lanhuiying, duzidong, liushaoli, taojinhua, handong, luotao, guoqi, chentianshi}@ict.ac.cn; L. Li, Institute of Software, Chinese Academy of Sciences, CAS; University of Chinese Academy of Sciences; email: liling@iscas.ac.cn; Y. Xie, Department of Electrical and Computer Engineering, UCSB, Santa Barbara, CA; email: yuanxie@ece.ucsb.edu.

0734-2071/2019/08-ART9 $15.00
https://doi.org/10.1145/3331469

informative instructions (control signals) directly corresponding to high-level functional blocks of an ML technique (such as layers in neural networks) or even an ML as a whole. Although straightforward and easy to implement for a limited set of similar ML techniques, the lack of agility in the instruction set prevents such accelerator designs from supporting a variety of different ML techniques with sufficient flexibility and efficiency.

In this article, we first 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. We then extend the application scope of Cambricon from NN to ML techniques. We also propose an assembly language, an assembler, and runtime to support programming with Cambricon, especially targeting large-scale ML problems. Our evaluation over a total of 16 representative yet distinct ML techniques have demonstrated that Cambricon exhibits strong descriptive capacity over a broad range of ML 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 [7] (which can only accommodate three 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 and 7 other ML benchmarks. Compared to the recent prevalent ML accelerator PuDianNao, our Cambricon-based accelerator is able to support all the ML techniques as well as the 10 NNs but with only approximate 5.1% performance loss.

CCS Concepts: • **Hardware** → *Application specific instruction set processors*;

Additional Key Words and Phrases: Instruction set architecture, machine-learning accelerator, machine learning

ACM Reference format:
Yunji Chen, Huiying Lan, Zidong Du, Shaoli Liu, Jinhua Tao, Dong Han, Tao Luo, Qi Guo, Ling Li, Yuan Xie, and Tianshi Chen. 2019. An Instruction Set Architecture for Machine Learning. *ACM Trans. Comput. Syst.* 36, 3, Article 9 (August 2019), 35 pages.
https://doi.org/10.1145/3331469

1 INTRODUCTION

Machine learning (ML) techniques have been proved to be efficient for a large scope of applications. Among the many technique families in ML, Artificial Neural Networks (NNs), especially recent deep learning, emerged as the most promising one. NNs are a large family of ML techniques initially inspired by neuroscience and have been evolving toward deeper and larger structures over the past decade. Though computationally expensive, NN techniques as exemplified by deep learning [30, 34, 36, 37] have become the state-of-the-art across a broad range of applications (such as pattern recognition [12] and web search [26]), some have even achieved human-level performance on specific tasks such as ImageNet recognition [23] and Atari 2600 video games [45].

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 [10, 15, 63]. Hardware accelerators customized to NNs have been recently investigated as energy-efficient alternatives [5, 7, 16, 39, 44]. 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 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 [7] can efficiently support the Multi-Layer Perceptrons (MLPs) [68] but cannot accommodate the Boltzmann Machines (BMs) [56] 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 article, we first study the design of the ISA for NN accelerators due to the significance of NN techniques, inspired by the success of RISC ISA design principles [50]: (a) First, decomposing complex and informative instructions describing high-level functional blocks of NN techniques (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, 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 5). 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. Furthermore, we extend Cambricon by introducing eight new instructions, which can efficiently support key primitives in ML techniques.

We evaluate Cambricon over a total of 16 benchmarks, including 10 representative yet distinct techniques (MLP [4], CNN [38], RNN [21], LSTM [21], Autoencoder [67], Sparse Autoencoder [67], BM [56], RBM [56], SOM [66], HNN [48]), and 6 representative classic ML techniques (*k*-NN [1], *k*-means [19], decision tree [2, 54, 55], linear regression [13], support vector machine [11], and naive Bayes [35]). We 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 [7] (which can only accommodate three types of NN techniques), our Cambricon-based accelerator prototype (Cambricon-ACC) implemented in TSMC 65nm technology incurs only negligible latency, power, and area overheads (5.1%/8.0%/3.3%, respectively), with a versatile coverage of 10 different NN benchmarks. Compared with the state-of-the-art ML accelerator, PuDianNao [39], Cambricon-ACC is able to support all the ML techniques as well as the 10 NNs but with only approximate 5.1% performance loss.

Although Cambricon has been proved to be flexible and efficient for many ML techniques, there is a gap between Cambricon and developers due to the lack of programming support. To bridge the gap, we propose a high-level assembly language (called Cambricon-AL), composed of pre-defined blocks and built-in data structures to ensure efficiency in both execution and developing.

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 thoroughly analyze the computational primitives in traditional ML techniques and accordingly enhance the ISA by appending eight extra instructions to the Cambricon proposed in our previous work;

(4) We design an assembly language, an assembler, and runtime to enable efficiency in both execution and developing;

(5) We evaluate the effectiveness of Cambricon with an implementation of the first Cambricon-based accelerator using TSMC 65nm technology;

(6) We evaluate the effectiveness of the assembler and our proposed optimization techniques.

The rest of the article is organized as follows. Section 2 briefly discusses a few design guidelines followed by Cambricon and presents an overview to Cambricon. Section 3 introduces computational and logical instructions of Cambricon. Section 4 extends the scope of applications of Cambricon from NNs to ML techniques. Section 5 presents a prototype Cambricon accelerator. Section 6.1 introduces the programming support for Cambricon, including an assembly language, an assembler, and runtime. Section 7 empirically evaluates Cambricon and compares it against other ISAs. Section 8 presents the related work. Section 9 concludes the whole article.

2 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.

2.1 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 [14]) 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 and does not support random vector generation in synapse initialization, dropout [12], and Restricted Boltzmann Machine (RBM) [56]. 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*

Table 1. 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, scalar 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

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 the size of loaded vectors is no longer limited by fixed-width vector register files, as the vector sizes are variables in instructions.

2.2 An Overview to Cambricon

We design the Cambricon following the guidelines presented in Section 2.1 and provide an overview of the Cambricon in Table 1. The Cambricon is a load-store architecture that only allows the main memory to be accessed with load/store instructions. Cambricon contains sixty-four 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 3).

Control Instructions. The Cambricon has two control instructions, *jump* and *conditional branch*, as illustrated in Figure 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.

Data Transfer Instructions. Data transfer instructions in Cambricon support variable data size to flexibly support matrix and vector computational/logical instructions (see Section 3 for such

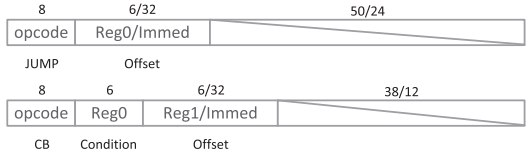

Fig. 1. Top: Jump instruction. Bottom: Condition Branch (CB) instruction.

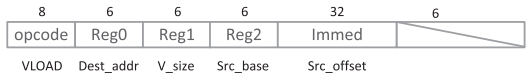

Fig. 2. Vector load (VLOAD) instruction.

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. Figure 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.

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 32×512-bit vector registers have been baked into Intel AVX-512 [27], 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 and 768KB for matrix instructions. Yet Cambricon does not impose specific restriction on bank numbers of scratchpad memory, leaving significant freedom to microarchitecture-level implementations.

Applicability of Cambricon. Cambricon does not reply on any specific microarchitecture-level implementation, and, thus, one can apply Cambricon to different design choices (such as "systolic array") with adequate control logic. From the ISA perspective, with only specifying functional capabilities (input/output model, computation operations, support data types) as shown in Table 1, Cambricon allows multiple implementations. From the hardware perspective, if an accelerator can perform every Cambricon instruction, then it is a Cambricon accelerator. For example, a Cambricon TPU, which use "systolic array" as the major computing component, is totally possible with modifying the control logic of original TPU. In a Cambricon TPU, the *Matrix Multiply Unit*, which performs matrix multiplying matrix or convolution, can support matrix/vector computing instructions such as *MMV, VMM*, and *MMS* directly. The *activation* module, which performs the nonlinear functions and pooling operations, can be modified with control logic to support vector logical, vector element-wise, vector transcendental instructions. The data transfer instructions specify the *load/store* to transfer data between on-chip and off-chip memory (`Read_Host_Memory`

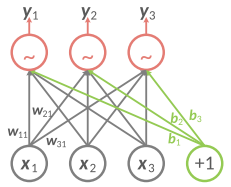

Fig. 3. Typical operations in NNs.

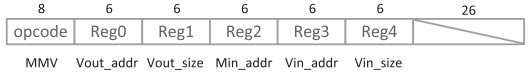

Fig. 4. Matrix Mult Vector (MMV) instruction.

and `Write_Host_Memory` in TPU) and *move* to move data among on-chip buffers. Also, the *control* model can be modified to perform scalar instructions, including control instructions.

3 COMPUTATIONAL/LOGICAL INSTRUCTIONS

In neural networks, most arithmetic operations (e.g., additions, multiplications, and activation functions) can be aggregated as vector operations [15, 63], 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) [61]. 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.

3.1 Matrix Instructions

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) [68], 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 Figure 3. More specifically, the output neuron y_i ($i = 1, 2, 3$) in Figure 3 can be computed as $y_i = f(\sum_{j=1}^{3} w_{ij}x_j + b_i)$, where x_j is the jth input neuron, w_{ij} is the weight between the ith output neuron and the jth input neuron, b_i is the bias of the ith output neuron, and f is the activation function. The output neurons can be computed as a vector $\mathbf{y} = (y_1; y_2; y_3)$:

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

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 3.2).

A critical step in Equation (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 Figure 4, where $Reg0$ specifies the base scratchpad memory address of the vector output ($Vout_{addr}$); $Reg1$ specifies the size of the vector output ($Vout_{size}$); and $Reg2$, $Reg3$, 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.

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 [29], which can be formulated as a vector multiplied by a matrix. If we implement it with the MMV instruction, then 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 that 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) [56].

3.2 Vector Instructions

Using Equation (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, $\mathbf{a}$) can be decomposed into three consecutive steps and are supported by three instructions, respectively:

1. Computing the exponential e^{a_i} for each element $(a_i, i = 1, \ldots, n)$ in the input vector $\mathbf{a}$. 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 $(e^{a_1}, \ldots, e^{a_n})$. 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 $i = 1, \ldots, n$. 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 [32]. 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 3.3.

