HUBPA: High Utilization Bidirectional Pipeline Architecture for Neuromorphic Computing

Houxiang Ji
Shanghai Jiao Tong University
jihouxiang@sjtu.edu.cn

Li Jiang*
Shanghai Jiao Tong University
ljiang_cs@sjtu.edu.cn

Tianjian Li
Shanghai Jiao Tong University
ltj2013@sjtu.edu.cn

Naifeng Jing
Shanghai Jiao Tong University
sjtuj@sjtu.edu.cn

Jing Ke
Shanghai Jiao Tong University
kejing@cs.sjtu.edu.cn

Xiaoyao Liang
Shanghai Jiao Tong University
liang-xy@cs.sjtu.edu.cn

ABSTRACT

Training Convolutional Neural Networks(CNNs) is both memory- and computation-intensive. The resistive random access memory (ReRAM) has shown its advantage to accelerate such tasks with high energy-efficiency. However, the ReRAM-based pipeline architecture suffers from the low utilization of computing resource, caused by the imbalanced data throughput in different pipeline stages because of the inherent down-sampling effect in CNNs and the inflexible usage of ReRAM cells. In this paper, we propose a novel ReRAM-based bidirectional pipeline architecture, named HUBPA, to accelerate the training with higher utilization of the computing resource. Two stages of the CNN training, forward and backward propagations, are scheduled in HUBPA dynamically to share the computing resource. We design an accessory control scheme for the context switch of these two tasks. We also propose an efficient algorithm to allocate computing resource for each neural network layer. Our experiment results show that, compared with state-of-the-art ReRAM pipeline architecture, HUBPA improves the performance by 1.7× and reduces the energy consumption by 1.5×, based on the current benchmarks.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; *Neural networks*;

KEYWORDS

ReRAM, Neuromorphic Computing, CNN, Edge Computing, On-device Training

ACM Reference Format:
Houxiang Ji, Li Jiang, Tianjian Li, Naifeng Jing, Jing Ke, and Xiaoyao Liang. 2019. HUBPA: High Utilization Bidirectional Pipeline Architecture for Neuromorphic Computing. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287674

*Corresponding Author

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287674

1 INTRODUCTION

Deep Neural Networks, e.g., Convolutional Neural Networks (CNNs) have been widely applied in a broad range of applications. Moreover, with the emergence of semi-supervised learning [4, 10] and reinforce learning [7], superior architectures are expected to *support on device DNN training on the edge* [5, 6]. Conventionally, clusters of multiprocessors and GPGPUs [2, 12] are suitable for the heavy workloads in DNN training while application-specific accelerators [1, 11] are designate to the DNN inference. However, all existing solutions are challenged by the slow-down of Moore's Law and fail to support both training and inference tasks with constrained computing power on edge.

The metal-oxide resistive random access memory (ReRAM) crossbars are capable of in-memory processing, and thereby avoid the huge data movement between the computation units and memories. Moreover, ReRAM-crossbar shows its great potential in DNN acceleration because of its inherent efficiency for matrix-multiplication. Synaptic weights(w) can be encoded as the conductance of ReRAM cells. The multiplication and addition between weight matrix(W) and the input vector (A) execute in a direct and fast way by the bit-line current accumulation. Recent advancement affiliates ReRAM-crossbar with other superior characteristics like high density [18].

Several ReRAM-based pipeline architectures are proposed to support CNN inference and training [15, 17]. However, the computation efficiency and resource utilization in existing ReRAM-based pipeline architecture are degraded by following issues: 1) Considerable amount of computation units are idle in most of the time. The existence of pooling layer and convolutional layer in CNN inevitably incurs the load imbalance among layers and thereby idles the computing resources. These idle hardware are called *Bubbles* for simplicity. The overall hardware utilization is less than 6% in cutting-edge architecture for CNN [15] when executing VGG16, and decreases dramatically as the CNN goes deeper. Although hardware redundancy [17] can alleviate such imbalance, the overhead is not affordable when the CNN goes too deep. 2)Unequal cost for reading and writing on ReRAM cells. The write operation on ReRAM cells is more energy-consuming than the read operation and degrades the endurance. Thus, designers tend to avoid the weight update in ReRAM cells, which also prohibits the share of ReRAM-crossbar among different weight matrices for utilization enhancement.

