DWMAcc: Accelerating Shift-based CNNs with Domain Wall Memories

ZHENGGUO CHEN and QUAN DENG, National University of Defense Technology
NONG XIAO, Sun Yat-sen University
KIRK PRUHS and YOUTAO ZHANG, University of Pittsburgh

PIM (processing-in-memory) based hardware accelerators have shown great potentials in addressing the computation and memory access intensity of modern CNNs (convolutional neural networks). While adopting NVM (non-volatile memory) helps to further mitigate the storage and energy consumption overhead, adopting quantization, e.g., shift-based quantization, helps to tradeoff the computation overhead and the accuracy loss, integrating both NVM and quantization in hardware accelerators leads to sub-optimal acceleration.

In this paper, we exploit the natural shift property of DWM (domain wall memory) to devise DWMAcc, a DWM-based accelerator with asymmetrical storage of weight and input data, to speed up the inference phase of shift-based CNNs. DWMAcc supports flexible shift operations to enable fast processing with low performance and area overhead. We then optimize it with *zero-sharing*, *input-reuse*, and *weight-share* schemes. Our experimental results show that, on average, DWMAcc achieves 16.6× performance improvement and 85.6× energy consumption reduction over a state-of-the-art SRAM based design.

CCS Concepts: • **Computer systems organization** → **Embedded hardware**; • **Hardware** → *Hardware accelerators*; Non-volatile memory;

Additional Key Words and Phrases: CNN accelerators, domain wall memory

ACM Reference format:
Zhengguo Chen, Quan Deng, Nong Xiao, Kirk Pruhs, and Youtao Zhang. 2019. DWMAcc: Accelerating Shift-based CNNs with Domain Wall Memories. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 69 (October 2019), 19 pages.
https://doi.org/10.1145/3358199

1 INTRODUCTION

The convolutional neural networks (CNNs) have recently achieved great successes in a wide range of application domains, e.g. image classification [18] and speech recognition [10]. To achieve better inference accuracy, modern CNNs are becoming increasingly complicated, making them both

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2019.
Authors' addresses: Z. Chen, College of Computer, National University of Defense Technology, Changsha, Hunan, China, 410000; email: zchen@pitt.edu; Q. Deng, College of Computer, National University of Defense Technology, China; email: dengquan12@nudt.edu.cn; N. Xiao (Corresponding author), School of Data and Computer Science, Sun Yat-sen University, China; email: xiaon6@mail.sysu.edu.cn; K. Pruhs and Y. Zhang (Corresponding author), Department of Computer Science, University of Pittsburgh, USA; emails: {kirk, zhangyt}@cs.pitt.edu.

1539-9087/2019/10-ART69 $15.00
https://doi.org/10.1145/3358199

memory and computation intensive, e.g., the VGG-16 neural network is a 16-layer image classification network that contains 138M weight parameters. Given a convolution layer, the core building block of CNNs, each filter is convolved across the input, which involves a large number of multiplication, add, and other arithmetic and logic operations.

Recent studies have proposed hardware accelerators to address the memory and computation intensity of modern CNNs. DaDianNao is a PNM (processing-near-memory) design that integrates eDRAM within NFUs (neural functional units), i.e., their specially designed units, for accelerating CNNs [3]. Prime [6] and ISAAC [19] exploit ReRAM analog characteristics to speedup dot product and thus achieve PIM (processing-in-memory) based CNN acceleration. Neural Cache [9] is an SRAM-based reconfigurable CNN accelerator. DRISA [15] is a DRAM based CNN accelerator that leverages charge sharing across multiple DRAM rows.

The intensity reduction has also been achieved through algorithm renovations. Holt et al. [11] revealed that CNNs exhibit significant error tolerance such that adopting narrow fixed point number representation reduces storage demands with tolerable accuracy losses. Later studies [7, 14, 17] further compacted the weight parameters to two values (i.e. +1 or −1) or three values (i.e. +1, 0, and −1) with modest inference accuracy degradation. A generalized approach, referred to as *shift-based neural networks* (ShiftNNs) [27], quantizes weights as a small set of power-of-2 values to achieve a flexible tradeoff among computation overhead, weight storage and inference accuracy. Ding et al. [8] show that ShiftNNs realize higher accuracy among quantized neural networks.

In this paper, we adopt ShiftNNs to devise an accelerator with high accuracy for mobile devices. Intuitively, integrating quantization algorithms in NVM (non-volatile memory)-based accelerators may combine the storage reduction and performance benefits from these designs. However, most existing accelerators are designed for speeding up multiplication and addition operations rather than shift operations, leading to sub-optimal performance improvement on ShiftNN-based ones.

In this paper, we address the issue with DWMAcc, a DWM-based CNN accelerator that utilizes the natural shift property of DWM. We summarize our contributions as follows.

- We elaborate the limitations of existing accelerator designs when being adapted on ShiftNNs. Our study shows that laying out weights and inputs in memory exhibits large overhead, which degrades system performance significantly.
- We propose DWMAcc, a DWM-based ShiftNNs accelerator with asymmetrical storage of weight and input data at the cache level, to address the performance issue of ShiftNNs. DWMAcc exploits the natural shift property of DWM with three designs to improve performance and energy efficiency. To our best knowledge, this is the first PIM-based DWM accelerator.
- We evaluate DWMAcc and compare it to the state-of-the-art SRAM-based PIM-accelerators. Our experimental results show that DWMAcc achieves on average 16.6× performance improvement and 85.6× energy consumption reduction over the state-of-the-art.

2 BACKGROUND

2.1 Basics of Domain Wall Memory (DWM)

The traditional SRAM and DRAM technologies face severe scalability and energy consumption in the deep submicron regime. Among many emerging non-volatile memory technologies, e.g., PCM [2], ReRAM [6], STT-MRAM [5], and Domain wall memory (DWM) [16, 26], DWM is one of the most promising ones for on-chip memory construction.

DWM access. A DWM cell is one ferromagnetic nanowire (also referred to as racetrack, or RT), shown in Figure 1(a). One RT consists of multiple domains separated by domain walls, a couple of shifters at the two ends, and one or multiple access ports. By storing one logic bit (i.e., 0 or 1) in

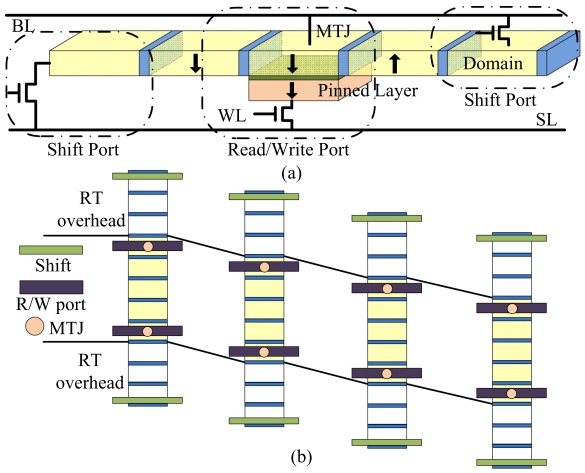

Fig. 1. The structure of DWM cell.

one domain and multiple bits in one racetrack, DWM achieves ultra-high density, i.e., more than 10 bits/F^2. As a comparison, ReRAM crossbar has $4F^2$ cell size while modern DRAM has about $8F^2$ cell size. In addition, DWM has fast read and write speeds and low leakage power [21, 23]. DWM is compatible with CMOS technology and has outstanding scalability.

