MARSv2: Multicore and Programmable Reconstruction Architecture Using SRAM CIM-Based Accelerator with Lightweight Network

Chia-Yu Hsieh
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan

Shih-Ting Lin
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan

Zhaofang Li
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan

Chih-Cheng Lu
Information and Communication Labs
Industrial Technology Research Institute
Hsinchu, Taiwan

Meng-Fan Chang
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan

Kea-Tiong Tang
Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan

***Abstract*—Computing-in-memory (CIM) systems reduce the degree of large-scale data movement by performing computation on the memory; this avoids a von Neumann bottleneck. Because of its low-power characteristic, CIM has demonstrated great potential for increasing the energy efficiency of edge devices. This paper presents a multicore and programmable reconstruction architecture using static random-access memory (SRAM) CIM-based accelerator with lightweight network. The proposed architecture uses SRAM CIM macro as the processing element, supporting sparse convolutional neural network computing. This architecture achieves 15.16 TOPS/W system energy efficiency and 747.6 GOPS on the CIFAR10 data set.**

Keywords—computing-in-memory, deep learning, sparsity, CNN accelerator

I. Introduction

Deep learning, a type of artificial intelligence, has achieved outstanding performance. Convolutional neural networks (CNNs) have been particularly crucial to image recognition. However, the large numbers of calculations and parameters required in CNNs result in high energy consumption and relatively low inference speed and efficiency, which is unfavorable for edge device applications.

To address these problems, in relation to hardware, studies have optimized the data reusability of the convolution operation. However, use of the traditional von Neumann architecture has continued in these studies. In this architecture, the computing and memory units are separate; thus, all operations require data to be transferred between the memory and computing units. This consumes a substantial amount of energy and time and is referred to as the von Neumann bottleneck. To solve this problem, a computing-in-memory (CIM) [1],[2],[4],[9],[10],[11] architecture has been proposed.

With regard to software, CNNs require relatively few parameters and calculations employing two algorithms—quantization and pruning— to compress the network model. Quantization reduces the bit width of the parameters, and pruning reduces the number of redundant parameters in the network. One study [5] proposed the operation and structure of a CIM macro by using software and hardware co-design to reduce the number of parameters through structure pruning. In addition to removing redundant weights in the network, making the network sparser, this approach also enables the zeros in the network to be arranged regularly after the pruning, enables the hardware to use the sparsity-aware mechanism, means that zero-valued data do not need to be calculated, and reduces the time and energy consumed. Nevertheless, challenges still exist. First, the system is unable to store all the weights that have limited on-chip static random-access memory (SRAM) CIM after model compression. Consequently, the weights must be continually updated during the calculation. CIM has the advantage of high parallelism in the calculation mode, and multiple word lines can be activated simultaneously. However, the write weight mode is limited by the write bandwidth, and usually, only one word line can be activated at a time. Updating the weight leads to high memory access latency and performance loss. Second, lightweight networks [3] have been proposed, but most of the current CIM accelerators focus on standard convolution-based models, such as VGG16 and ResNet18. The number of parameters in lightweight networks such as MobileNet and ShuffleNet is much smaller than those in models based on standard convolution without model compression. Fewer parameters means considerably less power is consumed by transferring data from dynamic RAM to SRAM CIM.

Considering the aforementioned factors, we propose a reconstruction multicore SRAM CIM CNN accelerator in this paper. Our contributions are summarized as follows:

- Multicore and programmable reconstruction SRAM CIM with an intercore parallel and intercore pipeline computation architecture is proposed.
- A mapping method and data flow are used that combine depth-wise separable convolution with a sparsity-aware mechanism.
- A core search arbiter with an asynchronous first-in-first-out (FIFO) design for the multicore data transfer router is proposed.

II. Preliminaries

In this paper, we adopt the pruning method used by [5]. A state-of-the-art 6T-SRAM chip [2] with 56.7 GOPS and 23.26 TOPS/W for 4-bit input × 8-bit weight per macro is adopted in our hardware architecture design (Fig. 1). Each CIM macro contains eight partitions, and each partition can store 64 groups of 16 × 8-bit weights. The eight partitions share the same 16 × 4-bit inputs and perform eight sets of 16-vector inner products. The two SRAM CIM macros are combined into a CIM core in our architecture so that each cycle can perform 16 sets of 16-vector inner products. The 16 × 8-bit weights in channel order are defined as a weight group