To alleviate the above challenges, we propose a novel bidirectional ReRAM-based pipeline architecture, named HUBPA, to improve the hardware utilization and the throughput in CNN training. This paper makes the following contributions:

- We design the specific hardware and software control schemes to support dynamic context switching between forward and backward propagation (two basic reverse operations in training) to achieve higher throughput and utilization.
- We propose efficient layer-wise hardware allocation algorithm which supports various neural networks. Moreover, in each layer, the algorithm raises an explicit task switching scheme for every ReRAM crossbar.
- We evaluate HUBPA on several state-of-the-art CNNs, and HUBPA improves the throughput by 1.5× and reduce the energy consumption by 1.6× averagely compared with the state-of-the-art ReRAM-based pipeline architecture architectures.

The rest of the paper is organized as follows: Section 2 introduces the background. Section 3 analyzes the existing architecture and motivates this paper. The architecture design and data flow are presented in Section 4. We show the evaluation result in Section 5 and conclude this paper in Section 6.

2 BACKGROUND

In this section, we illustrate the training processes of CNN in the state-of-the-art ReRAM-based pipeline architectures.

2.1 The process of CNN training

CNN contains convolution layers, pooling layers, and fully connected layers. Rational and dedicated combination of these primary layers builds up the CNN with notable accuracy. The input data flows through layers and generates standard feature maps between layers. The high-level features are extracted by layer-wise convolution and downsampling. Convolution operation consists of basic multiplication and addition. Downsampling occurs in the pooling layer. A 4-to-1 max pooling layer, for example, picks the maximum value from a 2×2 window and send to the next layer [6].

We then present how the CNN training works on the ReRAM-based architecture. The training process includes the Forward Propagation(FP) and Backward Propagation(BP) while the inference process only includes FP. *FP* is a sequence of dot product between weight matrix in layer L (W_L) and vector (i), $y_L = f(\vec{i} \otimes W_L)$. As indicated in Fig. 1, ReRAM crossbar (XB) holds the weight matrix W and performs the dot-product operation.

During training, FP derives output results by pushing input samples through layers. The comparison between labels and output generates error is used to tune the weights in BP, and consequently to minimize the error. *BP* is a sequence of error propagation and weight updates. Loss functions ($\delta_L = f' \circ (y_L - t)$) generates errors ($\delta_L$) in layer L, where t is the existing label. δ_L joins the convolution to generate error in previous layer $L-1$: $\delta_{L-1} = \delta_L \otimes (W_L)^T \circ f'$. The partial derivatives of weight in layer L (ΔW_L) are generated by convolution on the error (δ_L) and intermediate results in layer $L-1$, $\Delta W = y_{L-1} \otimes {\delta_L}^T$. When the CNN is trained in *batch*, the weights can be updated with the average partial derivatives derived in every n (batch size) iterations of BPs.

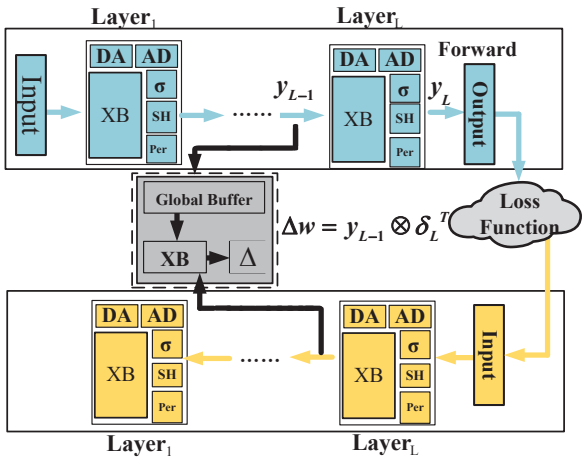

Figure 1: Basic Training flow of Convolutional Neural Networks on ReRAM-based Architecture.