DWM operations. DWM supports three basic operations — shift, read, and write. A DWM RT has two shifters placed at the two ends. By injecting spin-polarized currents, the two shifters can drive all the domains and the domain walls in the RT in two directions, respectively.

A DWM cell may integrate read ports and write ports for read and write operations, respectively, or read/write ports that can perform both read and write operations. The DWM cell in Figure 1(a) adopts one read/write port that consists of a transistor and a magnetic tunneling junction (MTJ). The transistor is controlled by the wordline while the MTJ is a sandwich structure including a domain at top, an MgO barrier in the middle, and a pinned layer with preset magnetic direction in the bottom. The MTJ is connected to the wordline, sourceline, and bitline of the cell. The transistor enables read and write operations by controlling the current density as well as the access latency.

The domain above the read/write head can be read or written directly. Other domains need to be shifted above the read/write head before read or write. Thus, the middle domain in Figure 1(a) can be accessed directly while it needs shift operations to access other domains. The stored logic bit in each domain is represented by its magnetic direction — it is '0' if its direction is in parallel with that of pinned layer; or '1' if otherwise. If we read the three domains in continuous left-shift operations, we get "001". Otherwise, we get "100". The two domains at the far ends of the RT are buffers to avoid data loss when shifting data in both directions.

DWM cell arrays. DWM-based cell arrays adopts the traditional 2D structure. Since a transistor in shift or read/write port is much wider than the RT, simply placing multiple RTs horizontally shall lead to wasted die area between two RTs. To improve area efficiency, recent designs group four DWM RTs as a macro unit (MU) and place them next to each other with a placement offset between two adjacent RTs [23], as shown in Figure 1(b). For clarity, the figure does not show the exact width of the transistor. With the width of a transistor being 4× that of RT, the transistors overlap with RTs due to placement offset. Dummy domains are deployed at both ends to assist shift operations. They are referred to as RT overhead [21] in Figure 1. The number of dummy domains at one end equals to the largest shift distance of the RT. Integrating more read/write ports can reduce shift distance, which improves access performance and reduces RT overhead. In the figure, we deploy

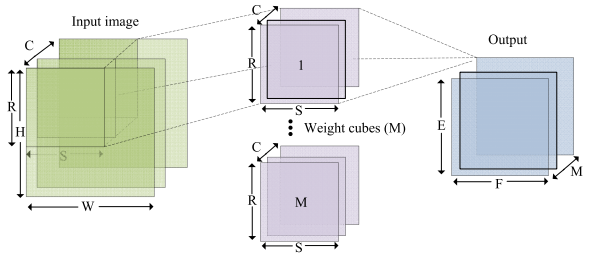

Fig. 2. The computation for a CONV layer.

two read/write ports for each RT, which reduces the worst case shift distance from N-1 to N/2-1 (where N is the number of valid domains in one RT).

The macro unit, being the basic building block of DWM arrays, depends on three parameters, i.e., the number of domains in one RT, the number of read/write ports, and the number of RTs in one MU. Different configurations exhibit different latency, power, and area efficiency tradeoffs [25]. Without losing generality, in this paper, one MU has four RTs while each RT has 64 domains (64-domain RTs are widely adopted in recent DWM studies [12, 21]).

In summary, by storing one logic bit in one domain and multiple bits in one RT, DWM achieves ultra-high density. DWM also achieves fast read and write speeds and low leakage power [21, 23]. DWM exhibits outstanding scalability and similar access latency as SRAM [24], making it one of the most promising technologies for constructing future caches.

2.2 Hardware CNN Accelerators

A typical CNN (Convolutional neural networks) consists of three types of layers: convolutional (CONV), pooling, and fully connected (FC) layers. Figure 2 shows a CONV layer of which an input image has three dimensions, i.e., width (W), height (H) and channel (C) while a weight cube has four dimensions, i.e., width (S), height (R), channels (C) and batch (M). A weight cube that contains many pieces of weight matrix is overlaid in the input image and each pixel in the weight cube is multiplied by corresponding pixel of input image. A convolutional result is the dot product of input and weight values across R, S, and C three dimensions. After sliding the window in the input image, it produces one element for output (E×F). Each weight cube has C channels while there are M cubes, which produces an output of E×F×M. For a CNN, a pooling layer samples feature maps using a pooling algorithm, e.g. max-pooling while a fully connected layer computes the dot product between a feature map and a weight cube.

Hardware CNN Accelerators. The most computation-intensive operation in CNNs is the dot product operation that appears frequently in CONV and FC layers. In this paper, we focus on speeding up the dot product operation while the accelerator serves to execute all layers.

Modern CNNs are computation and memory intensive. When using the CPU to compute the dot product, we not only have to load weights, input feature maps, and output feature maps, but also need to conduct expensive multiplications, which results in low performance and high energy consumption. To address these challenges, recent hardware accelerators avoid transferring data to and from the CPU by using PIM (processing-in-memory) and PNM (processing-near-memory) approaches.

As shown in Figure 3(a), PNM designs place computation units next to the memory modules for exploiting the high bandwidth of internal buses. DaDianNao is such a design that integrates multiple eDRAM modules within NFUs (neural functional units) to parallelize memory and computation operations [3]. PNM designs are often constrained by the number of computation

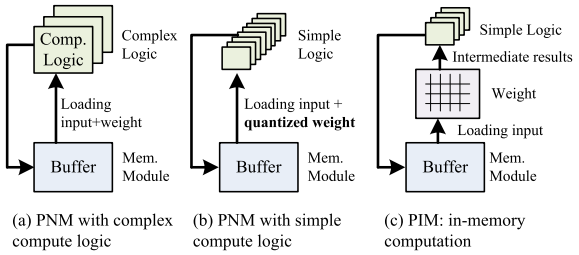

Fig. 3. Comparing PNM and PIM based accelerators.

units — the more complex the computation unit is, the fewer units the accelerator can integrate, the lower throughput the accelerator can achieve. To achieve higher throughput, [7] eliminates multiplication by compacting all weights to either +1 or −1, which converts the dot product to a sequence of add operations. To minimize the potential inference loss, a generalized approach is to quantize weights as a small set of power-of-2 values [27] (discussed in next chapter), as shown in Figure 3(b). Depending on the number of power-of-2 values in the set, this approach can achieve a flexible tradeoff between weight storage and inference accuracy.

The PIM approach is to exploit the intrinsic characteristics of different memory technologies to speed up the computation. Prime [6] and ISAAC [19] exploit ReRAM analog characteristics to speedup dot product computation through current accumulation. As shown in Figure 3(c), Prime [6] first loads weights in the memory-based compute module, and then loads the input feature maps. After computing the dot products, Prime produces the output using simple logic such as adders, etc.

3 MOTIVATION

3.1 ShiftNNs Achieves A Better Tradeoff

From preceding discussion, adopting quantized weights reduces storage and computation overhead. We focus on Shift-based Neural Networks (ShiftNNs) in this paper as they provide better tradeoff between weight storage reduction and inference accuracy.

ShiftNNs convert weights to power-of-2 values, which reduces computation overhead by converting multiplication operations in dot product to shift operations. Assume we are to compute the dot product of two 3-element vectors A and B, and the elements of B are represented using power-of-2 values. Without losing generality, we assume $B = (2^{-1}, 2^0, 2^1)$ and thus

$$A \otimes B = \sum_{i=0}^{2} (A_i \times B_i) = (A_0 >> 1) + A_1 + (A_2 << 1) \tag{1}$$

In this equation, the dot product is the sum of shifted A's elements. It is clearly more efficient to execute shift operations than to do full-precision multiplication. ShiftNNs not only improves performance but also reduces storage overhead and energy consumption over full-precision neural networks. Regarding the inference accuracy, a 5-bit-quantized-weight ShiftNN achieves the similar accuracy as that of a full-precision CNN [27]. This accuracy is better than those of BNNs and TNNs, e.g., a 2-bit-quantized-weight ShiftNN achieves approximately 5% top-1 accuracy improvement over TNNs [27].

