DrAcc: a DRAM based Accelerator for Accurate CNN Inference

Quan Deng
College of Computer
National University of Defense Technology
dengquan12@nudt.edu.cn

Lei Jiang
Intelligent Systems Engineering
School of Informatics and Computing
Indiana University Bloomington
jiang60@ie.edu

Youtao Zhang
Computer Science Department
University of Pittsburgh
zhangyt@cs.pitt.edu

Minxuan Zhang
College of Computer
National University of Defense Technology
mxzhang@nudt.edu.cn

Jun Yang
Electrical and Computer Engineering Department
University of Pittsburgh
juy9@pitt.edu

ABSTRACT

Modern Convolutional Neural Networks (CNNs) are computation and memory intensive. Thus it is crucial to develop hardware accelerators to achieve high performance as well as power/energy-efficiency on resource limited embedded systems. DRAM-based CNN accelerators exhibit great potentials but face inference accuracy and area overhead challenges.

In this paper, we propose DrAcc, a novel DRAM-based processing-in-memory CNN accelerator. DrAcc achieves high inference accuracy by implementing a ternary weight network using in-DRAM bit operation with simple enhancements. The data partition and mapping strategies can be flexibly configured for the best trade-off among performance, power and energy consumption, and DRAM data reuse factors. Our experimental results show that DrAcc achieves 84.8 FPS (frame per second) at 2W and 2.9× power efficiency improvement over the process-near-memory design.

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have made great progress in recent years. The error rate of CNN based visual recognition decreased from 28% in 2010 to 3% in 2016, surpassing human-level performance at 5% [8]. CNNs are being integrated in modern embedded systems to address image classification and pattern recognition problems, e.g. automated driving systems. However, large CNNs could have millions of parameters and require up to tens of billions of operations for processing one image frame [5], exhibiting the need for designing hardware CNN accelerator designs to improve performance and power/energy consumption.

It is challenging to design CNN accelerators for resource limited embedded systems. FPGA-based accelerators [15] achieve good power/energy efficiency but often have low throughput due to limited memory bandwidth. ASIC based accelerators achieve high performance with energy efficiency through highly optimized computation engines, but need to use large on-chip buffers to store the intermediate results [1, 3, 11]. They consume not only large dynamic power/energy on moving data into and out of the computation engines but also large static power/energy for the large buffers [4]. ReRAM based accelerators adopt a processing-in-memory (PIM) strategy such that most computation operations are performed inside the memory arrays, which eliminates expensive data movements [19]. However, they demand large peripheral circuits such as ADC, DAC and router. ReRAM based accelerators not only face the endurance problem but also demand a special fabrication process that introduces extra cost.

Recently, several DRAM-based CNN accelerators were proposed to exploit bit operation capability inside DRAM cell arrays. They exhibit great potential for high performance and low power/energy consumption on embedded systems. These designs choose binary weight neural networks (BWN) [2, 6, 16] that shrink 16-bit or 32-bit values to two values ('-1' or '+1'). There are two choices: one is to convert all values, i.e. weights, inputs, and intermediate results; the other is to convert only the weights [2]. Most DRAM-based CNN accelerators [9] adopt the first choice so that they eliminate multiplication operations and use only XNOR operations in CNN inference. This choice suffers from accuracy loss, e.g. about 11% accuracy loss on ImageNet (top-5) [13]. In this paper, we follow the second choice and adopt a ternary weight network [20] to ensure inference accuracy. We further optimize it to achieve energy efficiency on embedded systems.

The work most related to our design is DRISA[14]. While both DRISA and DrAcc adopt DRAM-based processing-in-memory framework, DRISA is a heavyweight design. Its 1T1C-NOR variant adds one NOR gate and one latch to each bitline and a full-fledged shifter to each subarray. DRISA demands high power and has large area overhead — the area of a 4Gb 1T1C-NOR engine is close to that of 8Gb DRAM while its power consumption is more than 50W [14]. In contrast, DrAcc is a lightweight design. DrAcc relies mostly on cell operations; it places the indispensable yet less frequently used shifter outside of the cell subarrays. DrAcc adds less than 2% area overhead and consumes less than 2.5W power.