2.2 ReRAM-based Pipeline Architecture

As discussed in section 1, the existence of bubbles significantly degrades the performance. Replication of computing resources, ReRAM crossbar and accessory components like ADC/DAC, alleviates the workload imbalance among layers and thus reduce the bubbles. However, enormous replications are required to eliminate all bubbles, e.g., 180× replications of computing resource for a VGG-16 CNN [17]. The desired replication exponentially increases with the CNN going deeper and broader.

Pipelayer [17] allocates dedicated pipelines for FP and BP, dividing the computing resource spatially to avoid frequent update in ReRAM cells. We named it Space Divided Multiplexing (FP/BP) Propagations (SDMP). In SDMP, FP and BP execute in parallel on their dedicated hardware. Every piece of hardware is only able to support a single task, either FP or BP, all the time. Use a CNN with three conv-layers (L_1 to L_3) as an example. A 4-to-1 max pooling layer attaches each layer. S_i denotes the amount of computing resource assigned to layer i. The blocks under S_i denote the unit of computing resource occupied by layer i. In Fig. 2(a), $S_1 = 8, S_2 = S_3 = 2$. SDMP finishes one FP and one BP in 6 cycles. They execute in parallel, and no computation unit switches between two tasks. In each cycle, four convolutional results are generated in S_1 and enter the pooling layer. Only one result goes to S_2 from the pooling layer. S_3 has waited for 5 idle cycles (bubbles) to derive one result. Moreover, the BP has the same amount of bubbles as it performs the symmetric operations. Note that the BP process in SDMP is one iteration earlier than the FP process.

3 MOTIVATION AND KEY IDEAS

3.1 Time Divided Multiplexing Propagation

We propose an alternative approach named Time Divided Multiplexing (FP/BP) Propagations (TDMP) as a baseline. In TDMP, FP or BP occupies all the computing resource at one time; the hardware can support the context switching between FP and BP. *TDMP* consists of a sequence of FP and BP in the same iteration. TDMP decouples the pipelines of FP and BP in the time domain. The basic

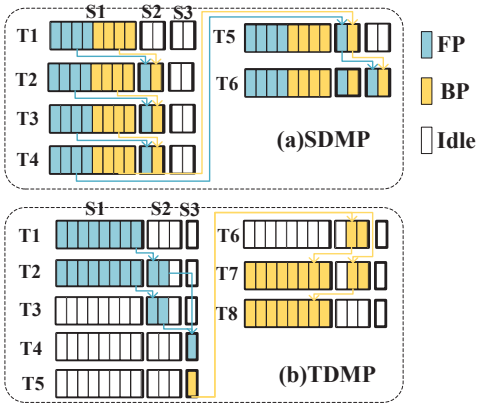

Figure 2: Utilization of limited computing resource in SDMP and TDMP

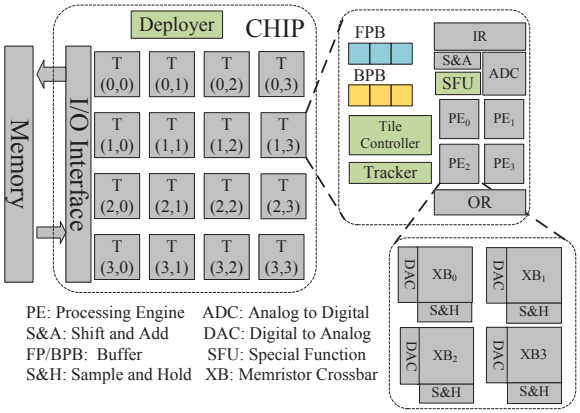

Figure 3: Proposed Architecture

timeline of *TDMP* is shown in Fig. 2(b). The hardware allocation is the same with SDMP example: $S_1 = 8, S_2 = S_3 = 2$. TDMP finished the FP in the first four cycles and symmetrically finished the BP in the last four cycles sequentially. Note that the FP and BP processes belong to the same iteration of training. Compared with SDMP, however, TDMP spends a longer time to accomplish one iteration of training.