Given ShiftNNs achieves higher accuracy than full-precision neural networks as well as BNNs and TNNs, they are suitable to be deployed for future high-end mobile devices that have limited computation and storage resources but demand high inference accuracy, e.g., automated driving systems.

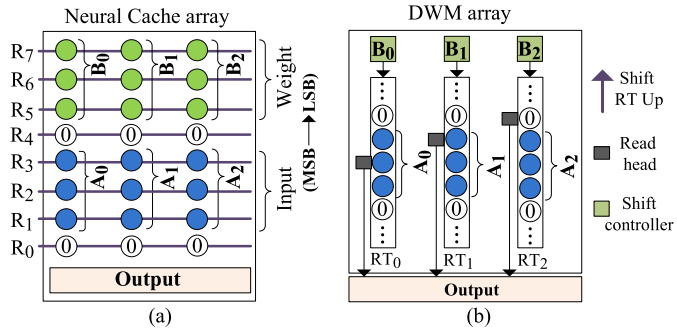

Fig. 4. Comparing shift implementation in Neural Cache and DWMAcc.

3.2 Natural Support of Shift Operations in DWM

While PIM-based CNN accelerators exhibit great performance and energy efficiency advantages, a simple integration of ShiftNN with PIM designs results in sub-optimal results. In particular, existing PIM designs are to speed up multiplication and addition rather than shift operations. ISAAC [19] and Prime [6] contain significant power and area overhead of analog-to-digital converter and digital-to-analog converter. While DRISA [15] may support shift operations, it integrates shifters into each subarray, which introduces large hardware overhead.

The work that is most similar to our design is Neural Cache [9], an SRAM-based accelerator for speeding up CNN inference. Neural Cache maps the bits of one value vertically in a column rather than horizontally in a row. In Figure 4(a), assume we need to compute the dot product of two 3-bit vectors, i.e., the input A and the weight B. Each vector has three elements. To compute $A_i \times B_i$ (i = 0,1,2), Neural Cache starts from the wordline R_7, and check whether $B_i(0)$ (i = 0,1,2) is 1 or not, where $B_i(j)$ represents the j-th bit of B_i. If $B_i(0) = 1$, e.g., $B_1(0) = 1$, it adds A_0 to output bit-by-bit in O(n) steps. Otherwise, it has no operations. Similarly, it activates R_6 and R_5 to check $B_i(1)$ and $B_i(2)$, respectively. Neural Cache computes $A_i \times B_i$ (i = 0,1,2) after 9 addition operations. To summarize, Neural Cache computes the n-bit multiplication in $O(n^2)$ steps with the multiplication along different bitlines being performed in parallel.

Given a value is vertically laid out along the bitline, reading the value with a different starting wordline naturally implements the shift operation. For example, reading wordline R2, R1 and R0 for A0 actually computes A0»1, i.e., a shift-operation can be implemented in O(n) steps. However, when implementing ShiftNN algorithms, even though their weights are power-of-2 values, not all the weights are the same, e.g., we may need to compute A0»1, A1»0, and A2«1 simultaneously. Neural Cache cannot effectively implement shift operation at the array level. Instead, it recognizes these weights as arbitrary binary numbers and implements the multiplication through repetitive additions. That is, the shift-based multiplication operations of ShiftNN are still implemented in $O(n^2)$ steps in Neural Cache.

For illustration purpose, we next show a potential DWM based accelerator design. In Figure 4(b), a DWM subarray consists of multiple RTs and their shift controllers. Assuming we represent B = $(2^{-1}, 2^0, 2^1)$ with three bits (two integer bits and one fraction bit), we have $B = (001_b, 010_b, 100_b)$. We first exploit the value $B = (001_b, 010_b, 100_b)$ to move the second bit of A_0, the first bit of A_1, and the padded 0 bit of A_2 under the read heads of the three RTs, respectively. We then read these bits, move the read heads to the next position synchronously, and repeat the steps three times to get the three shift-based multiplication results, respectively. In this way, the computation of A_0»1, A_1»0, and A_2«1 are done simultaneously, that is, a DWM-based accelerator

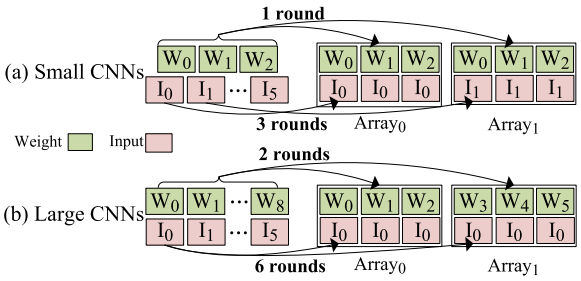

Fig. 5. Different CNNs have different input and weight sizes.

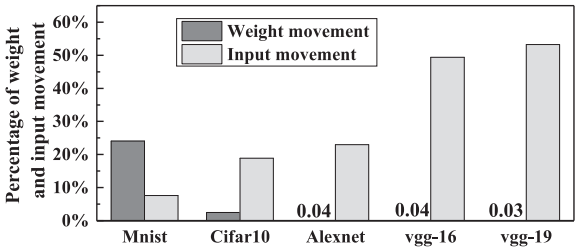

Fig. 6. Comparing the weight and input data movement overhead.

can finish all shift-based multiplications in O(n) steps. We next elaborate the design details and the optimizations for performance improvement and power consumption reduction.

3.3 Non-negligible Data Loading Overhead

We next elaborate a major constraint in PIM-based accelerator designs, i.e., it takes long latency to load both weight and input data into memory before computation. In Figure 5, we assume a CNN convolution layer consists of multiple weight cubes and input cubes while each Neural Cache array can store three cubes of each type to enable in-memory computation. For a small CNN that contains three weight cubes (Figure 5(a)), we may allocate two memory arrays so that we can process two input cubes simultaneously. For a large CNN that contains nine weight cubes (Figure 5(b)), even we may distribute weight cubes to two memory arrays, we need two rounds to finish nine cubes. In addition, each input cube needs to be loaded to both arrays to finish fully convolution. Different design strategies may be designed to prioritize either weight loading or input loading for maximized convolution efficiency.

Figure 6 compares the weight and input movement overhead for different CNNs. We report the percentage of the overall latency on weight and input movement, respectively. The experiment settings are detailed in Section 6. From the figure, the weight movement overhead is significant only for small networks, e.g., about 25% for `Mnist`. For large CNNs, the input data movement becomes the bottleneck, e.g., for `VGG19`, it takes more than 50% of the inference latency to move input data in and out of memory arrays. Therefore, we need to devise appropriate memory layout to enable efficient PIM-based acceleration.