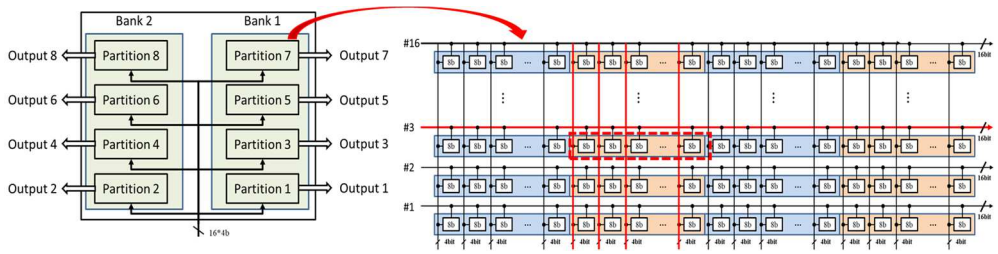

Fig. 1. SRAM CIM

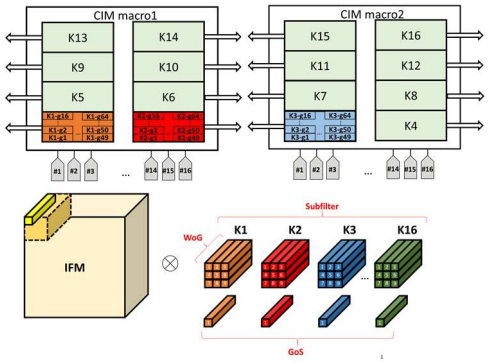

Fig. 2. Convolution behavior for CIM

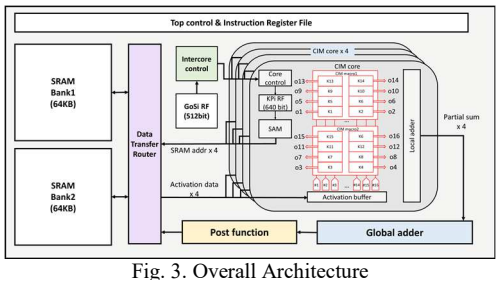

Fig. 3. Overall Architecture

(WG), and the 16 WGs are defined as a group of sets (GoS). Each set of 16 consecutive kernels is called a subfilter (Fig. 2), and the sparse distribution of WGs in a subfilter is consistent with the algorithm described later. Because the WGs stored in the SRAM CIM are not in order in the pruning algorithm, the system requires an index code to determine their relative position in the original kernel ([5] provides a detailed description).

III. Proposed Architecture

Fig. 3 depicts the overall CIM accelerator architecture, which consists of two global SRAM banks, four CIM cores, a clock domain crossing data transfer router, a GoS number index code register file (GoSi RF), a quantization factor register file, an instruction register file with a top controller, a global adder, and a post-function module. Each CIM core contains two SRAM CIM macros, a position index code register file, a local adder, and a sparsity-aware module (SAM). The two global SRAM banks are the primary storage for the input and output feature maps. The data stored in the SRAM is accessed using the four CIM cores simultaneously through the router.

The GoSi RF stores the number of GoSs for every subfilter in a layer to determine how many CIM cores should be used. In this study, the architecture design uses four cores because the maximum number of GoSs in a subfilter requires four cores for data storage. If the maximum number of GoSs in a subfilter requires more cores, the core number can be increased. In each core, the SAM accesses the feature map data through a position index code to convolute and accumulate the partial sum in the local adder (intracore accumulation). Subsequently, the data are sent to the global adder to accumulate the partial sum from different cores (intercore accumulation). Finally, the data are sent to the post-function module for quantization and pooling, then to the rectified linear unit, if necessary, and then back to the SRAM.

A. Intercore Parallel and Pipeline Computation

As already mentioned, the number of CIM cores determines the maximum number of GoSs in a subfilter that can be stored in a network. Therefore, we use subfilters as a unit of calculation. Fig. 4 illustrates the intercore parallel computation, in which the weights of a subfilter are scattered across four CIM cores for storage. When the convolution is begun, the four cores access the feature map SRAM data through the SAM in accordance with the corresponding index code stored in each core. After each core has accumulated its partial sum in the local adder, the data are sent to the global adder for intercore accumulation. After completing a pixel calculation, the sliding window scans to the next pixel.