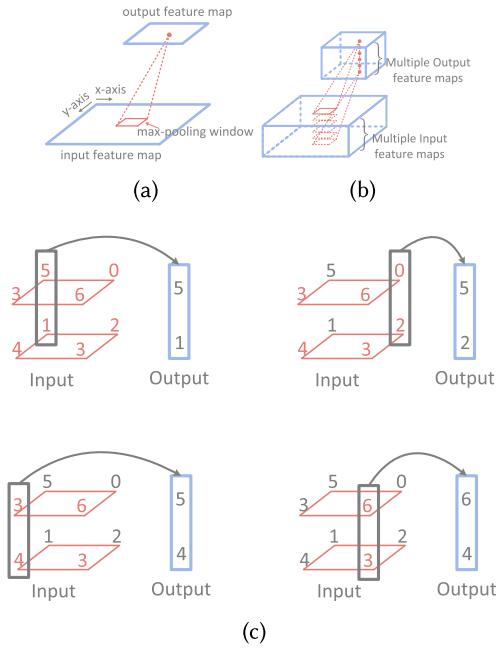

Fig. 5. Max-pooling operation.

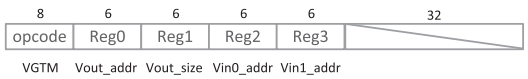

Fig. 6. Vector Greater Than Merge (VGTM) instruction.

Furthermore, the random vector generation is an important operation common in many NN techniques (e.g., dropout [12] and random sampling [56]) 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 [43], with the help of vector arithmetic instructions and vector compare instructions in Cambricon.

3.3 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 Figure 5(a) 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 Figure 5(b)). Cambricon supports the max-pooling operation with a Vector-Greater-Than-Merge (VGTM) instruction, see Figure 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 3.5, which aggregates neurons at the same position

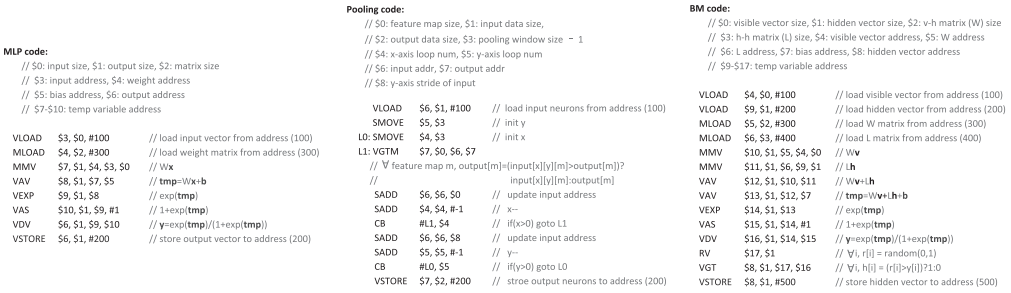

Fig. 7. Cambricon program fragments of MLP, pooling, and BM.

of all input feature maps in the same input vector, iteratively performs VGTM, and obtains the final result (see also Figure 5(c) for an illustration).

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.

3.4 Scalar Instructions

Although we have observed that only 0.008% arithmetic operations of the GoogLeNet [61] 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 1, which have been formally defined as Cambricon's scalar instructions.

3.5 Code Examples

To illustrate the usage of our proposed instruction sets, we implement three simple yet representative components of NNs, a MLP feedforward layer [68], a pooling layer [30], and a Boltzmann Machines (BM) layer [56], 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 Figure 7, and we observe that the code density of Cambricon is significantly higher than that of x86 and MIPS (see Section 7 for a comprehensive evaluation).

4 CAMBRICON FOR CLASSIC ML TECHNIQUES

In this section, we extend the application scope of Cambricon from NN to ML techniques. Initially, with the scalar instructions, original Cambricon ISA (refer as Cambricon-NN in the rest of article) is able to support ML techniques naturally but very inefficiently. Therefore, we construct Cambricon-ML built upon Cambricon-NN to support ML techniques. From the perspective of hardware architecture, an ML technique can be characterized by its computational primitives and local property [39]. In this section, we will focus on decomposing the ML primitives into vector/matrix operations and enhance the Cambricon-NN ISA based on the lacked or existed but inefficient functionalities. We will focus on the challenge of locality in Section 6.

```
Counting:
// $1: input vector, $2: vector size
// $3: vector of all 1
// $4: reg stored the label
// $5: output vector
// $6: reg store count result
 VEQS   $5, $2, $1, $4
 VDOT   $6, $2, $3, $4
```

Fig. 8. *Counting* implemented with Cambricon-NN.

4.1 Cambricon-NN for ML Techniques

We choose six representative classic ML techniques from Reference [39] as our benchmark for analysis. By accommodating each ML technique with Cambricon-NN, we are able to identify the unsupported or time-consuming computational primitives.

***k*-means.** *k*-means is a unsupervised ML algorithm that divides the dataset into *k* clusters. The centroids of clusters are iteratively updates. In a iteration, each sample is classified into the cluster that has closest centroid. The centroid will recompute its location by averaging on all samples in its cluster. Two computational primitives are involved in this process: *dist* computes distance of two vectors (e.g., input samples and centroids), and *argmin* finds the index having minimal value (e.g., nearest centroid). The computing of distance can be decomposed into VSUB, VMUL, and VDOT and utilizes the data-level parallel of the ISA. However, to accommodate *argmin* of a *k*-length vector, we have to traverse the vector using scalar and branching instructions to obtain the index of minimal value, as the original ISA does not include instructions that return indexes or addresses. This will incur a dramatic performance loss.

***K*-Nearest-Neighbour (*k*-NN).** *K*-NN is a supervised learning algorithm using lazy training, which does not require a training phase. A sample x_i is classified by two steps. The first step is to compute the distances of x_i and N training samples, where N is the total number of training samples. The second step is to find the labels of the nearest k samples. The majority label of the k samples will be the label of x_i. There are two computational primitives, *dist* and *k-sort-by-key*. *Dist* can be accommodated with Cambricon-NN, but *k-sort-by-key* is a tricky operation, as it is implemented by decomposing into *k argmin* operations, each of which returns a minimum distance.

Decision Tree (DT). Decision tree is a supervised learning algorithm, generating a treelike model in the training phase and consequentially using to predict new samples in the inference phase. In the training phase, DT splits the original set S into subsets and keeps splitting subsets into smaller subsets until each subset cannot be divided. In each splitting, DT selects a best splitting strategy according to the learning metrics (e.g., information gain for ID3 or gini value for CART). In this process, there are three most frequently appeared computational primitives, and none of them can be directly accommodated by Cambricon-NN. The first primitive is *counting*, which is used to calculate the metrics. Using Cambricon-NN, *counting* can be accommodated with vector instructions, as shown in Figure 8. There are three most frequently appeared primitives, that is, *counting*, *filter*, and *argmax*, in the decision tree algorithm. *Filter* is used while calculating the entropy of a specific feature and generating new data nodes. In the former computation, label data are selected, and in the latter phase, sample data are filtered. *Argmax* is used to get the index of the chosen attribute according to the information gain. All primitives are expended in Cambricon-ML.

Support Vector Machine. Support Vector Machine (SVM) is a supervised learning algorithm for classification or regression; it finds the optimal hyperplane in data space that has the maximum distances to the nearest training samples. In the training phase, SVM iteratively adjusts a set of coefficient α_i that represents the current hyperplane to maximize that distances. The most commonly

Table 2. Representative Classic ML Techniques, Their Computational Primitives, and Instructions in Cambricon-NN Used to Perform These Primitives

Technique	Primitive	Cambricon-NN
K-means	*Dist*, *Argmin*	VMULV, VSUBV, VDOT, Scalar Inst., CB
K-NN	*Dist*, *k*-sort-by-key	VMULV, VSUBV, VDOT, Scalar Inst., CB
Decision Tree	*Counting, Filter, Argmax*	VEQV, VDOT
Support Vector Machine	Kernel function	VEXP, VADD, VSUB, VMUL
Linear Regression	Multiplication between matrix and vector	VMM, MMV, VADD
Naive Bayes	*Counting*	VEQV, VDOT

used training algorithm is the SMO [52], which used in this article. And in the inference phase, we use the set of α_i to predict the label (y) of a test sample (x) by $y(x) = \sum_{i=1}^{N} \alpha_i y^{(i)} k(x, x^{(i)} + b)$, where N is the number of training samples; $x^{(i)}$ and $y^{(i)}$ are the ith training sample and its label, respectively; $k(\cdot, \cdot)$ is the kernel function; and b is a constant. There are three types of computational primitives in SVM: vector operations for calculating kernel functions and updating coefficient α_i (e.g., expectation, vector addition, vector multiplication); multiplication between matrix and vector for compute label y; and *argmax*, which is used to find label for a testing sample in multiple-class classification. The first two types of primitives can be directly supported with vector/matrix instructions in Cambricon-NN, but the *argmax* primitive is otherwise for the same reason with *argmin*.

Linear Regression (LR). Linear regression is a linear approach for supervised learning. It uses a linear predictor function $y = \sum_{i=0}^{d} \theta_i x_i$ that models the relationship between a scalar dependent variable y and a d-dimensional vector $x = (x_1, x_2, \ldots x_d)^T$. LR has two learning phases. In the training phase, parameters of the predictor function (θ) are iteratively tuned with the Back Propagation (BP) algorithm. The inference phase is to predict the label for each testing samples using the predictor function. The computational primitives in LR is similar to that in NN techniques, i.e., multiplication between vector and matrix and vector/matrix element-wise operations, therefore, LR can be directly accommodated with Cambricon-NN.

Naive Bayes (NB). Naive Bayes is a supervised learning technique that trains a probabilistic classifier based on Bayes's theorem [35]. NB has two phases: the training phase, which estimates individual conditional probabilities from training samples, and the inference phase, which classifies the test samples. The primary task of the training phase is to estimate all $p(F_i = f_{ij}|C = c_k)$, where $F_i (i = 1, \ldots d)$ is the features, $f_{ij} (j = 1, \ldots a)$ is the values that F_i can take from, and $C_k (k = 1, \ldots b)$ is the value of the class. Total $d \times a \times b$ probabilities are estimated and construct a table that is used in the inference phase. The computational primitive in NB is *counting*, which is repeatedly used to obtain frequency for estimating the probabilities. As we discussed above, *counting* is a time-consuming primitive for Cambricon-NN. Moreover, NB suffers the same issue as DT. To estimate all $p(F_i = f_{ij}|C = c_k)$, where $i = (1, \ldots, a)$, we first select samples that belong to class c_k and form a much smaller subset and on which we compute the frequencies of each f_{ij} by *counting*. Same with DT, although samples of other classes are filtered, they are still counted while using binary representation to implement.