4 THE DWMACC DESIGN

In this section, we elaborate the DWMAcc design for speeding up *lightNN-1* [8], a popular ShiftNN implementation that represents each weight using, e.g., 8 bits. A weight in *lightNN-1* contains at most one bit 1 for the 8 bits. Note that DWMAcc is independent of the choice of ShiftNN algorithms

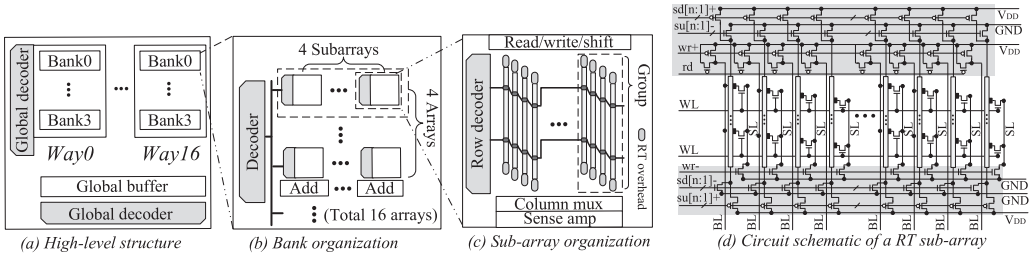

Fig. 7. The architectural overview of DWMAcc.

such that other ShiftNN implementations, e.g., *INQ [27]* and *lightNN-2* [8], can be smoothly adapted with more shift operations.

We represent the weight data in quantized format and the input data in fixed point binary format, respectively. Both types of data are represented in 8 bits[1]. In particular, each weight value is of the form of $\pm 2^{-m}$ (m = 0, 1, . . . , 7), indicating that it can right shift an input value by up to 7 bits.

4.1 An Overview of DWMAcc

Figure 7 presents an architectural overview of DWMAcc. As shown in the figure, DWMAcc shares the similarity of a 17-way set associative cache with each way containing four banks. One bank consists of sixteen arrays, and one array consists of four subarrays. The global decoders and the global buffer are laid out to enable not only traditional cache read and write accesses but also DWM shift operations. Every four arrays within one bank share the sixteen adder units to support accumulation and activation. Only one array occupies them at a time. One subarray consists of 64 RTs, their read and write ports, and control logic (including the shift control). With one RT saving 64 bits, the capacity of one subarray is 64 × 64 bits. Figure 7(d) presents the circuit schematic of one RT subarray.

To speed up the *lightNN-1* algorithm, DWMAcc uses the first 16 ways for convolution (convolution way) and the 17th way for saving the output (output way). DWMAcc loads the input data in the domains of the RTs, exploits the weight data to shift the RTs accordingly, and then accumulates the intermediate results to compute the final dot product results.

We next elaborate the design details of DWMAcc.

4.2 Consecutive Bit Access in DWMAcc

Given one 64-bit RT, we layout four 8-bit input data values so that two values are interleaved by eight 0s, as shown in Figure 8(c). In our baseline design, each RT consists of four access heads. While each of the four access heads can read and write the RT, only one head may be active at any time. By default, the MSBs (most significant bits) of four input data values are under the heads.

A major difference between DWMAcc and traditional DWM-based cache designs is that, **one DWMAcc access reads (or writes) eight consecutive bits from the same RT** while, for most DWM-based cache designs, one access reads/writes one bit from one RT [21, 23]. By accessing consecutive bits from the same track, DWMAcc implements a shift operation by setting the initial head positions. For example, given $A = a_7a_6 \ldots a_0$, if DWMAcc moves the domains along the downstream by 7 bits from the default positions and then read 8 bits, it reads A, i.e, the original value. If it moves the domains along the downstream direction by 6 bits and then read the 8 bits, we get 'a_1,a_2, . . . , a_7,0', which represents '0$a_7a_6 \ldots a_1$', or A»1 (right shifting A by one bit).

[1]Existing studies [9, 28] have shown that 8-bit precision achieves sufficient accuracy for DNNs.

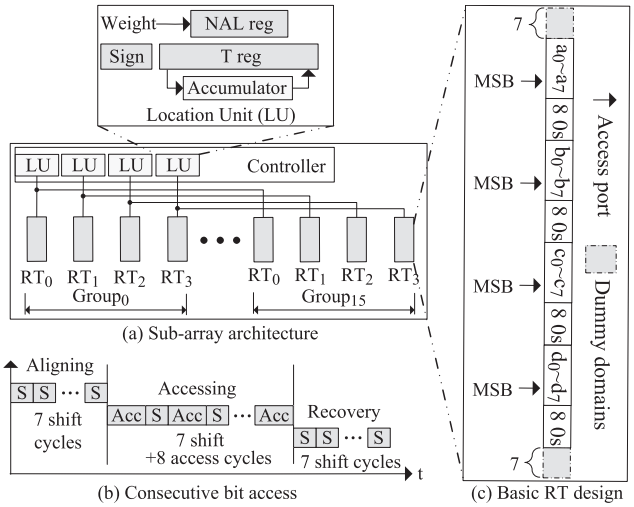

Fig. 8. Implementing shift operation through head initialization and consecutive bit access.

The 0s placed between two input values allow us computing the results with different shift amounts. Given an input value, e.g., *A*, has eight bits, we may compute *A*»7, i.e, right shifting seven bits. Note, right shifting eight bits gives all zeros; and the *lightNN-1* algorithm only needs right shifts.

Figure 8 illustrates the hardware design to enable the consecutive bit access. Each track is associated with a `T-reg` register to track the head position. To resetting the head to a specific position, the target position is compared with `T-reg` so that DWMAcc can determine the shift amount and shift direction, control the 'su[n:1]+' and 'su[n:1]-' (or 'sd[n:1]+' and 'sd[n:1]-') signals to control the RT to shift up (or down). Here, we choose $n = 4$, meaning we split the 64 RTs in one sub-array into 16 groups with one group containing consecutive four RTs. While we use the four pairs of signal wires to control different RTs in one group, respectively, we use the same pair to control the aligned RTs in all groups. That is, as shown in Figure 8(a), the first `T-reg` ensures that the first RTs from all 16 groups shift the same direction and the same amount.

To enable consecutive bit accesses, the `NAL` register tracks the starting position of the next access, which is initiated to a weight value as we shall describe in the next subsection. With the `T-reg` tracks the current head position, DWMAcc generates the 'su[n:1]+'/'su[n:1]-' and 'sd[n:1]+'/'sd[n:1]-' signals to set the initial head positions. Given different RTs may need different amount of time to initialize, DWMAcc assumes the worst case and waits for seven cycles (alignment stage). DWMAcc then accesses one bit from each RT, shifts all RTs by one domain along the downstream direction, and continues to access the next bit (accessing stage). After finishing accessing eight bits, DWMAcc resets the heads to the default positions, which may take another seven cycles in the worst case (recovery stage).

4.3 Data Mapping and Multiplication

We next describe the data placement as well as the convolution details. For discussion simplicity, we assume R = S = 4, C = 16, and the convolution step is 2. Figure 9 illustrates the mapping of input and weight data for one channel (e.g., k = 0) while Figure 10 illustrates the mapping for multiple channels.

For one channel (Figure 9), we distribute the four rows of the weight matrix to four subarrays, e.g., $w_{000}/w_{010}/w_{020}/w_{030}$ are sent to the first subarray only. Within each subarray, the input

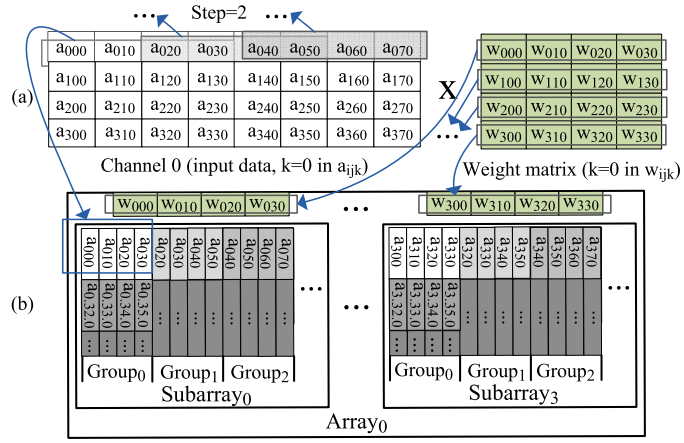

Fig. 9. Mapping scheme of data from one channel.