3.2 Resource Sharing for Simultaneous Forward/Backward Propagation

To maximize the utilization of these idle computing resource, we propose the resource sharing scheme.

The resource sharing between FP and BP stands on the following fact: the FP and BP computations in the same layer involve a weight matrix and its transposition, respectively. Thus, FP and BP processes in the same layer can share the same computing resource. As we described in Section 1, in the ReRAM crossbar architecture, the matrix transposition can be easily implemented by altering the direction of voltage entrance and current accumulation on the same crossbar(XB). Besides, both FP and BPs can use the same ADCs/DACs, sample-and-hold (S+H), and other peripheral circuits (Per). These are essential basics for our proposed architecture.

In *SDMP*, FP and BP have their dedicated hardware resources. This dedicated hardware allocation incurs many bubbles and prohibits the potential benefits from resource sharing between two propagations. In *TDMP*, FP and BP share the computing resource, but the FP and BP task must execute sequentially. The sequential execution of FP and BP also incurs many bubbles. This idle hardware gives us the opportunity for further improvement. In this work, the computing resource in a single layer can simultaneously carry out both the FP and BP from two subsequent training iterations. The control scheme and devise components are essential to support the resource sharing between FP and BP.

The resource allocation for subsequent training iterations also delimits the overall resources utilization and the performance. It is desired to design an efficient resource allocation algorithm to reduce the bubbles, given the CNN topology and the number of total computing resources.

4 THE PROPOSED HUBPA DESIGN

This section presents the proposed bidirectional ReRAM pipeline architecture followed by the details of the control mechanism for the resource sharing, the software preparation, and the data flow.

4.1 Overall Architecture

Fig. 3 represents our architecture design extended from the ISAAC design [15]. The colored components are our extension. A ReRAM chip composes of multiple tiles (T), connected with an on-chip concentrated mesh (c-mesh). The chip contains a *deployer* to flexibly deploy CNN among tiles and to load the control vectors into the tile controllers. A chip uses the I/O interface to access the off-chip memory and communicate with other chips.

A tile is composed of multiple processing engines (PEs) connected by a shared bus, tile controllers, memory trackers, eDRAM on-chip buffers, and Input/Output registers. A special function unit (SFU) inhabits in each tile, which contains a comparator, multipliers and logic circuits supporting the activation functions of CNNs. Each tile can dynamically allocate the computing resource between FP and BP process. PE is composed of multiple memristor crossbars (XBs), each of which connects a DAC and sample-and-hold circuits (S&H). The chip loads/stores feature maps and gradient maps from/to *off-chip memory* by the I/O interface. The eDRAM buffers temporarily hold the feature maps that are reused in the following convolutional operations. We double the on-chip buffers (FP Buffer, BP Buffer) to support both FP and BP modes. The input *register*, a 1KB SRAM, fetches the data from eDRAM buffer by the shared bus. The PE array then fetches data from the input register by the same shared bus. The crossbars can transfer as much as 2KB data within a 100ns cycle. The designed eDRAM and shared bus thus can afford the maximum bandwidth requirement. The output register is used to accommodate the output results of each convolution layer.

4.2 Control Mechanism

Three control components are devised to support complete CNN executions. A tile is a basic execution unit, on which all resources execute the same propagation process (either FP or BP).