Summary. Table 2 summarizes the computational primitives of classic ML techniques and their corresponding implementations in Cambricon-NN. As discussed above, only the LR technique can be effectively accommodated with Cambricon-NN without performance loss; the remaining five techniques contains primitives that cannot be directly supported, e.g., *counting* or *argmax*.

```
float v1[N];
float filtered_v[N];
float v2[N];
float a = A;
// possible values
float b[bN];
// results
int count[bN];
// select v2[i] where v1[i] == a
int c1 = 0;
for (int i = 0; i < N; ++i) {
    // filter
    if (v1[i] == a) {
        filtered_v[c1] = v2[i];
        c1++;
    }
}
// count different values in filtered_v
for (int j = 0; j < bN; ++j) {
    count[j] = 0;
    for (int k = 0; k < c1; ++k) {
        if (filtered_v[k] == b[j])
            count[j]++;
    }
}
```

Fig. 9. Key computational primitive 1: Counting under conditions.

```
float key[N];
float value[N];
float results[K];
for (int k = 0; k < K; ++k) {
    // argmax
    int max_idx = 0;
    float max = -inf;
    for (int i = 0; i < N; ++i) {
        if (key[i] > max) {
            max = v[i];
            max_idx = i;
            results[k] = value[i];
        }
    }
    key[max_idx] = -inf;
}
```

Fig. 10. Key computational primitive 2: K-sort-by-key.

4.2 Key Computational Primitives

Although Cambricon-NN is able to support all primitives in classic ML techniques despite the efficiency, some primitives are key to certain techniques, thus causing severe performance loss (see Section 7). In this section, we introduce such two key primitives and analyze them for Cambricon-ML construction.

4.2.1 Key Primitive: Counting under Conditions. The computational primitive, *counting under conditions*, is extensively used in DT and NB, as shown in pseudo code in Figure 9. Both algorithms need to count the number of instances that satisfies certain conditions. In DT, instances that have certain labels and attribute values are counted. Specifically, *v1* represents an array of attributes, *v2* is the array of labels for all samples, and N is the total number of the training samples. The *i*th label in *v2* is selected if the attribute of the same sample (i.e., *v1*[i]) is equal to a specific value (*b*) to form a subset of label with size *c1* (i.e., *filtered_v*). Then *counting* is applied to get the numbers of different labels. In NB, the primitive is used to count the numbers of attribute possible values. Specifically, after computing $p(F_i = f_{ij}|C = c_k)$ for all $j = 1 \ldots b$, where *v1* represents the labels of all training samples, *float a* equals c_k, *filtered_v* is the attribute that belongs to class c_k, and *c1* is the length of *filtered_v*, the attributes of class c_k are selected. Then, *counting* is applied to get the number of each possible value of that attribute. The value of c1 could be much smaller than N. For example, UCI Convertype dataset has a total of 522,000 training samples with live classes, where each class accounts for approximate 20%, which construct the subset that is used for counting. This means that if we use binary vector to do the counting, about 80% computations are redundant. In this primitive, there are two atomic operations: counting a specific value in a vector and filtering out elements that satisfy certain conditions from a vector. From this primitive, we observe two frequently used computational patterns: filtering and counting certain conditions for a vector.

4.2.2 Key Primitive: k*-sort-by-key.* The other key primitive is *k-sort-by-key*, which finds the k values with maximum or minimum keys. When $k = 1$, it becomes the *argmax/argmin*. This primitive is widely used in many ML techniques, e.g., *k*-NN, *k*-means, and SVM. Using Cambricon-NN,

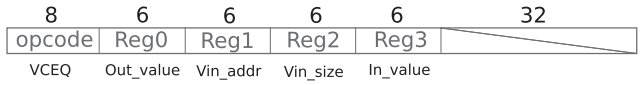

Fig. 11. Vector Count Equal (VCEQ) instruction.

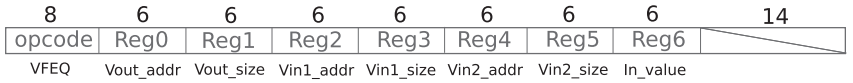

Fig. 12. Vector Filter Equal (VFEQ) instruction.

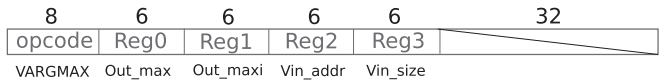

Fig. 13. Vector Argmax (VARGMAX) instruction.

k-sort-by-key can only be performed with conditional branch (CB) and scalar instructions, which is time-consuming. The code of *k*-sort-by-key (finding the maximum *k* values) is listed in Figure 10, where code of line 7 to 18 performs the operation of *argmax*. Each time finding a value, the corresponding key needs to be modified so that it will not be considered any more in later computations. The code clearly shows that *k*-sort-by-key can be expressed as *k* times *argmax/argmin* operations, anad, thus, naturally, we extract the atomic operation of *argmax/argmin* from this primitive. *K*-sort-by-key is implemented by multiple *argmax/argmin* in addition to a few scalar operations.

4.3 New Instructions

To keep the same spirit for high efficiency, we build Cambricon-ML upon Cambricon-NN by adding three types (eight in total) of new instructions to support the key primitives in classic ML techniques. All new instructions are logical vector instructions, as they all contain logical operations, i.e., comparison.

Counting Instructions. Counting instructions counts the number of elements that equal to (VCEQ), greater than (VCGT), or less than (VCLT) a certain value in a vector. Figure 11 shows the format of VCEQ instructions (VCGT and VCLT have the same format). For an input vector *Vin*, it counts the amount of element that equals to *In_value*, which is a scalar stored in Reg3, and stores the amount, *Out_value*, to Reg0.

Filter Instructions. The format of filter instructions is shown in Figure 12. As with the Counting instructions, there are three filter instructions to deal with the three comparison operations, i.e., equal to (VFEQ), greater than (VFGT), or less than (VFLT). One thing special about our filter instruction is that elements in *Vin1* are filtered and selected according to another vector (*Vin2*) instead of *Vin1* itself. *Vin2* functions as an index. An element *Vin1[i]* is selected if *Vin2[i]* = *In_value*. Eventually, the selected elements will form a new vector *Vout*.

Argmax/argmin Instructions. The last two appended instructions are vector argmax (VARGMAX) and argmin (VARGMIN) instructions, the format of which is depicted in Figure 13. The function of these two instructions are straightforward—to return the maximum value in *Vin* and its corresponding index, and store these two scalars in registers. While computing *k*-sort-by-key, *Out_maxi* is used as an offset to an initial address of a vector, so that we can modify the value at the same position of a certain vector or retrieve that value from the vector.

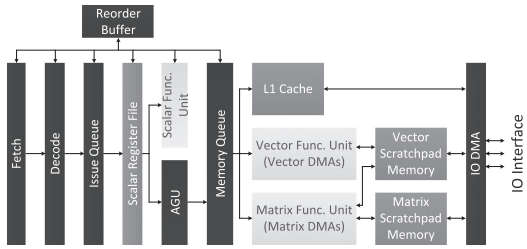

Fig. 14. A prototype accelerator based on Cambricon.

5 A PROTOTYPE ACCELERATOR

In this section, we present a prototype accelerator of Cambricon. We illustrate the design in Figure 14, 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 7.2.1), conciseness (Section 7.2.2), and efficiency (Section 7.2.3) of the ISA. We did not seek to explore the emerging techniques (such as 3D stacking [69] and non-volatile memory [8, 53, 64, 65]) 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 Figure 14, 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, and data transfer/computational instructions for matrices will be sent to matrix functional unit. After the execution, such an 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 thirty-two 16-bit adders and thirty-two 16-bit multipliers and is equipped with a 64KB scratchpad memory. The matrix unit contains 1,024 multipliers and 1,024 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.

[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.

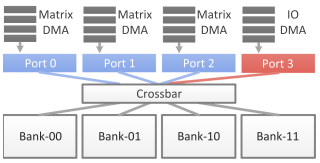

Fig. 15. Structure of matrix scratchpad memory.

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 Figure 15). Concretely, we decompose the memory into four banks according to addresses' low-order two bits and 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.

6 PROGRAMMING FRAMEWORK

As a flexible ISA, Cambricon provides the possibility to support a wide range of ML techniques but meanwhile complicates the programming task, as developers need to manually aggregate low-level instructions in Cambricon to perform high-level algorithms. Fortunately, although the scope of ML techniques is large, they share a limited set of computational blocks that are repeatedly used in these techniques. For example, MLP, RNN, LSTM, and Autoencoder all utilize the fully connected block, and both *k*-NN and *k*-means contain the computation of distance between two vectors. Therefore, in this section, we propose a programming method for Cambricon that leverages this characteristic inherent in ML techniques to ease the burdon of programming.

An intuitive solution is to provide a code generator written in high-level programming languages that supports commonly used blocks like convolution block, pooling block, and fully connect block. A code generator is easy to use and efficient if it is written by professions of the hardware architecture, but the shortcoming is also obvious—the flexibility of Cambricon is obliterated. Users are not able to develop new blocks as the instructions are not exposed to them.

To balance the flexibility and user-friendliness, we propose a high-level assembly language (Cambricon-AL) and an assembler (Cambricon-ASM), which promise users the full accesses to the hardware resources so that they can utilize the flexibility of Cambricon and a set of pre-defined computational blocks to improve friendliness.

6.1 Cambricon-AL

6.1.1 The Basic Language. The basic language as the basic material to construct programs is to promise the accessibility of the hardware and the flexibility of the language. Although the basic language is intentionally designed to be close to the original ISA, its statements and types are not identical to those in the ISA. As Cambricon-ISA is open to evolve, Cambricon-AL is also designed to support new operations and datatypes. The basic language currently supports three data types: dense *vector*, dense two-dimensional *matrix*, and *scalar* (all using 16-bit fixed-point data), which are derived from the three data types in Cambricon, *vector*, *matrix*, and *scalar*. In addition, Cambricon-AL also support macro statements to provide abstraction.

```
.code
entry:
  mem_copy t1, t1_cpu, 9216, host2acc;
  @fc_fp t2, t1, f1, b1;
  @fc_fp t3, t2, f2, b2;
  mem_copy t3, t3_cpu, 1024, acc2host;

.data
  @Tensor t1, 1, 1, 1, 1, 1, 1, 1, 9216, 256, 256
  @Tensor t2, 1, 1, 1, 1, 1, 1, 1, 4096, 256, 256
  @Tensor t3, 1, 1, 1, 1, 1, 1, 1, 1024, 256, 256
  @Tensor b1, 1, 1, 1, 1, 1, 1, 1, 4096, 256, 256
  @Tensor b2, 1, 1, 1, 1, 1, 1, 1, 1024, 256, 256
  @Filter f1, 4096, 256, 256, 1, 1, 1, 1, 1, 1, 9216, 256, 256
  @Filter f2, 1024, 256, 256, 1, 1, 1, 1, 1, 1, 4096, 256, 256
```

Fig. 16. Implementing a two-layer FC network by using NN blocks.