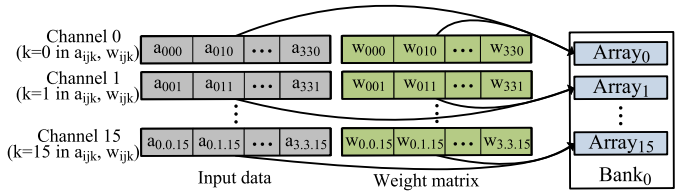

Fig. 10. Mapping scheme of data from 16 channels.

values for computing one dot product are placed at aligned positions on different RTs, e.g., $a_{000}/a_{010}/a_{020}/a_{030}$ are placed as the first values of the four RTs in $Group_0$ of $Subarray_0$. Given the convolution step is 2, $a_{020}/a_{030}/a_{040}/a_{050}$ are placed as the first values of the four RTs in $Group_1$. Note that a_{020} are placed twice to enable smooth convolution. For one RT, e.g., the first RT of $Group_0$ of $Subarray_0$, it holds the input values a_{0j0} with j = 0, 32, 64, ...

With each group in one subarray computing four intermediate values, the aligned groups, $Group_0$, from four subarrays compute 16 values to be summed up to one dot product. Since all RTs in each subarray are activated in computation, the four subarrays can compute 16 dot products simultaneously.

Figure 10 illustrates the data mapping for 16 channels, i.e., $0 \le k \le 15$. The weights with the same k index are serving one array. With 16 arrays in one bank, we support up to 16 channels in one bank. With each array produces one intermediate result, the final dot product is generated from summing up all intermediate results.

4.4 Accumulation and Activation

For the preceding example, after computing $a_{ijk} \times w_{ijk}$ ($0 \le i \le 3$, $0 \le j \le 3$, $0 \le k \le 15$), i.e., all 256 shift-based multiplication results, we need to accumulate them to get one dot product result. Each subarray computes four out of these 256 partial results, i.e., $a_{IjK} \times w_{IjK}$ ($0 \le I \le 3$) for a fixed I and K value pair. Each subbank computes 16 out of these 256 partial results, i.e., $a_{ijK} \times w_{ijK}$ ($0 \le$ i,j ≤ 3) for a fixed K value. In other words, the 256 shift-based multiplication results computed in one subbank belong to 16 different dot products.

DWMAcc first accumulates the 16 multiplication results belonging to the same dot product to a MAC (following the term used in Neural Cache [9]) and then accumulates the 16 MACs together

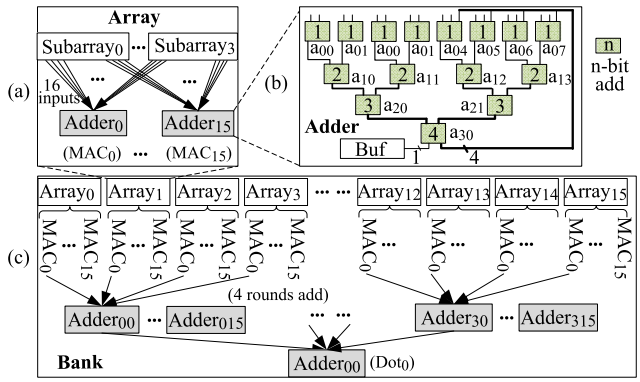

Fig. 11. (a) Addition of MAC; (b) Adder unit design; and (c) Reduction procedure.

to one final dot product. As shown in Figure 11(a), the first step is done using the adders within each bank and each array produces 16 MACs. DWMAcc integrates 16 buffers to assist the computation of MACs for 16 dot products. Figure 11(b) shows the 4-stage adder design. The MAC computation and shift-based DWM multiplication are done in parallel to maximize the acceleration performance.

The second step is done by transferring the MACs between aligned adders for accumulation (Figure 11(c)). In the example, every four consecutive arrays use the 16 adders in time-sharing mode and accumulate 4 aligned MACs to get an intermediate result within four rounds addition. Then it transfer the four aligned intermediate results for accumulation. The final 16 dot products appears in the first 16 adders within a bank. In Figure 11(c), the Dot_0 is in $Adder_{00}$.

DWMAcc then outputs the dot products to the output way (the 17-th way of the conventional organization that has no adder units). DWMAcc supports simple activation algorithms such as ReLU. That is, DWMAcc sends the dot products computed in the buffers to the output way. If the corresponding adder unit detects the sign bit is 1, DWMAcc outputs eight 0s instead of the buffer content.

DWMAcc achieves the maximized resource utilization if the weight cube shape is $4 \times 4 \times 16$ ($R \times S \times C$). For weight cubes of other sizes, DWMAcc adopts the cube splitting or packing scheme [9] to adjust the shape. In summary, DWMAcc exploits the bitline level parallelism in DWM to reduce the multiplication overhead from O(n^2) to O(n), which exhibits great acceleration potentials.

4.5 The Scalability of DWMAcc

Given the fact that large CNNs, e.g., vgg-16 and vgg-19 [13, 20], have large amounts of weights and input data, a hardware CNN accelerator may not store all the data due to its limited memory capacity. DWMAcc addresses the scalability issue in two folds. Firstly, given DWMAcc is based on DWM, it achieves outstanding scalability through ultra-high memory density [24]. A DWMAcc module tends to hold increasingly larger CNNs with the advances of DWM technology. Secondly, DWMAcc can reuse the limited memory to support larger CNNs. That is, it first loads partial weights to compute a subset of dot products, and then fetches new weights from DRAM to compute more dot products, similar as that in existing hardware accelerators [9, 15].

5 OPTIMIZATION

In this section, we propose three optimization schemes, i.e., *zero-sharing*, *input-reuse*, and *weight-share*, for further performance improvement and energy reduction.

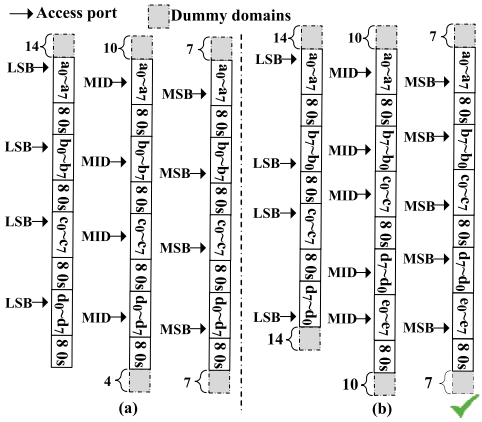

Fig. 12. (a) Simple scheme; (b) Zero-sharing scheme.

5.1 The Zero-sharing Scheme

In the basic DWMAcc design, we pad eight 0s to every input value stored in RT so that shift operations can be implemented by setting initial head positions. Such a design lowers the storage efficiency as one RT can only store four 8b input values. We then study different data layout strategies and propose *zero-sharing* to improve storage efficiency.