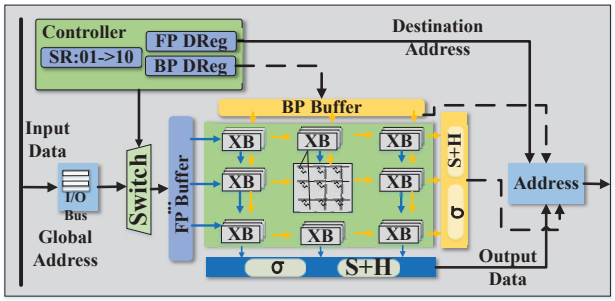

Figure 4: Mode Switch from FP(blue) to BP(orange)

Chip Deployer provides control signals for each tile, by loading the control vector into the tile controller on each tile and maintains the status table for each tile. The status table maintains as many status bits as the number of tiles. Each bit records the status for corresponding tile, idle (0) or occupied (1). The controller fetches the weight matrix according to the destination address maintained in the status table and loads the weight matrix into the ReRAM crossbar of the tile. The chip deployer configures these destination addresses based on the resource deployment algorithm described in the section 4.4.

Tile Controller controls the mode of each tile in the runtime. The tile controller contains a register with two bits to represent the three computation modes of each tile: FP (01), BP (10), idle (00 & 11), and local registers (BP and FP separately) to keep the target eDRAM buffer addresses and different accumulator addresses for different modes. The switches and muxes activated by the control signals from the tile controller make the context switching between different assigned modes. The results from each tile will be delivered to the target tiles based on the stored destination address later.

Memory Tracker: Data synchronization in every cycle is required to keep the correct functionality of the neural network. Memory tracker records read/write memory accesses, and ensures that the access sequence confirms the memory addresses in the tile controller. Memory tracker also tracks the intermediate results stored in each eDRAM buffer. These results are sent to the pooling layer, and consequently the next layer. The memory tracker refers to tile controller for the addresses of the target tile deployed with the weight matrix of the next CNN layer.

4.3 Mode Switching

The context switching between FP and BP follows three rules: 1) change as few tiles as possible; 2) avoid interruptions on FP/BP; 3) the ahead layers have higher priority since they demand more computing resources when propagation arrives.

Fig. 4 demonstrates how the tile controller switches the tile between the FP and BP modes. The state register in the tile controller records the state of tile and then sends the signal to the switch mux. When the state indicates FP, I/O input data enters the FP data buffer (indicated by blue lines). The DAC converts the data to the analog signal and horizontally applied to the ReRAM crossbar. The currents accumulate at the bottom of the crossbar. The result is delivered to the memory according to the address stored in the destination address register. When the state changes to 10 (BP), the data goes through the BP data buffer (indicated by orange lines). The signal enters the crossbar at the top interface and leaves at the right side. We ignore the ADC/DAC components for clarity. Afterward, the result of FP and BP is processed in different components. It is clear that only the FP and BP operated in the same layer shares the same ReRAM crossbar.

4.4 Resource Deployment

Based on the CNN topology and computing resource, this section describes how to allocate the resource for each CNN layer and generates the corresponding information.

In previous works, the resources are simply allocated equally for each layer of CNN which idles a large fraction of resources in the tail layers. The problem of resource allocation is formulated as below: Given a N convolution layer (L_1 to L_N) CNN, the pooling size between layer i and layer $i+1$ is P_i and the resource allocated to layer i is S_i. There are S tiles in total. Here S_i,S,N,P are all integers.

Subject to :

$$S = \sum_{i=1}^{N} S_i \quad \begin{cases} S_i \geq 1 \\ \frac{S_i}{S_{i+1}} \leq P_i \end{cases} \tag{1}$$

To minimize :

$$f = \sum_{i=1}^{N} \left(\frac{S_i}{S_{i+1}} - P_i \right)^2 \tag{2}$$

In the subjection (1), we assure each layer has at least one tile to guarantee the normal execution; The ratio of resource amount in adjacent layers cannot exceed the pooling layer size to avoid the potential waste because no bubble exists if $\frac{S_i}{S_{i+1}} = P_i$. The target function (2) aims to keep the pipeline as busy as possible by making the ratio of resources between adjacent layers closer to the pooling layer size. Note that the algorithm does not take the size of a layer into consideration. A direct and explicit explanation is shown using S_i as example. The subjection changes when it comes to *SDMP*. In *SDMP*, dedicated resources are assigned to FP and BP, separately; therefore at least two or multiples of two are required for each layer:

$$S = \sum_{i=1}^{N} S_i \quad \begin{cases} \frac{S_i}{S_{i+1}} \leq P_i \\ S_i \geq 2 \\ S_i mod 2 = 0 \end{cases} \tag{3}$$