6.1.2 Blocks. The set of *block* in Cambricon-AL serves like a built-in library, and each function in this library (i.e., a block) is implemented by purely basic statements to ensure execution efficiency. The set covers a small but representative range of computational primitives and allows users to implement their own user-defined blocks. By using blocks, the developing efficiency can be largely improved. Figure 16 demonstrates an example of implementing a network with two fully connected layers by using blocks and data structures. Data are declared in the **.data** section. *mem_copy* is a built-in function used to copy data between host and device. An important attribute of the *block* is the ability to process arbitrary size of input and output data. Due to the limitation of on-chip memory, an operation needs to be partitioned into several sub-operations so that the required data can be fit in.

Block is designed to resolve two major obstacles of programming with Cambricon: (1) allowing reuse of high-level computational primitives, especially when the implementations of these primitives are fairly complex (for example, the convolution operation requires multiple loop nests and complex memory access patterns), and (2) eliminating the need for programmers to manually handle data partitioning. Due to the limitation of on-chip memory, an operation needs to be partitioned into several sub-operations so that the required data can be fit in. This problem greatly increases the burdon of programming, especially for NN techniques, which require a lot of data transferring. In addition to merely ensure the data is sliced to a fitable size, we also concern about the reuse strategy that effects the performance and tightly coupled with the partitioning strategy. We will discuss the partitioning strategy in Section 6.3.2.

6.1.3 Data Structure. Cambricon-AL supports three data types, *Tensor*, *Filter*, and *Parameter*. The input and output of NN operators are mostly n-dimensional arrays ($1 \leq n \leq 4$). However, operands in Cambricon are represented by two-dimensional matrix and one-dimensional vector. To feed proper data to NN operations, we level up the one- and two-dimensional data types in Cambricon-ISA to a multi-dimensional array by the name of *Tensor* and *Filter*. *Tensor* represents a n-dimensional array ($1 \leq n \leq 4$) that will be finally broken down to be accessed and operated as *vectors*, e.g., input and output neuron, gradient in backward propagation, and bias. *Filter* represents a n-dimensional matrix ($1 \leq n \leq 4$) that will be finally broken down to be accessed and operates as *matrix*, e.g., synapses in convolution and fully connected layers. *Parameter* are scalars that specify the network topology (e.g., image dimensions, kernel sizes and strides) or affect the training process, e.g., learning rate, momentum.

Table 3. Parameters of Data Structures

Structure	Parameter	Explanation
Tensor	batch.	batch size
	H	height dimension
	SS_h	segment size of the height dimension
	Str_h	stride of the height dimension
	W	width dimension
	SS_w	segment size of the width dimension
	Str_w	stride of the width dimension
	C	channel dimension
	SS_c	segment size of the channel dimension
	Str_c	stride of the channel dimension
Filter	CO	output channel
	SS_{co}	segment size of the output channel
	Str_{co}	stride of the output channel
	H	height dimension
	SS_h	segment size of the height dimension
	Str_h	stride of the height dimension
	W	width dimension
	SS_w	segment size of the width dimension
	Str_w	stride of the width dimension
	CI	input channel
	SS_{ci}	segment size of the input channel
	Str_{ci}	stride of the input channel

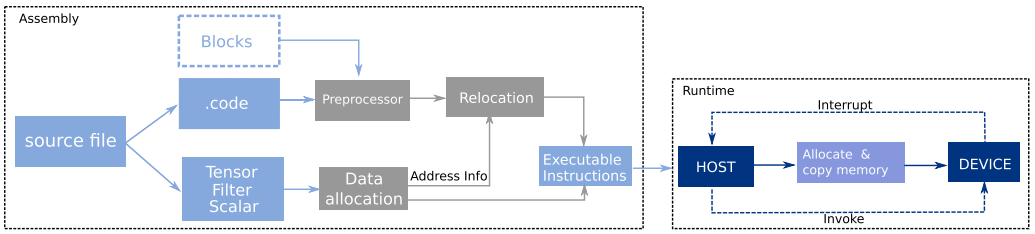

Fig. 17. The architecture of the assembler and runtime for Cambricon-AL.

As mentioned, large-scale data needs to be partitioned and sequentially processed. Therefore, each dimension in *Tensor* and *Filter* has at least two parameters, the total size and the segment size, which are utilized in loop tilings. A complete parameter list of *Tensor* and *Filter* is tabulated in Table 3. The *Str* denotes the moving stride of a dimension, which is a common accessing pattern in convolution and pooling. For example, in a convolution of stride 1, the index of H or W dimension of input data is shifted for one instead of segment size. The default stride is set to the segment size.

6.2 Assemble Process and Runtime

In Figure 17, we demonstrate the architecture of the assembler, which takes a source file as input and produces executable instructions combined by code for both the host and the device. There are three phases in the assembler: preprocessing, data allocation, and reallocation. (1) Preprocessing: To achieve high performance, macro is heavily used in Cambricon-AL. While preprocessing, *blocks*

and macro statements are substituted by statements in the basic language. (2) Data allocation Cambricon-AL supports static data whose scales are determined before execution. We believe that static data are enough for ML techniques, as their topologies are fixed during execution. Addresses on the device are sequentially assigned according to declarations in the **.data** section. Data can be transferred between the host and the device through a driver. (3) Relocation: The relocation phase uses the address information from data allocation to recompute the addresses in the program. It replaces relative addresses in the preprocessed source file with absolute address and then outputs the replaced file. This replaced file is the target executable binary machine code that will be sent to the host for execution.

Runtime. For mainstream programming languages executing on common operating systems, runtime is handled by the operating system—for example, a loader&linker first performs the necessary memory setup and links the program with any dynamically linked libraries it needs before starting execution from the program's entry point. We also design runtime support for Cambricon.

The runtime of Cambricon-AL is presented in the right of Figure 17. First, the program is called by the host, and data are allocated and initialized. Then, the CPU sends the program to Cambricon-ACC and invokes the device by sending an extra signal. The ACC program will be executed until all instructions are finished, and then a finish signal is sent back to the host. The host will take over control and perform the rest of the operations (e.g., copying result data from device space to host space). During copying, data are rearranged according to the segment information defined in the **.data** section.

6.3 Exploit the Potential of Cambricon

In this section, we explore the potential of Cambricon by discussing optimizing the two most commonly used algorithms (layers) in NN techniques, convolutional layer and fully connected layer. We discuss the optimizations from two perspectives—improving parallelism between memory access and computations and improving data reuse.

6.3.1 Improving Parallelism. Although Cambricon includes matrix/vector operations to enable data-level parallelism, it also allows parallel for instructions. Cambricon allows four instructions to execute in parallel (i.e., a scalar instruction, a vector instruction, a matrix instruction and a memory access instruction). Due to the memory-intensive and computational-intensive nature of NN algorithms, increasing parallelism between memory access instructions and computational instructions can largely improve the performance.

As Cambricon does not provide a specific instruction for synchronizing, we use another approach for synchronization. Cambricon-ACC allows an issue width of 2 to achieve the parallel execution among computations and memory accesses. As Cambricon does not provide a specific instruction for synchronizing, we use another approach for synchronization—inserting a useless instruction (e.g., NOP or instruction with same OP but meaningless operands such as zeros) to separate the two related instructions into different issue time slots.

The challenge of maximizing parallelism in Cambricon comes from the contradiction between low-level computational instructions and the limited depth of issue queue. Using low-level instructions to implement a complex algorithm leads to a longer instruction sequence. For example, Cambricon needs nine instructions to perform a pooling operation (as shown in Figure 7). However, due to the issue width (i.e., 2), if we want to overlap these computational instructions with the next loading operation, then we need to manually adjust the order of instructions—moving the loading instruction to the position before the nine computational instructions, so that they can be executed in parallel with the loading instruction.

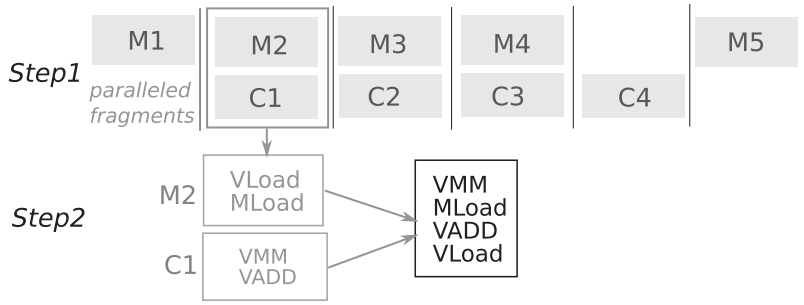

Fig. 18. Applying the two-step approach to a FC layer.

We propose a two-step approach to help programmers to write parallel programs using Cambricon-AL. We use a fully connected (FC) layer as a driving example to demonstrate the approach (see Figure 18).

Step 1. Identifying and arranging paralleled fragments. Due to the purpose of increasing parallel between memory accesses and computations, we first map the algorithm into operations composed of two types of operation, i.e., computation and memory access operations. Figure 18 shows a fully connected layer whose input data are partitioned into four segments, loaded (M1–M4) and computed (C1–C4) sequentially. The final results are stored back to main memory by M5. While computing ith data, i.e., C_i, we can load the next required data using M_{i+1}, and these two operations are identified as the paralleled fragments.

Step 2. Scheduling instructions for paralleled fragments. After identify the key paralleled fragments, we translate the operations into specific statements in the correct order. Generally, computation fragments are composed with more statements than memory access fragments; therefore, we suggest to insert memory access statements between computational statements. M_{i+1} is translated into two loading statements, i.e., VLOAD and MLOAD, and C_i is translated into two computational statements, i.e., VMM and VADD. We insert MLOAD after VMM to overlap with it, meanwhile blocking the VADD, and then insert VLOAD after VADD so that these two statements can be executed in parallel.

6.3.2 Improving Data Reuse. Cambricon equips with matrix scratchpad memory of 768KB, and vector scratchpad memory of 64KB, which is far from enough for large-scale networks [24, 34, 60]. For example, VGG16 [60] contains convolutional layers with parameters up to 4.5MB (*conv4_2*). Regarding convolution and fully connected layer, reasonable loop tiling could largely improve data reuse, so that reduce redundant memory accesses [5]. As the optimal reuse strategy can vary according to layer scales and segmentation sizes, we estimate the workloads of each layer according to formulas and select the strategy that minimizes the memory accesses.