There are three choices to set the default access head positions. For example, for input value A = $a_7a_6...a_0$, we may place the head at the LSB (a_0), MID (a_4), or the MSB (a_7) positions, respectively, as shown in Figure 12(a). Since our lightNN-1 scheme may right shift any input by x bits ($0 \leq x \leq 7$), a natural choice is to place the head at the MSB position. For comparison purpose, placing the head at different positions leads to different dummy domains reserved at the two ends of the RT. As an example, if we set the default head position to LSB, we need to place 14 dummy domains at the up end but there is no need to place any dummy domain at the down end. In total, we need 14 dummy domains.

Intuitively, the zero-sharing scheme is to let two input values share the same padded 0s, that is, instead of saving '$a_0a_1...a_7$ 00000000 $b_0b_1...b_7$00000000', we save '$a_0a_1...a_7$ 00000000 $b_7b_6...b_0$' on the RT. A consequence of adopting the new layout is, one DWMAcc access may read eight bits either upstream or downstream along the RT. The eight bits are always read downstream in the basic DWMAcc design. For different head position choices, the maximal shift amount are different. If we allow bidirectional DWMAcc read, we need to allocate the dummy domains for the worst case at both ends. As shown in Figure 12(b), we would have to allocate more dummy domains if we choose the default head position at the LSB or in the middle.

Adopting zero-sharing shall result in uneven head placement along the RT. Figure 12(b) shows that the first and the second heads are 9 domains apart from each other while the second and the third heads are 15 domains apart from each other. This is tolerable as existing RT designs [21] place eight heads along one 64-bit RT, leaving 8 domains apart from each other.

5.2 The Input-reuse Scheme

Given hardware accelerators have limited memory, they may not load all the weight and input data of modern large CNNs. Prioritizing weight reuse keeps weight data in the memory longer such that loading different input data results in computing different output results. Similarly, prioritizing input reuse keeps input data in the memory longer. For example, Figure 5(b) needs 2 rounds of weight loading and 12 rounds of input loading if it chooses weight reuse, and 6 rounds of input

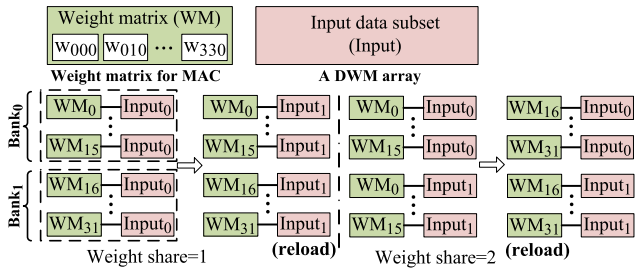

Fig. 13. Weight-share scheme overview.

loading and 12 rounds of weight loading if it chooses input reuse. Eyeriss [4] studied different CNN dataflows and presented an analysis framework to evaluate the performance and energy impacts on a spatial architecture. Due to batching process in CNNs, most hardware accelerators, e.g., Neural Cache, adopt weight reuse such that the same weight may be reused for multiple inference.

However, DWMAcc exploits asymmetrical storage of weight and input, resulting in unbalanced input and weight loading overheads. For the example in Figure 9(b), each weight shifts 16 RTs so that the amount of weight data to be loaded is much smaller that that in input loading. As such, DWMAcc prioritizes input reuse scheme and loads weight on-demand. DWMAcc does need to load weights multiple times to enable batching. However, our study show that input reuse achieves large benefits, as shown in the experiment section.

5.3 The Weight-share Scheme

In this section, we propose the weight-sharing optimization to further reduce the input loading overhead. As we shown in Figure 9, a weight matrix (16 weights) shifts all RTs in one array. And a weight cube consisting of 16 weight matrixes shifts 16 arrays in a bank.

In this scenario, both $input_0$ and $input_1$ are loaded twice. We define the number of banks that share one weight cube as *weight share*. Figure 13 shows the details of the weight-share scheme. Assuming the two input data set that each one occupies 16 arrays, e.g., 16 $input_0$ subsets and 16 $input_1$ subsets, should be shifted by two weight cubes respectively. Basically, in the case of *weight share* = 1, each bank stores one weight cube. It loads all $input_0$s in $bank_0$ and move one copy to $bank_1$ in the first round. Then in the second round it replaces the $input_0$ with $input_1$ in the two banks to finish computation.

In the case of *weight share* = 2, the two banks share one weight cube. So one same weight cube is loaded in the two banks in the first round. And it loads $input_0$ and $input_1$ in the two banks respectively. In the second round, it reloads another weight cube to finish computation. Comparing to *weight share* = 1, the case of *weight share* = 2 reloads the weight cube twice instead of input data set. Since loading overhead of weight cubes is much smaller than input data, increasing *weight share* contributes to performance by reducing the loading overhead of input data. However, it also increases loading overhead of weights. Thus, there is a tradeoff, shown in the following experiments.

6 EVALUATION

To evaluate the effectiveness of DWMAcc, we studied its area, performance and power consumption and compared it to the state-of-the-art CNN accelerators, including (i) Neural Cache, the design that is close to ours; (ii) DRISA, a DRAM-based CNN accelerator; and (iii) GPU, a GPU-based CNN implementation. We followed the existing studies [9, 15, 21] and used an in-house simulator to model DWMAcc, Neural Cache, and DRISA, respectively. The simulator includes a

Table 1. DWMAcc and Neural Cache Parameters

DWMAcc		Capacity	29.75MB
Cell size	$4F^2$	CMOS Tech.	45nm
Core Freq.	2GHz	Ways	17
Banks/way	4	Arrays/bank	16
Subarrays/array	4	Subarray size	64×64
DWM Area	16.24 mm^2	Leakage	3.1 µW
Latency(ns)	2.4(Read), 5.4(Write), 0.5/shift		
Power(nJ)	0.24(Read), 0.49(Write), 0.62/shift		
Adder Unit	Dynamic power:12.7µW, leakage: 3.86µW, Latency:1.3ns, Area: 166.2µm^2, Number: 56K		
T-reg register	7.5×10^{-16}J/access, leakage: 0.89µW, Area: 2.2µm^2, Number: 238K		
Neural Cache (SRAM)		Capacity	35MB
Cell size	$146F^2$	CMOS Tech	45nm
Area	103.04 mm^2	Leakage	46.3 µW
Latency(ns)	1.5(Read), 1(Write)		
Power(nJ)	0.38(Read), 0.31(Write)		

2-core out-of-order CPU that runs at 2GHz, and a 4Gb DRAM. Similar as that in Neural Cache, we constructed an accelerator with 14 slices, i.e., a 29.75MB DWM based cache at 45nm technology. Section 4 illustrates the architecture of each single slice. We adopted a revised version of NVSim [12] to generate related parameters. We modeled the shift operation following HDART [21] and have the adder units synthesized by Design Compiler [22] with an industry library. While the capacity of one way in Neural Cache is the same to that of one way in DWMAcc, Neural Cache has a slightly larger total capacity as it has twenty ways. Table 1 summarizes the configuration details.

In this paper, we focus on accelerating CNN inference, including input/weight loading, data mapping, convolutions, pooling and FC layers. We evaluated different CNN accelerators using a set of wide-adopted CNN applications, i.e., `Alexnet`, `vgg-16`, `vgg-19` , `Cifar10` and `Mnist` [13, 20]. Of these applications, `Alexnet`, `vgg-16` and `vgg-19` are large networks while `CIFAR-10` and `Mnist` are relatively small. In the following experiments, the default is basic design without optimizations.