For small-scale CNN layer, we replicate the computing resource allocated to it to compose a whole tile. For large-scale layer, we divide the resource requirement into several tiles; these tiles can communicate with each other and compute simultaneously. We divide the resource requirement based on the input channel so that a single tile only stores partial channels of feature maps. Consequently, some of the tiles have to act as accumulators, i.e., they accumulate the output results of different tiles and then apply them to the activation function. In the evaluation section, we have implemented several large-scale CNNs to evaluate our design.

4.5 Dataflow

We display the data flow of the proposed HUBPAD architecture, using the same example CNN in Fig. 2 for the fair comparison. The size of the weight matrix in layers is ignored for simplicity here as it can be simply scaled up during the implementation. In

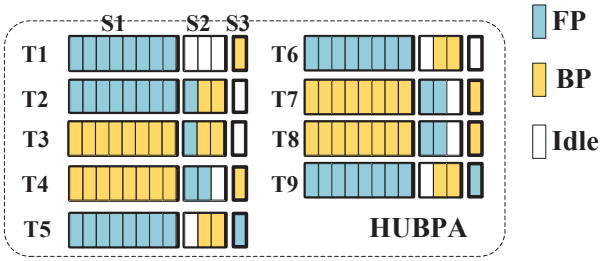

Figure 5: Utilization of limited computing resource utilization in HUBPA

this example, there are 12 tiles in total. In our resource allocation algorithm, the numbers of tiles allocated for each layer are 8, 3, 1 in HUBPA(and TDMP), and 8, 2, 2 in SDMP. This allocation can yield the best throughput given such limited resource according to section 4.4. As shown in Fig. 5 and Fig. 2, HUBPA finishes one FP and one BP in 5 cycles and two FP and two BP in 9 cycles. SDMP finishes one FP and one BP in 6 cycles. TDMP finishes one FP and one BP in 8 cycles. Furthermore, to finish $N(N > 1)$ FP and BP under this circumstance, HUBPA consumes $5 + 4 \times (N - 1)$ cycles while SDMP consumes $6 \times N$ and TDMP $8 \times N$. The larger N is, the more significant the improvement is. Section 5 shows Further experiments results.

5 EVALUATION

Parameter			Value
Chip	Tile		168
	Tile	PA	12
		eDRAM	64KB
		OR	2KB
		Pool/Sigmoid	1/2
		PA: ADC	4
		PA: DAC	4
		PA: Array	128x128
		PA: Array Num	4
		PA: Bit/cell	2
		PA: OR/IR	2/2
		PA: S+H	4
		Router: flit	32
		Router: port	8
	Hyper	Links/Fre	4/1.6GHz
		link bw	6.4GB/s

Figure 6: The parameters of proposed architecture.

5.1 Experiment Setup

Fig. 6 represents the parameters of ReRAM based chips. In order to evaluate the proposed architecture, we evaluate eight start-of-art deep convolutional neural networks. Five of the benchmarks are VGG [16]-based, and three of them are MSRA [8]-based. All networks are evaluated on the widely used ImageNet [14] data set. TDMP and SDMP[17] with the same amount of computing resource are compared with our HUBPA in the evaluation.

Performance Evaluation We build a detailed and cycle-level simulator based on the NVsim [3], implemented in C++. The simulator can record all events including the dot-product computations, data transfer and on/off-chip memory accesses. The write/read latency model of memristors refers to [19]. We integrate the DRAMsim2 [13] into the simulator to model the delay and energy of off-chip DRAM memory accesses.