For convolutional layers, there are three types of reuse strategies, input-reuse (InR), output-reuse (OutR), and synapse-reuse (SR), as shown in Figure 19, and for a fully connected layer, there are two types of reuse strategies, input-reuse (InR) and output-reuse (OutR). The time consumed by memory access is affected by two factors, the accessed workloads and the latency of each operation. We estimate the time considering both the factors. Table 4 tabulates the operation number and the corresponding segment size for each reuse strategy of Conv. and FC. The memory access time can be estimated by $AccessN \times (\frac{DataAmount}{Bandwidth} + Latency)$, where each accessing time is estimated by $\frac{DataAmount}{Bandwidth} + Latency$, and the total access time is calculated by multiplying access number (i.e., AccessN). In the context of estimating the memory access in a NN algorithm, this formular is reduced to $\frac{SS_i \times OpNumber}{B} + OpNumber \times L$, where $i = \{in, out, syn\}$, B, and L denote the bandwidth

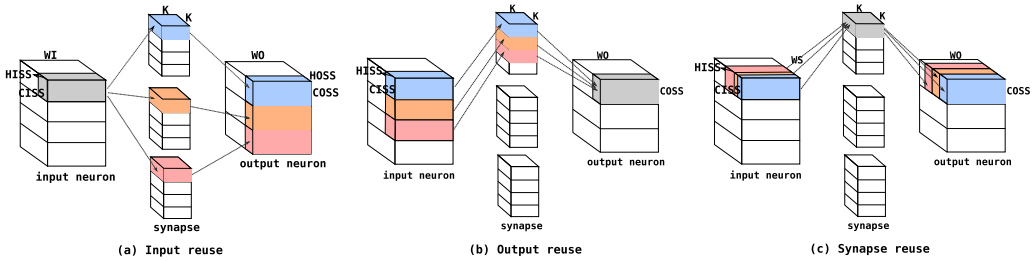

Fig. 19. Reuse strategies of a convolution layer. The colors represent three sequential operations under the specific reuse strategy. The gray part represents the reused data block, and the blue, orange, and pink blocks are the data that need to be loaded to the scratchpad memory to complete the operations.

Table 4. Formulas to Compute the Memory Workloads

Layer	Reuse	SegSize	Op number
Conv.	Input	SS_{in}	$SN_{ic} * SN_{oh} * SN_{ow}$
		SS_{out}	$SN_{ic} * SN_{oh} * SN_{ow} * SN_{oc} * 2$
		SS_{syn}	$SN_{ic} * SN_{oh} * SN_{ow} * SN_{oc}$
	Output	SS_{in}	$SN_{oc} * SN_{oh} * SN_{ow} * SN_{ic}$
		SS_{out}	$SN_{oc} * SN_{oh} * SN_{ow}$
		SS_{syn}	$SN_{oc} * SN_{oh} * SN_{ow} * SN_{ic}$
	Synapse	SS_{in}	$SN_{ic} * SN_{oc} * SN_{oh} * SN_{ow}$
		SS_{out}	$SN_{ic} * SN_{oc} * SN_{oh} * SN_{ow} * 2$
		SS_{syn}	$SN_{ic} * SN_{oc}$
FC.	Input	SS_{in}	SN_{ic}
		SS_{out}	$SN_{ic} * SN_{oc} * 2$
		SS_{syn}	$SN_{ic} * SN_{oc}$
	Output	SS_{in}	$SN_{ic} * SN_{oc}$
		SS_{out}	SN_{oc}
		SS_{syn}	$SN_{ic} * SN_{oc}$

SN and SS refer to segment number and segment size, respectively. ic, oc, oh, and ow denote dimensions of input channel, output channel, output height, and output width, respectively. in, out, and syn denote input, output, and synapse, respectively.

and latency. We select the strategy that minimizes the data movement. We call this strategy the dynamic data reuse strategy (DR), as it automatically selects the most suitable strategy according to the layer scale. Compared to use the same reuse strategy for all layers, DR can significantly improve the performance of IO-intensive algorithms (see Section 7.3).

7 EXPERIMENTAL EVALUATION

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

7.1 Methodology

Design Evaluation. We synthesize the prototype accelerator of Cambricon (Cambricon-ACC, see Section 5) 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

Table 5. 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 × 32)
bank width of scratchpad memory	512 bits (32 × 16-bit fixed point)
operators in matrix function unit	1,024 (32 × 32) multipliers & adders
operators in vector function unit	32 multipliers & dividers & adders & transcendental function operators

simulated Value Change Dump (VCD) file. We are planning an MPW tape-out of the prototype accelerator, with a small area budget of 60mm^2 at a 65nm process with targeted operating frequency of 1GHz. Therefore, we adopt moderate functional unit sizes and scratchpad memory capacities to fit the area budget. Table 5 shows the details of design parameters.

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

• *CPU.* The CPU baseline is an x86-CPU with 256-bit SIMD support (Intel Xeon E5-2620, 2.10GHz, 64GB memory). We use the Intel MKL library [28] 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 select the K40 GPU to have a relative fair comparison, as it has only ~2× peak performance (4.29Tops/s) with respect to our Cambricon-ACC (2.25Tops/s). We implement all benchmarks (see Table 6 and Table 7) with the NVIDIA cuBLAS library [47], a state-of-the-art linear algebra library for GPU.

• *NN Accelerator.* The baseline accelerator for evaluating NN techniques is DaDianNao, a state-of-the-art NN accelerator exhibiting remarkable energy-efficiency improvement over a GPU [7]. 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 (<60mm^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, thirty-two 16-bit adders, and thirty-two 16-bit multipliers; each leaf tile has 24KB SRAM, thirty-two 16-bit adders, and thirty-two 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. In addition, we use TPU [31], a state-of-the-art systolic-like architecture, as the baseline to evaluate RNN and LSTM techniques. We use the original architecture configurations including 256 × 256 PEs.

Table 6. Benchmarks of NN Techniques

Technique	Network Structure	Description
MLP	input(64) - H1(150) - H2(150) - Output(14)	Using Multi-Layer Perceptron (MLP) to perform anchorperson detection [4]
CNN	input(1@32×32) - C1(6@28×28, K: 6@5×5) - S1(6@14×14, K: 2×2) - C2(16@10×10, K: 16@5×5) - S2(16@5×5, K: 2×2) - F(120) - F(84) - output(10)	Convolutional neural network (LeNet-5) for hand-written character recognition [38]
RNN	input(26) - H(93) - output(61)	Recurrent neural network (RNN) on TIMIT database [21]
LSTM	input(26) - H(93) - output(61)	Long-short-time-memory (LSTM) neural network on TIMIT database [21]
Autoencoder	input(320) - H1(200) - H2(100) - H3(50) - Output(10)	A neural network pretrained by auto-encoder on MNIST dataset [67]
Sparse Autoencoder	input(320) - H1(200) - H2(100) - H3(50) - Output(10)	A neural network pretrained by sparse auto-encoder on MNIST dataset [67]
BM	V(500) - H(500)	Boltzmann machines (BM) on MINST dataset [56]
RBM	V(500) - H(500)	Restricted Boltzmann machine (RBM) on MINST dataset [56]
SOM	input data(64) - neurons(36)	Self-organizing maps (SOM) based data mining of seasonal flu [66]
HNN	vector (5), vector component(100)	Hopfield neural network (HNN) on hand-written digits dataset [48]

H Denotes Hidden Layer, C Denotes Convolutional Layer, K Denotes Kernel, P Denotes Pooling Layer, F Denotes Classifier Layer, V Denotes Visible Layer.

• *ML Accelerator.* We select PuDianNao, a state-of-the-art ML accelerator designed to support seven representative ML techniques while ensuring high efficiency of power and area, as our baseline ML accelerator. We re-implement the PuDianNao architecture at 65nm to have similar numbers of operators for a fair comparison. The re-implemented version of PuDianNao has 32 MLU, each of which processes 32 data in a cycle. In each MLU, there are 32+32+31+1+1= adders, coming from the Counter stage, the Adder stage, the Adder tree stage, the Acc stage, and the Misc stage, respectively. Each MLU contains 32+1=33 multipliers, coming from the Multiplier stage and the Misc stage, respectively. The three on-chip data buffers are also scaled. In the re-implemented version, the Cold buffer capacity is 32 × 24KB (equals to the the capacity of the matrix scratchpad memory), and the capacity of Hot buffer and Output buffer are scaled to 32KB (each accounts for a half of the capacity of the vector scratchpad memory).

Benchmarks. We select two set of benchmarks (total 16 benchmarks) to respectively evaluate NN techniques and classic ML techniques—10 representative NN techniques (see Table 6) and 6 classic ML techniques (see Table 7). Each benchmark is translated manually into assemblers to execute on Cambricon-ACC, DaDianNao, and PuDianNao. We evaluate their cycle-level performance with Synopsys VCS.

7.2 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.

7.2.1 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 ML accelerators. DaDianNao [7]

Table 7. Benchmarks of Classic ML Techniques

ML technique	Dataset	Probleme description
k-NN	MNIST	60,000 training samples, 10,000 testing samples, 784 features, $k = 20$, class = 10
k-means	MNIST	60,000 samples, 784 features, $k = 10$
LR	MNIST	12,000 training samples, 2,000 test samples, 784 features, class = 2
SVM	MNIST	data scale same with k-NN, use one-versus-one classification RBF kernel
NB	UCI Nursery	12,960 training samples, 1,296 testing samples, 8 features, 5 classes
DT (ID3)	UCI Covertype	522,000 training samples, 59,012 testing samples, 55 features, 7 classes

and DianNao [5] 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 6, 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). PuDianNao has the same limitation considering its ISA, capable of expressin only the ML benchmarks. TPU is able to processing LSTM and RNN, but only for the testing phase. In contrast, Cambricon defines a total of forty-three 64-bit scalar/control/vector/matrix instructions and is sufficiently flexible to express all 16 networks.

7.2.2 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 16 benchmarks implemented with Cambricon-ML, 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 Figure 20 Cambricon's reduction on code length over other ISAs. On average, the code length of Cambricon is about 5.87×, 6.68×, and 8.35× 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.