Our contributions are summarized as follows:

- We propose DrAcc, a DRAM-based CNN accelerator for embedded systems. DrAcc implements ternary weight neural networks

DAC '18, June 24–29, 2018, San Francisco, CA, USA

ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196029

[20] to achieve inference accuracy and energy efficiency. The implementation exploits the in-DRAM bit operation together with our simple enhancements.

- We propose flexible data partition and mapping strategies that match the dynamic resource availability and performance demands. DrAcc supports three modes and their mixes – the high throughput mode, the single frame mode, and the low power mode. They are specially designed to maximize system throughput, minimize single frame processing time, and minimize power consumption, respectively.
- Our experimental results show that DrAcc achieves 84.8 FPS (frame per second) at 2W and a 2.9× power consumption improvement over the process-near-memory design.

2 BACKGROUND

2.1 Ternary Weight Neural Networks

CNNs are mainly composed of convolution layers and fully connected layers. As shown in Figure 1, each convolution layer usually has 3 steps, i.e. convolution, activation, pooling. Here X is the input of one layer, and W is the weight matrix. Z is the convolution result, which enters the activation function units in the next step.

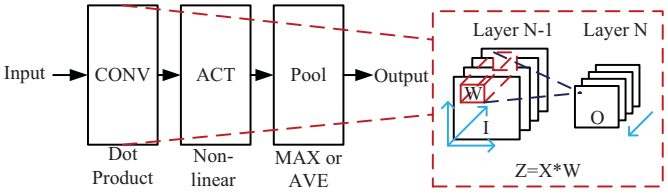


Figure 1: A CNN consists of many layers.

The recently proposed binary weight neural networks (BWNs) and ternary weight neural networks (TWNs) [2, 20] replace full precision weights with binary weights ('-1' or '+1') and ternary weights ('-1', '0' or '+1'), respectively. BWNs and TWNs achieve similar inference accuracy as that of a full-precision weight CNN. We choose TWN in this paper to exploit weight sparsity [20] for energy savings.

By converting 16-bit or 32-bit values to 2-bit ternary values, TWN achieves 8x or 16x memory space reduction. The conversion adopts a threshold-based ternary function as shown in Equation 1. Here, Δ_l is the minimum Euclidean distance between the full precision weight and the ternary weight. For normal distributions, δ_l is around $0.7E(|W|)$.

$$W_l^t = \begin{cases} +1 & w_l > \Delta_l \\ 0 & |w_l| \le \Delta_l \\ -1 & w_l < -\Delta_l \end{cases} \tag{1}$$

$$Z = X * W \approx X * (\alpha W^t) = \alpha(X \# W^t) \tag{2}$$

Equation 2 shows the flow of TWN for each convolution course. Here, X refers to the input data of each layer. Z refers to the output. $*$ refers to the inner product, while # refers to the inner product without multiplications. α is the scaling factor, which is the mean of the weights from one filter.

TWN computes inner products using accumulations only and greatly reduces the number of multiplication operations. With the only multiplication operation left in the final step of equation (2), TWNs achieves 100× or more reduction over the traditional CNN in AlexNet. TWN makes it possible to design adder-based accelerators rather than traditional multiplier-accumulator based accelerators.

2.2 DRAM

DRAM adopts a hierarchical design at the bank level. Each bank contains multiple subarrays, global buffers and decoders. Each global BL (bitline) connects several local BLs from different subarrays in different rows. The hierarchical structure helps to decrease BL length of BL and improve DRAM timing.

A DRAM cell consists of a capacitor and a transistor. The gate and drain of the access transistor connect to the WL (wordline) and the BL, respectively. It uses the voltage of charge in the capacitor to represent binary data. Ideally, it represents logic '1' when the capacitor is charged, and logic '0' when there is no charge.