However, because the sparsity rate of each layer is different, not every subfilter requires four cores to store the weights. As presented in Fig. 5, the subfilter uses only two cores. To improve core utilization and reduce the impact of the weight reload time, the core containing the weights of the first subfilter performs calculations in parallel. When subfilter1 is being calculated, the weights of subfilter2 are written into the remaining cores simultaneously. The calculations between the cores that store different subfilters can then become intercore pipeline computations. In our design, the cores are reconstructed flexibly to determine whether parallel or pipeline computation should be used with a particular core, using programmable instruction and control signals to reduce the weight reload time.

Fig. 6 illustrates how the programmable control signal controls the CIM core array. When the weight of a subfilter has been loaded, a four-bit signal would be input to the intercore control block. The intercore control block would decode the four-bit signal to tell the CIM core array the reconstruction information. Each bit means whether the core needs to calculate in this subfilter. For example, 4'b0111 represents that the core1, core2, and core3 calculate with intercore parallel computation for a subfilter. Then, core4 can store the next subfilter with intercore pipeline when core1, core2, and core3 are calculating.

B. Mapping Method for Depthwise separable Convolution

As mentioned earlier, the popular lightweight network use depth-wise separable convolution [3] rather than standard convolution. Depth-wise separable convolution can be divided into depth-wise and pointwise convolution. The following describes how depth-wise convolution and pointwise convolution operate through a sparsity-aware mechanism without additional control circuits being required.

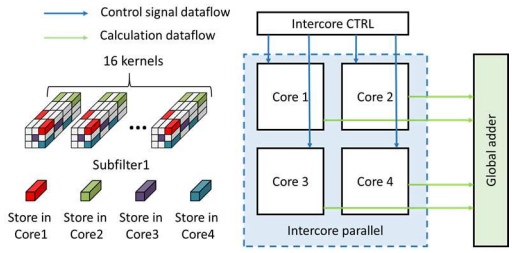

Fig. 4. Intercore parallel computation

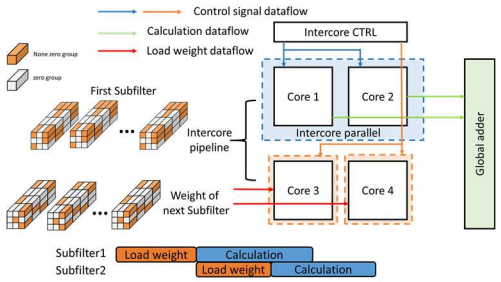

Fig. 5. Intercore pipeline computation

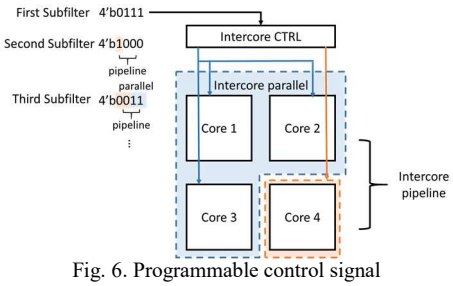

Fig. 6. Programmable control signal

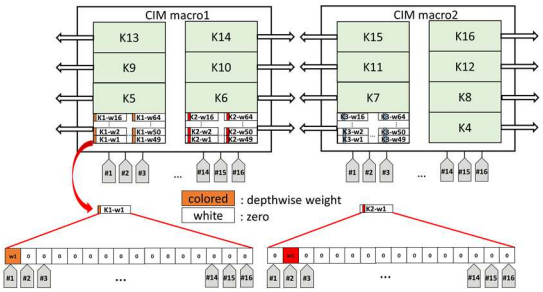

Fig. 7. Mapping method for depthwise convolution

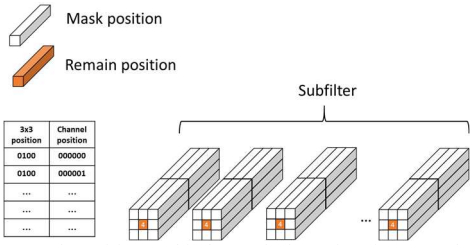

Fig. 8. Mask position and index code for pointwise convolution

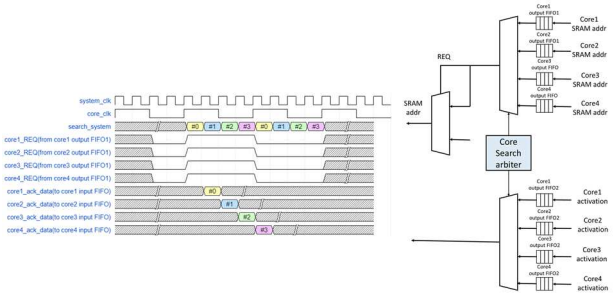

Fig. 9. Core search arbiter with asynchronous FIFO

1) *Depthwise Convolution*

As described in Section II, each partition of the CIM macro can store 64 groups of 16 × 8-bit weights, and the eight partitions share the same 16 × 4-bit inputs and perform eight sets of 16-vector inner products (16 MAC). However, depth-wise convolution does not accumulate in channel order. Therefore, for 16 MAC, we only use 1 MAC, and the weights of the other 15 MAC are stored as 0. To obtain the correct value, the weights with the same channel are mapped to the address relative to the corresponding input data, and the two macros correspond to the 16 channels (Fig. 7).