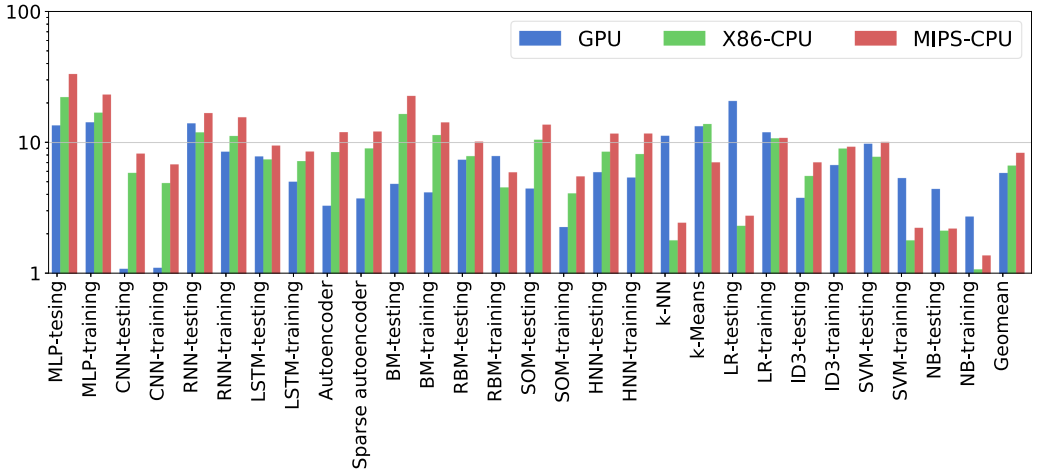

Fig. 20. The reduction of code length against GPU, x86-CPU, and MIPS-CPU for all benchmarks.

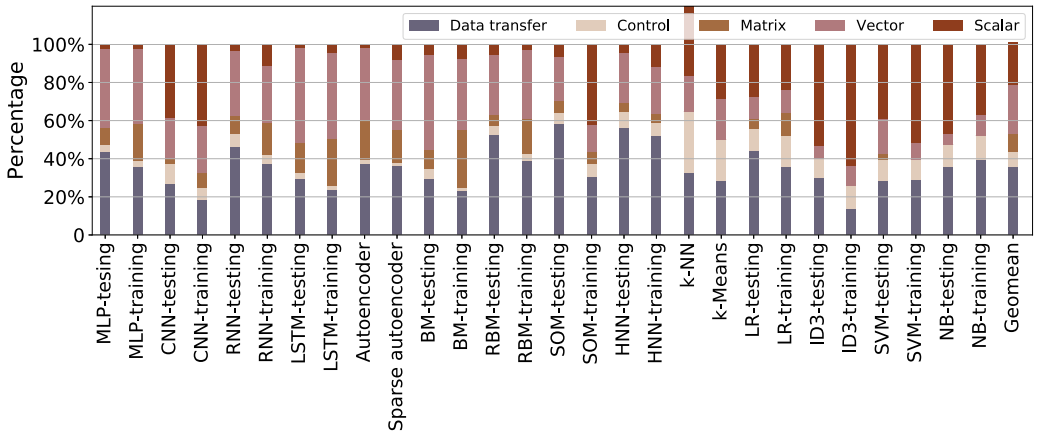

Fig. 21. The percentages of instruction types among all benchmarks.

Specifically, on MLP, Cambricon can improve the code density by 13.62×, 22.62×, and 32.92× 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 1.09×, 5.90×, and 8.27× reduction of code length against GPU, x86, and MIPS, respectively. This is because 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 16 benchmarks (see Figure 21). On average, 35.5% instructions are data transfer instructions, 8.3% instructions are control instructions, 9.1% instructions are matrix instructions, 25.9% instructions are vector instructions, and 22.3% instructions are scalar instructions. This observation clearly shows that vector/matrix instructions play a critical role in NN techniques, and thus efficient implementations of these instructions are essential to the performance of an Cambricon-based accelerator. As for ML algorithms, although the percentages of scalar and control instructions are higher than NN algorithms due to the more complex control flow, vector and matrix instructions still contribute

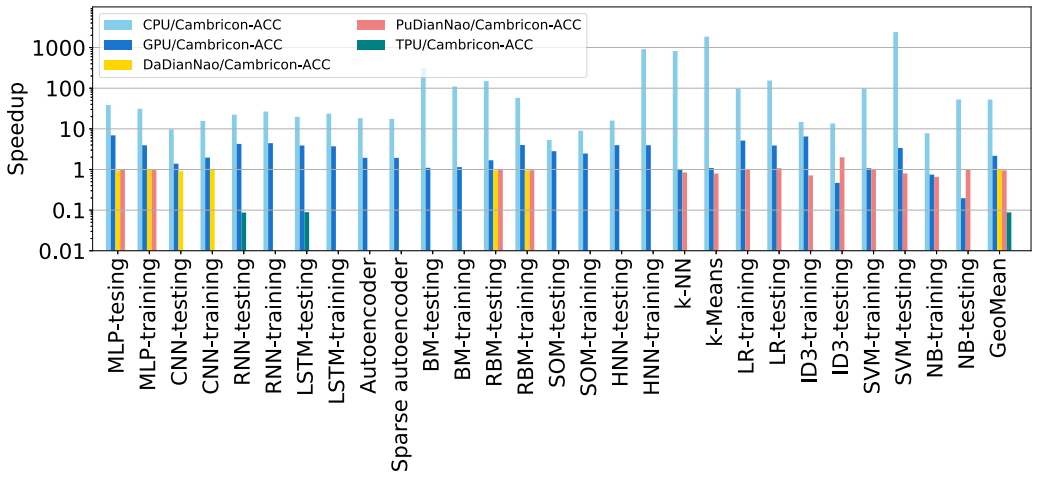

Fig. 22. The speedup of Cambricon-ACC against x86-CPU, GPU, DaDianNao and PuDianNao, TPU.

11.3% on average. We observe that ML benchmarks have higher percentages of control and scalar operations due to their more complex computational pattern.

7.2.3 Performance. We compare Cambricon-ACC against x86-CPU and GPU on all 16 benchmarks listed in Table 6 and Table 7. Figure 22 illustrates the speedup of Cambricon-ACC against x86-CPU, GPU, DaDianNao, and PuDianNao. On average, Cambricon-ACC is about 52.54× and 2.17× faster than x86-CPU and GPU, respectively. This is not surprising, because Cambricon-ACC integrates dedicated functional units and scratchpad memory optimized for ML techniques.

We note that Cambricon performs worse than GPU for ID3-testing and NB-testing, as most computations in these two benchmarks are sequential scalar computations, which can not be accelerated by the vector/matrix instructions in Cambricon.

However, due to the incomplete and restricted ISA, DaDianNao and PuDianNao can only accommodate 3 (i.e., MLP, CNN, and RBM) and 8 (i.e., MLP, RBM, *k*-NN, *k*-means, LR, ID3, SVM, and NB) of the 16 benchmarks, and thus their 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, and the corresponding performance loss is therefore negligible.

Cambricon-ACC is approximately 5.1% slower than PuDianNao. There are two reasons for such performance loss. First, Cambricon-ACC uses low-level instructions to perform the same functionality that could be achieved by an single VLIW instruction in PuDianNao, leading to pipeline bubbles. Moreover, the loss is exacerbated by the inherent complexity in ML techniques, which often require more instructions and loop nests to accomplish than the NN techniques. Second, Cambricon-ACC contains fewer numbers of some types of operators. We scale the PuDianNao according to the number of data an MLU is capable of processing, which is 16 in the original version and 32 in the reimplemented version. Each MLU in the original PuDianNao is equipsped with

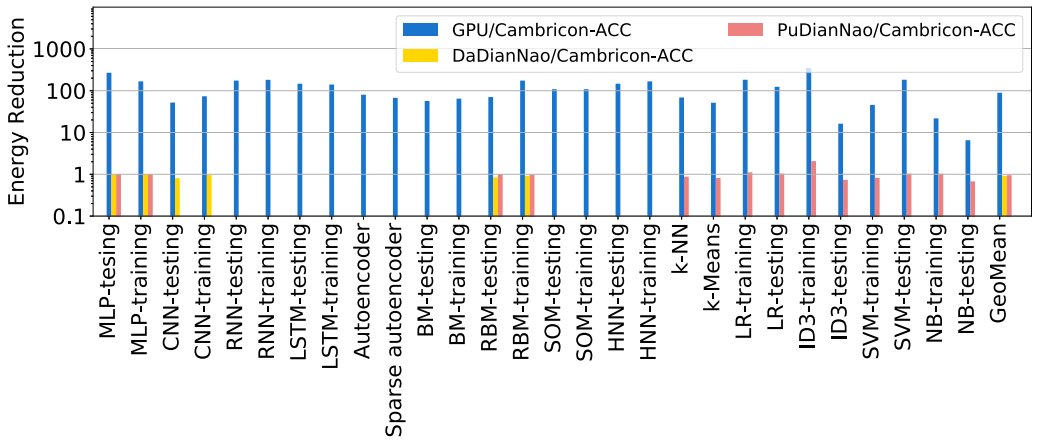

Fig. 23. The energy reduction of Cambricon-ACC over GPU, DaDianNao, PuDianNao.

16 counters; therefore, it is only reasonable to include 32 counters in the reimplemented version, which means that PuDianNao is able to count 32 × 32 data in a cycle. However, counting operation in Cambricon-ACC is a vector operation that process 32 data in a cycle. Therefore, theoretically the counting operation in Cambricon-ACC is 32 × slower than that in PuDianNao. This leads to performance losses in techniques that contain large amounts of counting, i.e., 0.71 × and 0.66 × performance ratios for ID3-training and NB-training, respectively.

We note that Cambricon-ACC outperform PuDianNao in ID3-testing as most of the computation in ID3-testing are scalar operations and irregular memory access. After the tree model and a mini-batch of samples are loaded to the on-chip memory, one value of a specific attribute in the sample will compares with the node in the decision tree. Cambricon is more efficient processing such tasks due to two reasons. First, scalar computation in Cambricon is more efficient than PuDianNao due to the reorder buffer. Second, Cambricon can efficiently obtain the the address of irregular data accessing by using scalar operations and general registers, which is not well supported in PuDianNao.

Cambricon achives 0.089x performance on average compared with TPU, with approximate 0.015× processing units, due to the small network scale that limits the efficiency of TPU.

7.2.4 Energy Consumption. We also compare the energy consumptions of Cambricon-ACC, GPU, DaDianNao, and PuDianNao, which can be estimated as products of power consumptions (in watts) and the execution times (in seconds). The power consumption of GPU is reported by the NVPROF, and the power consumptions of DaDianNao, PuDianNao, 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 [6], which well complements our experiments.

As shown in Figure 23, the energy consumptions of GPU, DaDianNao, and PuDianNao are 89.31×, 0.92×, and 0.95× that of Cambricon-ACC, respectively, where the energy of DaDianNao is averaged over 3 benchmarks, because it can only accommodate 3 of 10 benchmarks, and PuDianNao is averaged over the supported 7 benchmarks.