Energy Evaluation The energy and layout of memristor cells are also derived from models in [19]. Besides, the energy for the shift-and-add circuits, the max-pooling circuit, and the sigmoid operation inherited the analysis in ISAAC [15]. We use the CACTI 6.5 to model the eDRAM buffer, and the interconnect in the architecture. DRAMsim2 [13] is used to model the energy of off-chip memory access. A compact 8-bit ADC is used in our experiments. We deploy a 1-bit DAC for each row in ReRAM crossbar. We can explore multi-bit DACs by using the power model in [9].

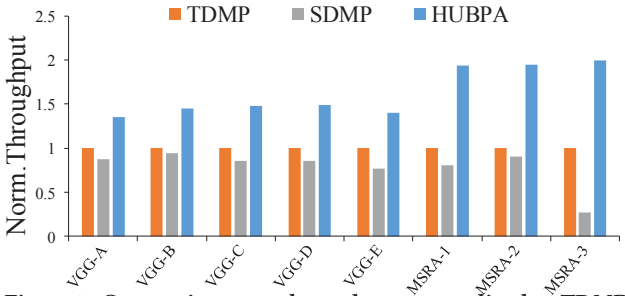

Figure 7: Comparisons on throughput normalized to TDMP across various CNNs

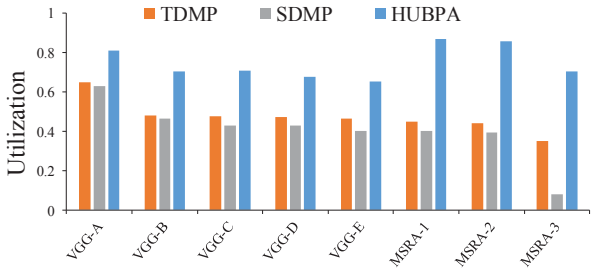

Figure 8: Comparisons on the utilization of computing resources across various CNNs

5.2 Results Analysis

Performance. Fig. 7 represents the **throughput** for a variety of pipeline designs and neural networks. The results are normalized to the throughput of TDMP. The throughput of SDMP is lowest, about 20% lower than TDMP. HUBPA achieves the highest average throughput, 1.6× higher than TDMP and 2.2× than SDMP. In the deepest network, MSRA-3, HUBPA achieves 2.1× throughput over TDMP. The more layers a neural network, the higher performance is achieved by HUBPA. Because a deeper neural network often has more down-sampling stages that lead to more bubbles.

Fig. 8 shows the **computing resource utilization** in TDMP, SDMP and HUBPA. The utilization degrades as the neural network becomes deeper (VGG-A to VGG-E, MSRA-1 to MSRA-3). In MSRA-1, TDMP achieves 47% utilization, and in MSRA-3, the utilization decreases to 35%. SDMP achieves 43% utilization in MSRA-1 and MSRA-2 on average, and it degrades to only 8% in MSRA-3. The

utilization of HUBPA is more than 70% in VGG-A,VGG-B, and VGG-C. In MSRA-1 and MSRA-2, the utilization can achieve above 85%. Even in the MSRA-3, HUBPA still maintains 70%. The utilization of HUBPA increases notably.

Fig. 9 represents the **computational efficiency (CE)** of different pipeline designs. What CE denotes is the number of 32-bit instructions performed per second per mm^2 on a chip. SDMP achieves about 180 $GOPS/mm^2$ on average. In MSRA-3, CE shrinks downward to only 32 $GOPS/mm^2$. The CE of TDMP is higher than SDMP. Compared to both TDMP and SDMP, HUBPA achieves an improvement of 400 $GOPS/mm^2$ CE. When the neural network becomes larger and deeper, the CE degrades in all these three architecture. However, HUBPA can obtain a high CE, and the improvement of HUBPA even increases when the network becomes deeper.