6.1 The Overhead

The overhead of DWMAcc comes mainly from the adder units and T-reg registers. From Table 1, T-reg occupies negligible area overhead (0.54mm^2), since a T-reg is shared by 16 RTs. The total area overhead of adder units is about 9.53 mm^2. The overhead is reasonable for a hardware PIM accelerator. Table 1 also shows that the adder and T-reg have small leakage power overhead.

6.2 Evaluating the Zero-sharing Scheme

We first studied the performance impact when adopting the zero-sharing scheme and summarized the results in Figure 14(a). `Default` and `Zero-sharing` represent the schemes that layout four and five 8-bit input values on one RT, respectively. The default head positions are at the MSB (most significant bit) as Section 5.1 shows that it is better than LSB and middle positions.

Since DWMAcc needs to prepare intermediate dot products in the 17th way for the next CNN layers, improving storage efficiency helps to keep more dot products instead of loading data from DRAM. We compare the dot product preparation overhead. From this figure, the two schemes achieves same performance in small CNNs (`Mnist` and `Cifar10`), because existing storage is

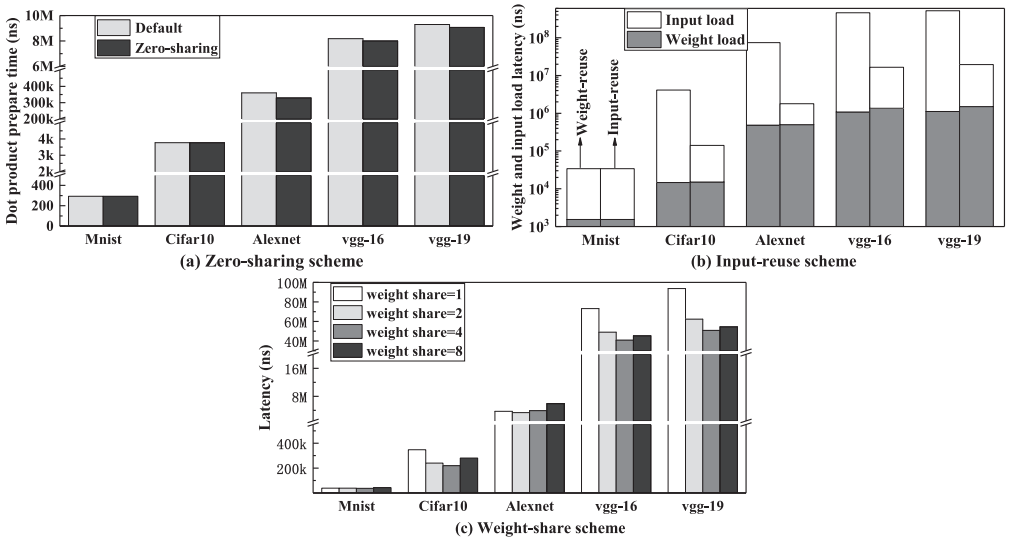

Fig. 14. Comparing the effectiveness of three optimization designs.

enough to store all intermediate results. However, `Zero-sharing` outperforms `Default` in large CNNs due to higher storage efficiency and reduces the performance overhead by up to 8.7% (`Alexnet`).

6.3 Evaluating the Input-reuse Scheme

Based on the zero-sharing scheme, we then studied the difference between weight reuse and input reuse. Figure 14(b) compares the weight and input loading latencies when we focus on weight reuse and input reuse, respectively.

From this figure, we observed that, for large CNNs, the weight-reuse scheme has slightly smaller weight loading latency while the input-reuse scheme significantly reduces the input loading latency. For small CNNs, e.g., `Mnist`, DMWAcc has enough space to store all the weights so that the two data loading schemes have the same latency. The input-reuse scheme achieves on average 77.3% overall latency degradation over the weight-reuse scheme.

6.4 Evaluating the Weight-sharing Scheme

Based on the zero-sharing and input-reuse schemes, we then evaluated the weight sharing scheme by comparing the overall inference latency when adopting different *weight share* values. The results are summarized in Figure 14(c).

The *weight share* achieves the tradeoff between the loading overhead of input and weight. From the figure, almost in all CNNs (except `Alexnet`), DWMAcc achieves the lowest inference latency at *weight share* = 4. This means in the case that *weight share* $\leq$4, although the weight-sharing optimization increases the weight loading overhead, it decreases more input loading overhead and improves the system performance. When *weight share* = 4, DWMAcc achieves up to 45.7% latency reduction (`vgg-19`).

6.5 Reducing Data-loading Overhead

Loading data to DWMAcc incurs non-negligible overhead, which includes loading both input data and weights. Figure 15 compares the data-loading overhead in the baseline design as well as the optimized design. From the figure, we observed that the optimized design has an average of 80%

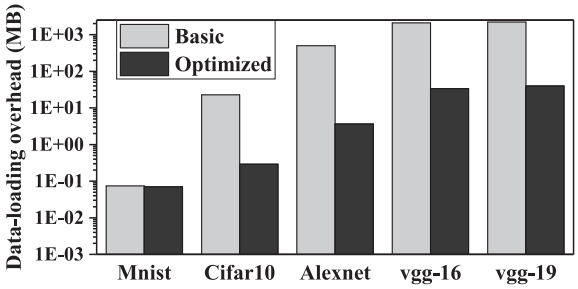

Fig. 15. Data-loading overhead.

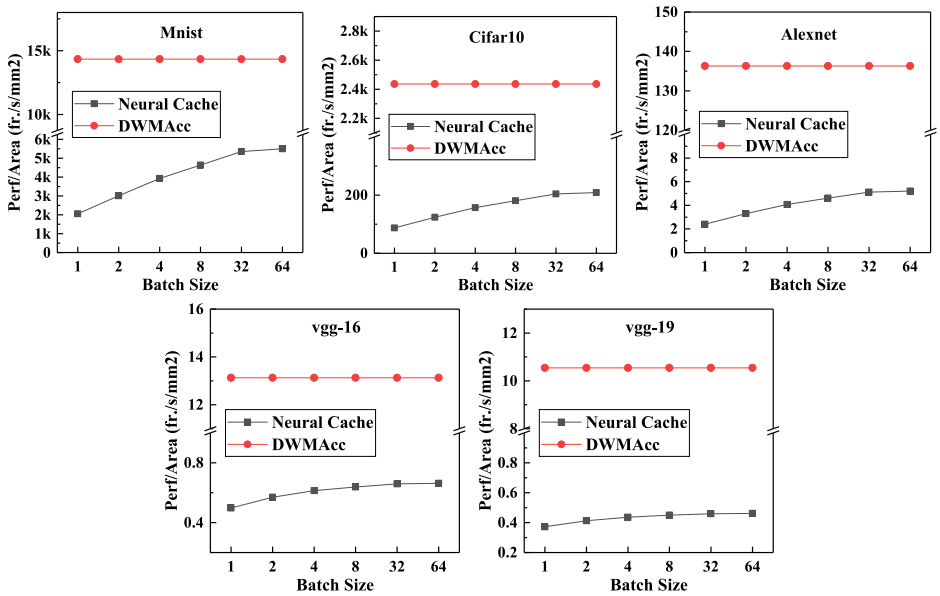

Fig. 16. Comparing the performance with Neural Cache [9].

reduction for the amount of data fetched from DRAM, which helps to achieve both performance and energy consumption improvements.

6.6 Performance

We then compared the performance between the optimized DWMAcc and Neural Cache with the results summarized in Figure 16. Since Neural Cache exploits batching to achieve higher throughput, we compared the inference performance for single frame as well as batches.