To illustrate the sources of efficiency, we breakdown the energy costs of Cambricon-ACC and compare against the GPU with our best estimation. Due to the lack of tools to profile GPUs' energy,

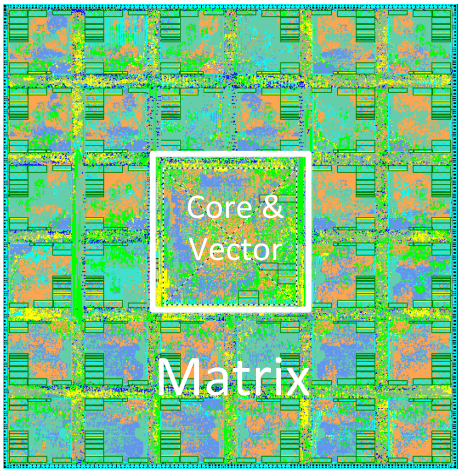

Fig. 24. The layout of Cambricon-ACC, implemented in TSMC 65nm technology.

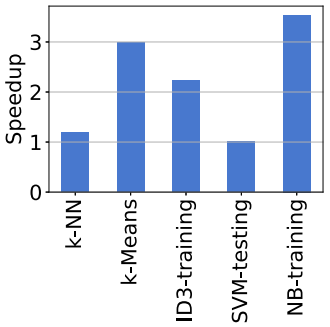

Fig. 25. Speedup of Cambricon-ML over Cambricon-NN.

we are unable to compare against GPUs directly; instead, we compare Cambricon-ACC to itself and GPU with our best effort. The energy savings, which are because of the customized design specially from neural networks, mainly come from three aspects when compared against GPUs. First, Cambricon-ACC achieves a smaller number of executed instructions, i.e., 5.70×, reducing dynamic power of instruction pipelines. Second, Cambricon-ACC leverages functional units that are customized to neural networks, using 16-bit fixed-point operators instead of 32-bit floating point operators. As a 16-bit fixed-point operator is 7.33× more energy efficient than 32-bit floating-point operator, and Cambricon-ACC achieves 13.98× more energy savings roughly than K40 GPU (2.25Tops/s vs. 4.29Tops/s) on the computing units. Also the 16-bit data reduce the total accessed data size to half, as well as the energy. Third, Cambricon-ACC enables efficient data movements with programmer visible control, improving data reuse and reducing the memory accesses for significant energy savings.

However, 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. The case is similar for PuDianNao: Energy consumed by the extra counters and ALUs of PuDianNao compensates the energy coming from instruction pipeline logic, memory queue, as well as the vector transcendental function unit.

We also compare Cambricon-ACC to a state-of-the-art GPU (NVIDIA Tesla V100) using three large networks, i.e., AlexNet, VGG16, and ResNet152. With only 1% peak performance compared to V100, Cambricon is able to achieve an actual performance and energy efficiency up to 34.1% and 4.8%, which clearly shows the efficiency of the architecture of Cambricon while processing NN techniques.

7.2.5 Chip Layout. We show the layout of Cambricon-ACC in Figure 24 and list the area and power breakdowns in Table 8. The overall area of Cambricon-ACC is 57.18mm^2, which is about 3.3% larger than of DaDianNao (55.34mm^2, re-implemented version), and about 3.36% smaller than PuDianNao (59.10mm^2, re-implemented version). The combinational logic (mainly vector and

Table 8. Layout Characteristics of Cambricon-ACC (1GHz) Implemented in TSMC 65nm Technology

Component	Area (μm^2)	(%)	Power (mW)	(%)
Whole Chip	57,181,693	100%	1721.47	100%
Core & Vector	6,003,293	10.49%	164.91	9.58%
Matrix	35,259,840	61.66%	1004.81	58.37%
Chanel	1,5918,660	27.84%	551.75	32.05%
Combinational	19,021,000	33.27%	502.84	29.21%
Memory	8,461,445	14.79%	174.14	10.12%
Registers	5,612,851	9.81%	300.29	17.44%
Clock network	877,360	1.53%	744.20	43.23%
Filler Cell	23,207,862	40.58%		

matrix functional units) consumes 33.26% area of Cambricon-ACC, and the on-chip memory (mainly vector and matrix scratchpad memories) consumes about 14.76% area.

The matrix part (including the matrix function unit and the matrix scratchpad memory) accounts for 61.66% 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 10.49% area. The remaining 27.84% area is consumed by the channel part, including wires connecting the core-and-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 1.721W (under 100% toggle rate), which is only about 1% of the K40M GPU. More specifically, the core-and-vector part and matrix part consume 9.57%, and 58.37% power, respectively. Moreover, data movements in the channel part consume 32.05% power, which is several times higher than the power of the core-and-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. Note that compared to the accelerator in our previous work [40], only about 1.67% additional area and 1.56% additional power are consumed, which are primarily caused by the instruction decoder.

7.2.6 Cambricon-NN vs. Cambricon-ML. Figure 25 shows the speedup of Cambricon-ML over Cambricon-NN of five benchmarks that can benefit from the appended instructions. For NB-training, counting operations that initially implemented by vector comparison and dot are replaced by counting instructions and gain a significant speedup of 3.54×. ID3-training uses *filter* instruction to split nodes under conditions and use *count* instructions to accelerate counting and achieves a speedup of 2.24×. *k*-NN and *k*-means leverage the VARGMIN/VARGMAX instructions to accelerate the computation of *k-sort-by-key* and *argmin/argmax* and achieves speedup of 1.19× and 2.99×. We observe that Cambrcon-ML shows few improvements to Cambricon-NN on SVM-testing. The reason is that most of the computations in SVM-testing are already supported in Cambricon-NN. The only exception is the argmax instruction used to compute the error rate while testing, which accounts for a small portion of the entire computation time.

7.3 Evaluating the Programming Method

In this section, we evaluate the programming model proposed in Section 6 from two perspectives: performance and developing efficiency. Section 7.3.1 shows the effect of two proposed optimizations, improving parallelism and the dynamic reuse strategy. Section 7.3.2 shows the code length

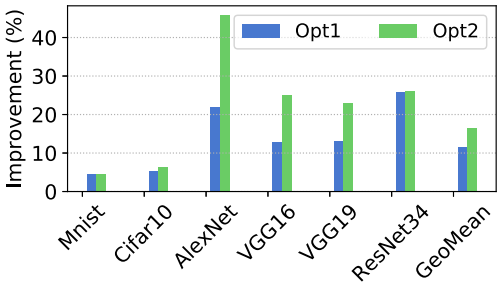

Fig. 26. Improvements of entire networks.

reduction of programming on large networks by using Cambricon-AL, indicating the developing efficiency of Cambricon-AL.

A development trend of ML techniques, particularly NN techniques, is that their network topologies are becoming larger and more sophisticated, which leads to the need for programming supports. In this section, we evaluate our proposed programming method, i.e., Cambricon-AL and Cambricon-ASM by using one medium size network, i.e., the Cifar10 quick model [33], and four large networks, i.e., AlexNet [34], VGG16 [60], VGG19 [60], and ResNet34 [25], which are the state-of-the-art networks recently proposed. Here, we evaluate the inference phase of these networks. To do the experiments, we first implement all *blocks* based on the programming approach introduced in Section 6 and then use these optimized *blocks* to construct the networks. For comparison, we implement a naive version by manually writting Cambricon instructions, which does not include any optimizations. We will focus the evaluation on two aspects, developing efficiency and executing efficiency.

7.3.1 Executing Efficiencies. We evaluate the two optimizations introduced in Section 6.3 in this section. Our baseline is a sequentially executed implementation, and all layers use output reuse strategy. We first apply *Opt1*, i.e., only parallelism optimization is applied, and then apply *Opt2*, where both parallelism and dynamic data reuse strategy are applied. The result is shown in Figure 26. *Opt1* improves the performance by 11.5% on average, and after applying the DR strategy, the improvement is up to 16.5%. For different networks, the effects of these two optimizations are different. For example, AlexNet is heavily improved by using the dynamic reuse strategy, but ResNet34 seems to have tiny improvements after using the DR strategy. To clarify such observation, we further analyze the speedup of all Conv. and FC. layers in AlexNet and ResNet34. As shown in Figure 27 and Figure 28, the FC. layers, which is an IO-intensive operation, are significantly improved by applying the DR strategy, which reduces the amount of memory accesses. Meanwhile, computational-intensive layers such as convolution may not benefit from such a strategy.

7.3.2 Developing Efficiency. By using Cambricon-AL, code lengths of the inference phase of Cifar10, AlexNet, VGG16, VGG19, and ResNet34 are reduced by 2.09×, 3.48×, 6.94×, 7.98×, and 10.65×, respectively. It is clearly that the deeper the network, the more the code length is reduced. The reduced code length come from two aspects: (1) Instructions that perform the repeatedly invoked computational primitives (e.g., the very commonly used convolutional layer and fully connected layers) are replaced by pre-defined *blocks*, and (2) the code for data management, including data allocation and segmentation, are wrapped in the built-in data declaration statements and memory copy functions.

8 RELATED WORK

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

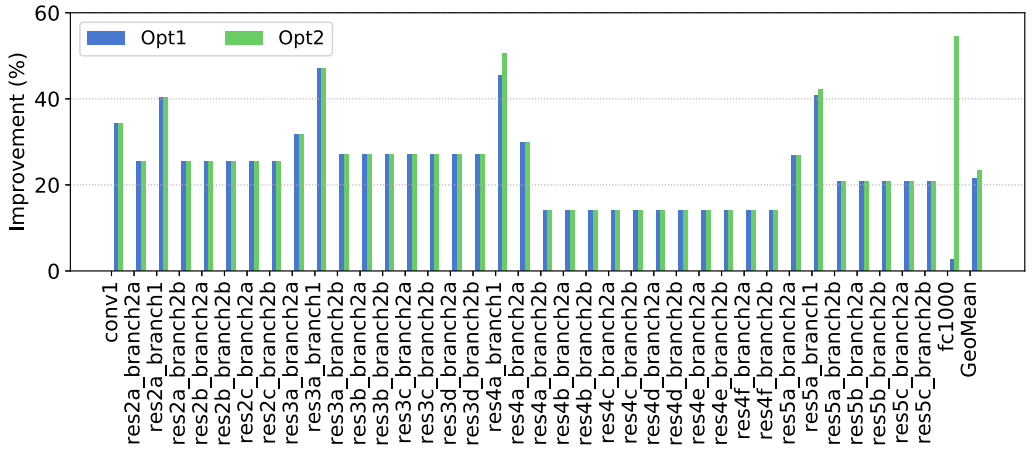

Fig. 27. Improvements of layers in ResNet34.