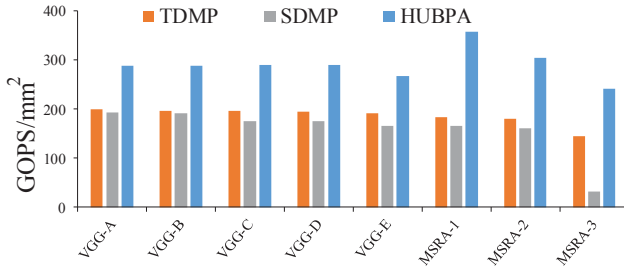

Figure 9: Comparisons on computational cost efficiency. The left axis corresponds $GOPS/mm^2$

Energy Efficiency. Fig. 10 represents the energy efficiency of different architectural designs. The energy efficiency is the number of 32-bit operations performed per second per waltz (W). From Fig. 10, the power efficiency results of TDMP and SDMP are similar, but the HUBPA chip is improved about 1.5×.

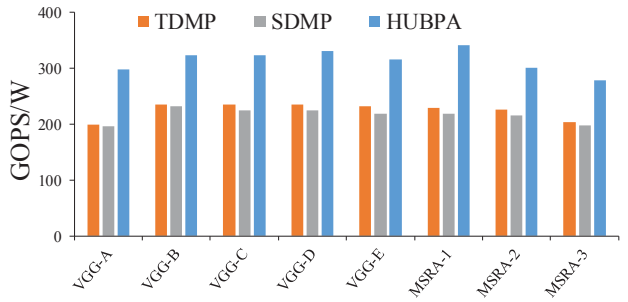

Figure 10: Comparisons on Energy Efficiency across various CNNs

6 CONCLUSION

ReRAM-based pipeline architecture with high energy and cost efficiency is promising to meet the demand for the deployments of CNNs in both edge and cloud computing. Our proposed bidirectional pipeline architecture, HUBPA, substantially improves the performance and energy efficiency for CNN training. We propose a novel dynamic context switching mechanism that the forward and backward propagation can share the computing resource in the time dimension and thus enhance the utilization of the computing resources. CNN deployment algorithm, cooperative switching scheme, and hardware support are described. The experimental results show our proposed architecture achieves 1.7× computing resource utilization and 1.5× energy efficiency in average compared with the state-of-the-art ReRAM-based pipeline architecture. This work also provides the efficient computing power for new learning paradigms, such as "semi-supervised learning" and "reinforcement learning" on edge.

ACKNOWLEDGEMENTS

This research was partially supported by National Natural Science Foundation of China (Grant No. 61834006, 61602300,61772331), Shanghai Science and Technology Committee(No.18ZR1421400), Shanghai Jiao Tong University Biomedical Engineering Research Foundation(No.YG2015MS17), Shanghai clinical ability construction of The three grade hospital(No.SHDC12015904).

REFERENCES

[1] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.
[2] Jeffrey Dean et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[3] Xiangyu Dong et al. 2014. NVSim: A circuit-level performance, energy, and area model for emerging non-volatile memory. In *Emerging Memory Technologies*. Springer, 15–50.
[4] Diederik P. Kingma et al. 2014. Semi-Supervised Learning with Deep Generative Models. In *NIPS*.
[5] Krizhevsky et al. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*. 1097–1105.
[6] Kaiming He et al. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.
[7] Shankar et al. 2017. Reinforcement Learning via Recurrent Convolutional Neural Networks. In *International Conference on Pattern Recognition*. 2592–2597.
[8] He Kaiming et al. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.
[9] Boris Murmann. 2017. ADC performance survey 1997-2017. *https://web.stanford.edu/ murmann/adcsurvey.html* (2017).
[10] Augustus Odena, Christopher Olah, and Jonathon Shlens. 2017. Conditional Image Synthesis With Auxiliary Classifier GANs. In *ICML*.
[11] Brandon Reagen et al. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 267–278.
[12] Minsoo Rhu et al. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016*. IEEE, 1–13.
[13] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10 (2011), 16–19.
[14] Olga Russakovsky et al. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115 (2015), 211–252.
[15] Ali Shafiee et al. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 14–26.
[16] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[17] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 541–552.
[18] C. Xu et al. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*.
[19] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie. 2011. Design implications of memristor-based RRAM cross-point structures. In *DATE*. 1–6. https://doi.org/10.1109/DATE.2011.5763125