2) *Pointwise Convolution*

Pointwise convolution is a 1 × 1 standard convolution. Only the periphery of the 3 × 3 kernel should be regarded as zero, and the WG should retain a position index of 3 (Fig. 8).

C. *Core Search Arbiter with Asynchronous FIFO*

As mentioned, we employ two feature map SRAM banks, one each for the input and output feature maps. When the number of GoSs of a subfilter means that more than one core is required for storage, multiple cores access the data from one SRAM bank simultaneously; however, the input and output of the SRAM bank only enable one core to access the data. If only one core is used at a time, the other cores are in idle state. To address this, in addition to the input and output FIFO group of the four CIM cores in our router, an additional core search arbiter is designed. As illustrated in Fig. 9, the operating frequency of the router and SRAM is N times the operating frequency of the CIM core, with N being the number of CIM cores. In a cycle of a core operating frequency, the core search arbiter counts from 0 to N – 1. When the arbiter is 0, whether the output FIFO1 of the first core has an address require signal is determined. If it does, the data are accessed from the SRAM by using the address. When the core search arbiter counts to 1, the output FIFO1 of the second core is determined, and so on. In output FIFO2, the value calculated for each core and using the SRAM address is stored.

IV. Experiment Results

In Fig. 10, we compare the average throughput of MARS[5] and MARSv2, our proposed architecture, and adopt the same core number and sparsity-aware mechanism as do the VGG16 and ResNet18 models using CIFAR10 and CIFAR100 data sets. MARSv2 demonstrates 1.68× and 1.532× performance gain on VGG16 and ResNet18 with CIFAR10. For CIFAR100, the improvement can be 1.808x and 1.498x on VGG16 and ResNet18, respectively. The difference between MARS and MARSv2 depicts in Fig. 11.

Table I compares the work proposed in this study with other state-of-the-art SRAM CIM accelerators. The data were obtained from synthesis results obtained using a Synopsys Design Compiler. The power of the CIM macro was estimated by referring to the measurement result in [2], and the system power was estimated using PrimetimePX.

Our proposed system achieved 747.6 GOPS and 15.16 TOPS/W in terms of system energy efficiency with CIFAR10 data set. For ImageNet, the system energy efficiency was 3.69 TOPS/W with VGG16 and 3.92 TOPS/W with MobileNet. For the figure of merit that considers energy efficiency, and the operation bit width, our proposed system outperforms [6]

Table I
Comparison with prior arts

	ISSCC20[6]	VLSI20[7]	ISSCC21[8]		MARS[5]	This work		
Technology	65nm	65nm	65nm		65nm	65nm		
Application	CNN/FC	CNN	CNN/FC		CNN	CNN		
Area	9.00 mm^2 (system)	7.57 mm^2 (system)	12.0 mm^2 (system)		6.52 mm^2 (system)	9.3 mm^2 (system)		
Supply Voltage	0.9-1.05V	1.0V	0.62-1.0V		0.9-1.1V	0.9-1.1V		
Single CIM size	64Kb	38Kb	16Kb		64Kb	64Kb		
Activation Bit-width	4/8	1-16	1-8		4-8	4-8		
Weight Bit-width	4/8	1-16	1-8		8	8		
Sparsity Support	Activation/ Weight	Weight	Weight		Weight	Weight		
Data set	CIFAR10	-	CIFAR10	ImageNet	CIFAR10	CIFAR10	ImageNet	ImageNet
Application	VGG16	VGG16	VGG16	ResNet18	VGG16	VGG16	VGG16	MobileNet
Sparsity Compression	20x	10x	4.9x	4.2x	20x	20x	12.5x	4.0x
Bit-width	W8A4	W1A16	W4A4	W4A8	W8A4	W8A4	W8A8	W8A4
Average Throughput (GOPS)	49.86~ 59.96	260.4	-	-	268.6	747.6	181.105	193.396
Energy Efficiency (TOPS/W)	13.12 (macro) 2.96 (system)	49.12 (macro)	16.73 (macro) 6.20 (system)	7.32 (macro) 2.75 (system)	49.74 (macro) -	51.50 (macro) 15.16 (system)	11.03 (macro) 3.69 (system)	8.95 (macro) 3.92 (system)
FOM[*1]	94.72	785.92	99.2	88	-	485.12	236.16	125.44