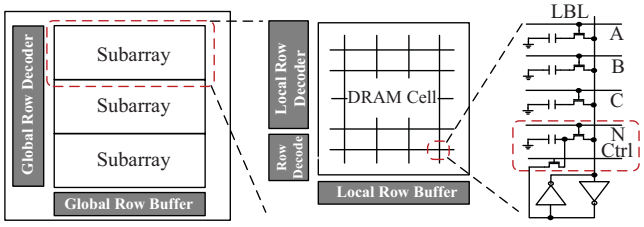


Figure 2: The hierarchical structure of Ambit [18].

2.2.1 Processing in DRAM. Seshadri *et al.* [17, 18] and Kim *et al.* [12] proposed a series of low level architecture optimizations in DRAM cell arrays, which expose the design opportunity of applying processing-in-memory strategy to DRAM cell arrays.

Our design is built on top of Ambit [18], which leverages charge sharing among different rows to conduct logic operations in commodity DRAM. Ambit uses three DRAM cells to compute AND and OR logic results, as shown in Figure 2. The bit 'C' selects the logic function while 'A' and 'B' are inputs and 'R' is the final result, as shown in Equation 3.

$$R = \begin{cases} A \,\&\, B & when\ C = 0 \\ A \parallel B & when\ C = 1 \end{cases} \tag{3}$$

Each logic operation requires three steps: charge sharing, restore and precharge. We next elaborate the operation assume ABC='011'. First, the wordlines of A, B and C are enabled while the local sense amplifier is closed. This leads to charge sharing among the capacitors of A, B and C through the local BL. The voltage goes to $2/3V_{DD}$, ignoring process variation, load in BL and other factors. Second, the local sense amplifier is activated. A voltage, which is higher than half V_{DD}, drives the local sense amplifier to be '1'. All the three DRAM cells are then restored to '1'. Finally, the local BL is precharged.

Two of the additinal rows added in Ambit subarrays are to enable NOT operation, i.e. their capacitor are connected with $\overline{BL}$. It first activates the WL to open a DRAM cell, and reads its data to the local sense amplifier. It then closes the input DRAM cell, and opens the special cell. $\overline{BL}$ writes the result back to the special cell.

All these operations are operated at the subarray level, where the global BLs are not used. They can be translated to device commands **A**ctivation and **P**recharge — 'AAP' refers to two activations back-to-back followed by a precharge. 'AP' refers to one activation followed by one precharge.

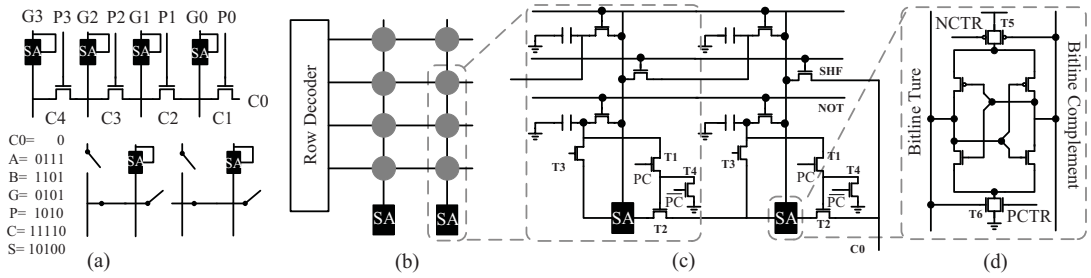


Figure 3: The DRAM-based in-memory adder mplementation.

3 THE IN-DRAM ADDER DESIGN

Since TWNs rely heavily on accumulation operation, we next devise a DRAM-based in-memory adder.