From the figure, we observed that without batching DWMAcc improves the performance by up to 56.9× (`Alexnet`). There are three reasons: (1) DWMAcc finishes multiplication in O(N) steps while Neural Cache needs $O(N^2)$ steps, where N is number of bits of the operands. (2) DWMAcc exploits weights to control head positions, which saves time from writing them in DWM domains. (3) DWMAcc exploits the high density of DWM to achieve higher capacity, i.e., higher PIM processing capability, over SRAM-based accelerators.

Neural Cache achieves higher performance improvements with increasing batch sizes, and peaks at batch size being 64. This is because Neural Cache focuses on weigh reuse and amortizes weight

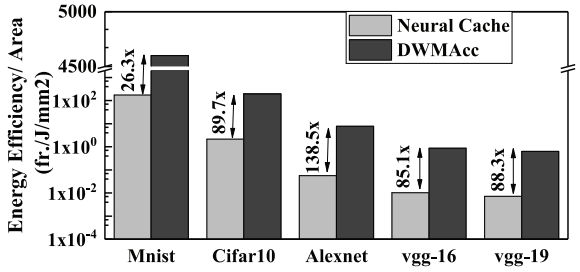

Fig. 17. Energy efficiency of different schemes.

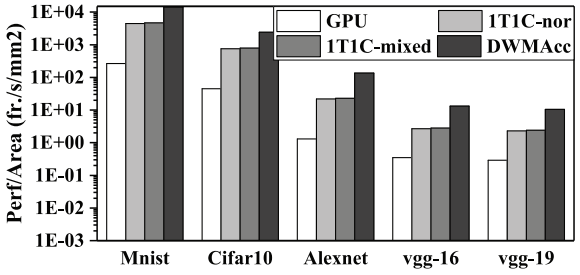

Fig. 18. Comparison of performance.

loading overhead by processing multiple input data. As a comparison, DWMAcc achieves stable performance improvement over different batch sizes, it utilizes the input-reuse scheme, so that it cannot benefit from batching. At batch size being 64, DWMAcc achieves 16.6× performance improvement over Neural Cache.

6.7 Energy Efficiency

At last, we compared the energy efficiency of DWMAcc and Neural Cache and summarized the results in Figure 17. From the figure, DWMAcc achieves higher energy efficiency than Neural Cache for all CNNs. DWMAcc improves the energy efficiency by up to 138.5× (`Alexnet`) with an average of 85.6×. There are two reasons. First, Neural Cache is an SRAM-based hardware accelerator, which consumes more dynamic energy at runtime (11.5% overhead averagely). Second, DWMAcc achieves large performance gain so that the overall energy consumption is smaller.

6.8 Comparison with Other CNN Solutions

In addition to the cache-level accelerator Neural Cache, we compared DWMAcc with other existing CNN accelerators — a GPU-based CNN accelerator and a DRAM-based in-memory CNN accelerator (i.e., DRISA [15]). For the former, we used two TITAN X (Pascal) GPUs that run at 1.5GHz [1]. For the latter, we chose two best designs of DRISA, namely `1T1C-nor` and `1T1C-mixed`, which have the highest energy efficiency and the best performance, respectively [15].

Figure 18 shows the comparison results of performance. From the figure, DWMAcc is on average 56.3× faster than `GPU`. This is because `GPU` has huge data movement overhead. Although `1T1C-mixed` is slightly faster than `1T1C-nor`, DWMAcc achieves on average 3.22× performance improvement over `1T1C-nor`. This is because DWM has fast access speed and high density.

Figure 19 summarizes the energy efficiency comparison with the results normalized to per unit area. We observed that `GPU` consumes the maximum energy during the inference process. Due to the adoption of highly energy efficient in-situ computation, the two DRISA solutions are about 25×

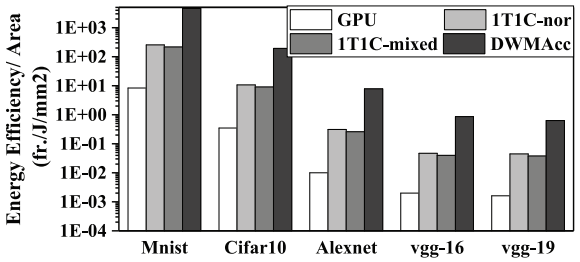

Fig. 19. Comparison of energy efficiency.

better than GPU. Comparing to the most energy-efficient DRISA solution (1T1C-nor), DWMAcc improves the energy efficiency by up to 24.1× (Alexnet) with an average of 17.7×. This is due to the low energy consumption of DWM accesses. In summary, DWMAcc achieves the best power and energy consumption among these accelerators.

7 CONCLUSION

In this paper, we proposed DWMAcc, a DWM-based accelerator with asymmetrical storage of weight and input data, to speed up ShiftNN inference on mobile devices. We designed DWMAcc at the cache level and proposed three optimizations for further performance improvement and energy consumption reduction. Our experimental results show that, on average, DWMAcc achieves 16.6× performance improvement and 85.6× energy consumption reduction over Neural Cache, a state-of-the-art PIM accelerator design.

ACKNOWLEDGMENTS

This work is supported in part by the National Key R&D Program of China under Grant NO. 2016YFB1000302, the National Natural Science Foundation of China under Grant NO. U1611261, 61872392, 61832020, 61802418, the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant NO. 2016ZT06D211, the Pearl River S&T Nova Program of Guangzhou under Grant NO. 201906010008, and US NSF CCF #1535755, CCF #1617071, and CCF #1718080. The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] 2016. NVIDIA TITAN X (pascal). http://www.geforce.com/hardware/10series/ titan-x-pasca.

[2] Mohammad Arjomand, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2016. Boosting access parallelism to PCM-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 695–706.

[3] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[4] 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.

[5] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H. Kang, and Yuan Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 109–114.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[8] Ruizhou Ding, Zeye Liu, Rongye Shi, Diana Marculescu, and R. D. Blanton. 2017. Lightnn: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 35–40.

[9] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 383–396.

[10] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine* 29 (2012).

[11] Jordan L. Holi and J.-N. Hwang. 1993. Finite precision error analysis of neural network hardware implementations. *IEEE Trans. Comput.* 42, 3 (1993), 281–290.

[12] Qingda Hu, Guangyu Sun, Jiwu Shu, and Chao Zhang. 2016. Exploring main memory design based on racetrack memory technology. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI*. ACM, 397–402.

[13] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in Neural Information Processing Systems*. 4107–4115.

[14] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[15] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 288–301.

[16] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai Li. 2014. Exploration of GPGPU register file architecture using domain-wall-shift-write based racetrack memory. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[17] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[19] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 14–26.

[20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[21] Zhenyu Sun, Xiuyuan Bi, Wenqing Wu, Sungjoo Yoo, and Hai Helen Li. 2014. Array organization and data management exploration in racetrack memory. *IEEE Trans. Comput.* 65, 4 (2014), 1041–1054.

[22] Synopsys Inc. 2017. *Design Compiler*. Synopsys Inc. https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[23] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. 2012. TapeCache: A high density, energy efficient cache based on domain wall memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, 185–190.

[24] Rangharajan Venkatesan, Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. Stag: Spintronic-tape architecture for gpgpu cache hierarchies. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 253–264.

[25] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and Weisheng Zhao. 2015. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 100–105.

[26] Xianwei Zhang, Lei Zhao, Youtao Zhang, and Jun Yang. 2015. Exploit common source-line to construct energy efficient domain wall memory based caches. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 157–163.

[27] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).

[28] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

Received April 2019; revised June 2019; accepted July 2019