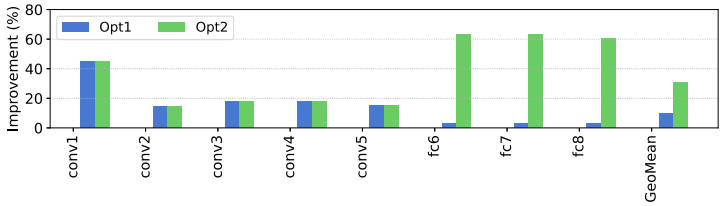

Fig. 28. Improvements of layers in AlexNet.

Machine-learning Techniques. ML techniques, especially NN techniques, have been proven to be efficient in a wide range of tasks. Existing NN techniques have exhibited significant diversity in their network topologies and learning algorithms. For example, Deep Belief Networks (DBNs) [58] consist of a sequence of layers, each of which is fully connected to its adjacent layers. In contrast, Convolutional Neural Networks (CNNs) [34] use convolutional/pooling windows to specify connections between neurons, and 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) [56] 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) [68], the Gibbs sampling algorithm for training Restricted Boltzmann Machines (RBMs) [56], and the unsupervised learning algorithm for training Self-Organizing Map (SOM) [46].

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.

ML 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 [5], 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 [18, 51, 57, 59] or as ASICs [5, 17, 20, 62]. Farabet et al. proposed an accelerator named Neuflow with systolic architecture [17] for the feed-forward paths of CNNs. Maashri et al. implemented another NN accelerator, which arranges several customized accelerators around a switch fabric [41]. Esmaeilzadeh et al. proposed a SIMD-like architecture (NnSP) for Multi-Layer Perceptrons (MLPs) [15]. Chakradhar et al. mapped the CNN to reconfigurable circuits [3]. Chi et al. proposed PRIME [9], 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 [22]. 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 [5]. 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 [7]. Therefore, the application scope of DaDianNao is still limited by its ISA, which is similar to the case of DianNao. In addition to accelerators customized for NN techniques, accelerators designed for classic ML techniques are also proposed [39, 42, 49]. However, these accelerators are also limited in flexibility. 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) [39]. 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 primitives in ML techniques 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.

9 CONCLUSION

In this article, we propose a novel ISA for machine-learning techniques called Cambricon, which allows ML accelerators to flexibly support a broad range of different ML techniques, including NN techniques and classic ML techniques. We compare Cambricon with x86 and MIPS across 10 diverse yet representative NNs and observe that the code density of Cambricon is significantly higher than that of x86 and MIPS. To improve the user-friendliness, we propose an assembly language (Cambricon-AL), an assembler and runtime to help develop Cambricon. We implement a Cambricon-based prototype accelerator in TSMC 65nm technology, and the area is 57.18mm^2, and the power consumption is only 1.721W. Thanks to Cambricon, this prototype accelerator can accommodate all 16 ML techniques, while the state-of-the-art NN and ML accelerator, DaDianNao and PuDianNao, can only support 3 and 8 of them, respectively. Even when executing the partial benchmarks, our prototype accelerator still achieves comparable performance/energy efficiency with the state-of-the-art accelerator with negligible overheads.

REFERENCES

[1] N. S. Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *Am. Stat.* 46, 3 (1992), 175–185.

[2] L. Breiman, J. H. Friedman, R. A. Olshcn, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth International Group, Belmont CA.

[3] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.

[4] 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. 2014. 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*.

[5] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. 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*.

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2015. A high-throughput neural network accelerator. *IEEE Micro* 35, 3 (2015), 24–32.

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

[8] Ping Chi, Wang-Chien Lee, and Yuan Xie. 2016. Adapting B-plus tree for emerging nov-volatile memory based main memory. In *Proceedings of the IEEE Conference on Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD'16)*.

[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. 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'16)*.

[10] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng. 2013. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning*.

[11] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Mach. Learn.* 20, 3 (1995), 273–297.

[12] G. E. Dahl, T. N. Sainath, and G. E. Hinton. 2013. 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*.

[13] A. L. Edwards. 1984. An introduction to linear regression and correlation. *Math. Gaz.* 69, 2 (1984), 1–17.

[14] V. Eijkhout. 2011. Introduction to High Performance Scientific computing. Retrieved from www.lulu.com.

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

[16] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 IEEE/ACM International Symposium on Microarchitecture*.

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

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

[19] E. W. Forgy. 1965. Cluster analysis of multivariate data : Efficiency versus interpretability of classifications. *Biometrics* 21, 3 (1965), 41–52.

[20] V. Gokhale, Jonghoon Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*.

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

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

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the International Conference on Computer Vision*. 1026–1034.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778. DOI: https://doi.org/10.1109/CVPR.2016.90

[26] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management*.

[27] INTEL. [n.d.]. AVX-512. Retrieved from https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[28] INTEL. [n.d.]. MKL. Retrieved from https://software.intel.com/en-us/intel-mkl.
[29] Pineda Fernando J. 1987. Generalization of back-propagation to recurrent neural networks. *Phys. Rev. Lett.* (1987), 602–611.
[30] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. 2009. What is the best multi-stage architecture for object recognition? In *Proceedings of the 12th IEEE International Conference on Computer Vision*.
[31] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 1–12. DOI: https://doi.org/10.1145/3079856.3080246
[32] V. Kantabutra. 1996. On hardware for computing exponential and trigonometric functions. *IEEE Trans. Comput.* 45, 3 (1996), 328–339. DOI: 10.1109/12.485571
[33] A. Krizhevsky. [n.d.]. cuda-convnet: High-performance c++/cuda implemen- tation of convolutional neural networks.
[34] Alex Krizhevsky, Sutskever Ilya, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*.
[35] Pat Langley, Wayne Iba, and Kevin Thompson. 1992. An analysis of bayesian classifiers. In *Proceedings of the 10th National Conference on Artificial Intelligence*. 223–228.
[36] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*.
[37] Q.V. Le. 2013. Building high-level features using large scale unsupervised learning. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*.
[38] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, Vol. 86. 2278–2324. DOI: 10.1109/5.726791
[39] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. 2015. PuDianNao: A polyvalent machine learning accelerator. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*.
[40] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. 2016. Cambricon: An instruction set architecture for neural networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. 393–405. DOI: https://doi.org/10.1109/ISCA.2016.42
[41] A. A. Maashri, M. DeBole, M. Cotter, N. Chandramoorthy, Yang Xiao, V. Narayanan, and C. Chakrabarti, C. 2012. Accelerating neuromorphic vision algorithms for recognition. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*.
[42] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 14–26. DOI: https://doi.org/10.1109/HPCA.2016.7446050
[43] G. Marsaglia and W. W. Tsang. 2000. The ziggurat method for generating random variables. *J. Stat. Softw.* 5, 8 (2000). https://EconPapers.repec.org/RePEc:jss:jstsof:v:005:i08.
[44] 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. 2014. A million spiling-neuron interated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.
[45] 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. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
[46] M. A. Motter. 1999. Control of the NASA langley 16-foot transonic tunnel with the self-organizing map. In *Proceedings of the 1999 American Control Conference*.

[47] NVIDIA. [n.d.]. CUBLAS. Retrieved from https://developer.nvidia.com/cublas.
[48] C. S. Oliveira and E. Del Hernandez. 2004. 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.*
[49] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-out acceleration for machine learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17).* 367–381. DOI: https://doi.org/10.1145/3123939.3123979
[50] David A. Patterson and Carlo H. Sequin. 1981. RISC I: A reduced instruction set VLSI computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture.*
[51] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. 2013. Memory-centric accelerator design for convolutional neural networks. In *Proceedings of the 31st IEEE International Conference on Computer Design.*
[52] John C. Platt, Nello Cristianini, and John Shawe-Taylor. 1999. Large margin DAGs for multiclass classification. In *Proceedings of the Advances in Neural Information Processing Systems 12 (NIPS'99).* 547–553. http://papers.nips.cc/paper/1773-large-margin-dags-for-multiclass-classification
[53] Matt Poremba, Tao Zhang, and Yuan Xie. 2016. Fine-granularity tile-level parallelism in non-volatile memory architecture with two-dimensional bank subdivision. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'16).*
[54] J. R. Quinlan. 1986. *Induction of Decision Trees.* Kluwer Academic Publishers, Amsterdam. 81–106.
[55] J. Ross Quinlan. 1996. Bagging, boosting, and C4.5. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference, (AAAI '96 and IAAI'96).* 725–730.
[56] R. Salakhutdinov and G. E. Hinton. 2012. An efficient learning procedure for deep boltzmann machines. *Neur. Comput.* 24, 8 (2012), 1967–2006.
[57] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors.*
[58] R. Sarikaya, G. E. Hinton, and A. Deoras. 2014. Application of deep belief networks for natural language understanding. *IEEE Trans. Aud. Speech Lang. Process.* 22, 4 (2014), 778–784.
[59] P. Sermanet and Y. LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *Proceedings of the 2011 International Joint Conference on Neural Networks.*
[60] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations.* http://arxiv.org/abs/1409.1556
[61] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the Computer Vision and Pattern Recognition.* 1–9.
[62] O. Temam. 2012. A defect-tolerant accelerator for emerging high-performance applications. In *Proceedings of the 39th Annual International Symposium on Computer Architecture.*
[63] V. Vanhoucke, A. Senior, and M. Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS'11).*
[64] Yu Wang, Tianqi Tang, Lixue Xia, Boxun Li, Peng Gu, Huazhong Yang, Hai Li, and Yuan Xie. 2015. Energy efficient RRAM spiking neural network for real time classification. In *Proceedings of the 25th Edition of the Great Lakes Symposium on VLSI.*
[65] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of cross-point resistive memory architectures. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture.*
[66] Tao Xu, Jieping Zhou, Jianhua Gong, Wenyi Sun, Liqun Fang, and Yanli Li. 2012. Improved SOM based data mining of seasonal flu in mainland China. In *Proceedings of the 2012 8th International Conference on Natural Computation.*
[67] Xian-Hua Zeng, Si-Wei Luo, and Jiao Wang. 2007. Auto-associative neural network system for recognition. In *Proceedings of the 2007 International Conference on Machine Learning and Cybernetics.*
[68] Zhengyou Zhang, M. Lyons, M. Schuster, and S. Akamatsu. 1998. Comparison between geometry-based and Gabor-wavelets-based facial expression recognition using multi-layer perceptron. In *Proceedings of the 3rd IEEE International Conference on Automatic Face and Gesture Recognition.*
[69] Jishen Zhao, Guangyu Sun, Gabriel H. Loh, and Yuan Xie. 2013. Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface. *ACM Trans. Arch. Code Optimiz.* 10, 4, Article 24 (Dec. 2013), 25 pages.

Received May 2018; revised April 2019; accepted May 2019