3.1 The Carry Look-ahead Adder (CLA)

In this paper, we choose to implement a carry look-ahead adder with the following algorithm steps. A and B are the inputs, C is the carry, and S is the final result. A_i indicates the i-th bit of A or B. G_i='1' indicates there is a carry to the higher bit. P_i='1' indicates it needs to propagate the carry to higher place.

$$\begin{aligned} G_i &= A_i \& B_i \\ P_i &= A_i \oplus B_i \\ C_{i+1} &= G_i \| (P_i \& C_i) \\ S_i &= A_i \oplus B_i \oplus C_i \end{aligned} \tag{4}$$

In our implementation, we exploit transmission gates to generate carry bits in carry lookahead adder in DRAM. An equivalent circuit is illustrated in the upper of Figure 3 (a). P controls the transmission gate, while G connects the input and the control of a buffer. When G='1', the buffer is open; otherwise, the buffer is closed. When P='1', G is propagated from the right to the left. In the example, we are to add two inputs A='0111' and B='1101'. C_0 is precharged to '0'. According to Equation (4), we have G='0101' and P='1010'. After changing the open NMOSs with connected lines and the closed NMOSs with disconnected lines, we get the bottom circuits in Figure 3 (a). The final carry bit is generated to be '11110'. The final result S is computed from 'P XNOR C', i.e. '10100'.

Addr.	Wordline(s)
B0	R0
B1	R1
B2	R2,R7
B3	R3
B4	R4
B5	R5
B6	R6
B7	Not
B8	R0, R3
B9	R1, R4
B10	R5, R6, R8
B11	R0, R1, R2
B12	R3, R4, R5
B13	R1, R6, Not
B14	R0, R1, R7
B15	R3, R4, R8
B16	Shift
B17	R1, R9, Not
A, D	Input Data
E0,E1	Fixed Data

AAP Operations

```
1:AAP (A,B8)        ; Copy A
2:AAP (D,B9)        ; Copy D
3:AAP (E0,B2)       ; Copy E0
4:AAP (E1,B10)      ; Copy E1
5:AP (B11)          ; G=A&B
6:AAP (B12,B7)      ; M0=!(A+B)
7:AAP (B13,B7)      ; P=A xor B
8:AAP (B0,B16)      ; Propagate C
9:AAP (B16,B9)      ; shift C
10:AAP(B7,B8)       ; Copy P
11: AP (B14)        ; M1=P&C
12:AAP (B15,B7)     ; M2=!(P+C)
13:AAP (B17,B7)     ; S=P xor C
```

Figure 4: The AAP operations of CLA.

We enhance Ambit [18] to implement our proposed in-memory CLA adder. Ambit adds eight extra rows for basic logic operations, i.e., AND, OR and NOT. Two of the eight rows are introduced to support NOT logic. The DRAM cells in those rows have extra connections with $\overline{BL}$, which are the same as T3s in Figure 3(c). DrAcc makes the following enhancements: (1) DrAcc changes the wire connection of one special row so that it can be used for carry shift instead of not-logic. We name the special row as SHF. (2) An extra path to T2 is added in the NOT row, i.e. T1 and T4 in Figure 3 (c). (3) DrAcc slightly changes the peripheral circuit. It adds one transistor to deliver the propagation signal, i.e. T2 in Figure 3(c); It adds two transistors in SA to provide an extra enable port for SA,i.e. T5 and T6 in Figure 3(d). Considering the hardware overhead, change (1) is minimal, i.e, the overhead of SHF and DRAM cell are similar; change (2) increases 3 more transistor per column in DRAM; change (3) is made in the peripheral circuits and thus is less critical. We name the specific rows for processing-in-DRAM as reserved rows. DrAcc adds less than 2% area overhead to the cell subarray, which is comparable to 1% area overhead of Ambit [18].

Figure 4 illustrates the DRAM command details of one accumulation operation. 'RN' refers to the N-th row in the reserved rows; 'Shift' refers to the SHF row, while 'Not' refers to the NOT row. AAP(A,B8) means the wordlines of A in the Addr. table on the left of Figure 4 is first open, then the wordlines of B8. Among the 13 commands, the first four copy the corresponding operands into the reserved rows to prevent the original data from being destroyed. Then it exploits charge-sharing to generate 'G' and 'P'. 'C' is generated through the propagation. After shifting 'C', an XOR operation is performed over 'P' and 'C' to generate the final result 'S'.