FOM[*1] = Activation bit-width * Weight bit-width * Energy efficiency

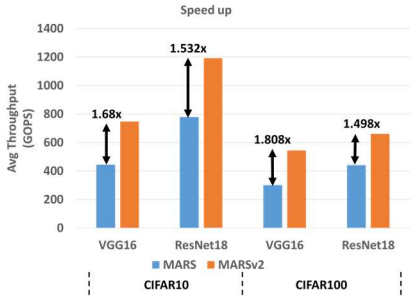

Fig. 10. Improvement on CIFAR10 and CIFAR100

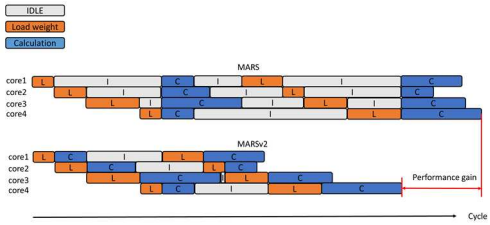

Fig. 11. Difference between MARS and MARSv2

others by up to 5.12×, with a 4.89× improvement on [8]. The improvement is mainly the result of skipping zeros and multicore parallel and pipeline computation.

V. Conclusion

In this study, we proposed a multicore and programmable reconstruction architecture using SRAM CIM-based accelerator with lightweight network. This architecture can reconstruct the cores flexibly to establish CIM cores through either parallel or pipeline computation by using programmable instruction and a control signal, supporting depthwise separable convolution for a lightweight network.

References

[1] K. -T. Tang et al., "Considerations of Integrating Computing-In-Memory and Processing-In-Sensor into Convolutional Neural Network Accelerators for Low-Power Edge Devices," 2019 Symposium on VLSI Technology, 2019, pp. T166-T167.

[2] X. Si et al., "15.5 A 28nm 64Kb 6T SRAM Computing-in-Memory Macro with 8b MAC Operation for AI Edge Chips," 2020 IEEE International Solid-State Circuits Conference - (ISSCC), 2020, pp. 246-248.

[3] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[4] Wei-Chen Wei et al., "A Relaxed Quantization Training Method for Hardware Limitations of Resistive Random-Access Memory (ReRAM)-based Computing-In-Memory", IEEE Journal on Exploratory Solid-State Computational Devices and Circuits (JXCDC), vol. 6 (1), pp. 45-52, June 2020.

[5] S. -H. Sie et al., "MARS: Multi-macro Architecture SRAM CIM-Based Accelerator with Co-designed Compressed Neural Networks," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[6] J. Yue et al., "14.3 A 65nm Computing-in-Memory-Based CNN Processor with 2.9-to-35.8TOPS/W System Energy Efficiency Using Dynamic-Sparsity Performance-Scaling Architecture and Energy-Efficient Inter/Intra-Macro Data Reuse," 2020 IEEE International Solid-State Circuits Conference - (ISSCC), 2020, pp. 234-236.

[7] J. -H. Kim et al., "Z-PIM: An Energy-Efficient Sparsity Aware Processing-In-Memory Architecture with Fully-Variable Weight Precision," 2020 IEEE Symposium on VLSI Circuits, 2020, pp. 1-2.

[8] J. Yue et al., "15.2 A 2.75-to-75.9TOPS/W Computing-in-Memory NN Processor Supporting Set-Associate Block-Wise Zero Skipping and Ping-Pong CIM with Simultaneous Computation and Weight Updating," 2021 IEEE International Solid-State Circuits Conference (ISSCC), 2021, pp. 238-240.

[9] Chih-Heng Pan et al., "An Analog Multilayer Perceptron Neural Network for a Portable Electronic Nose", Sensors, 2013, 13, 193-207.

[10] Shih-Wen Chiu et al., "A Fully Integrated Nose-on-a-Chip for Rapid Diagnosis of Ventilator-Associated Pneumonia", IEEE Transaction on Biomedical Circuits and Systems, vol. 8(6), pp. 765-778, 2014.

[11] Zhaofang Li et al., "A Miniature Electronic Nose for Breath Analysis", 2021 IEEE International Electron Devices Meeting (IEDM).