There are three steps in the propagation operation. (1) We enable NCTR and PCTR to activate SAs, read G out, and then deactivate NCTR and PCTR. SA remains being activated only when G is '1' and deactivated otherwise. (2) we activate PC and open gate T1. P is delivered to the gate of T2. (3) we close T1 and reactivate NCTR and PCTR to restore Carry to the corresponding row.

3.2 The Benefits of In-memory DRAM Adder

3.2.1 Benefit 1: it mitigates data movement and improves performance and computation efficiency. A major challenge to implement in-DRAM adder is the data movement between data block and logic block. For a processing-near-memory design, each operation needs to read out all operands from the memory and then write the result back to the memory. Since most operands are reusable, a lot of these movement operations are unnecessary.

DrAcc adder leverages the temporary data during accumulation to mitigate data movement. It copies data from one row to multi rows as the inputs of different logic operations, which avoids redundant copy of the same data. DrAcc needs 11 AAP and 2 AP commands to finish one 16-bit or 32-bit add. As a comparison, DRISA 1T1C-NOR design needs 21 AP commands. Given AAP and AP have about the same latency [18], DrAcc is about 1.5x faster.

3.2.2 Benefit 2: DrAcc adder decouples the voltage drop and the length of transmission line. DrAcc implements a transmission line based carry-look-ahead (CLA) adder. Such a design is less popular as a stand-alone implementation, due to its long latency and voltage drop along the transmission line. DrAcc leverages the local sense amplifier to drive adjacent bits. Together with the sources of transmission lines, DrAcc accomplishes add within short latency. Figure 5 shows the H-SPICE simulation results for a 16-bit propagation operation. We have G, P, C_{in} being '0000 0000 0000 0010', '1111 1111 1111 1100', and '0', respectively. As shown in the figure, the final 'C' is '1111 1111 1111 1110', showing no voltage drop in the operation. The average latency for one column propagation is 0.25ns. Therefore, it takes up to 4ns for processing 16-bit data.

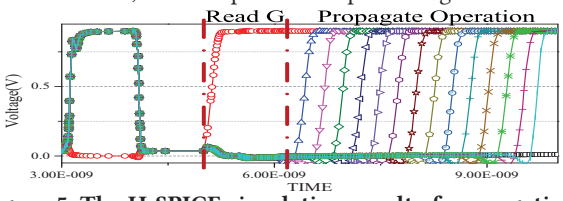


Figure 5: The H-SPICE simulation result of propagation.

4 THE DRACC ARCHITECTURE

4.1 An Overview

While adopting ternary weights eliminates most multiplication operations, we still need to perform a few, i.e. multiplying α in Equation (2). Since the DRAM-based PIM subarrays do not have multiplier functional unit, we decompose multiplication to shift and add operations, and perform shift operations using the logic layer. The shift operation is then deferred after pooling, which is referred to as *post-conv* in Figure 6. The rest of convolutional layer is untouched, referred to as *pre-conv*. The split is viable because multiplying α is a linear operation such that the computation result stays unchanged. Since pooling reduces the size, the split reduces the total number of shift operations.

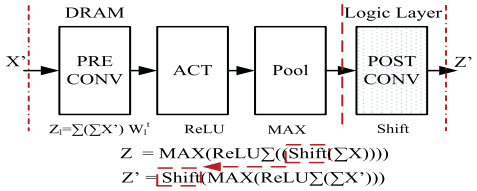


Figure 6: The workflow of DrAcc layer processing.

To summarize, CNN layer processing is implemented in four steps in DrAcc: pre-conv, activation, max pooling, and post-conv. The first three steps are implemented in the DRAM layer while the last step uses the logic layer.

4.2 The DrAcc Hardware Implementation

Figure 7 illustrates the hardware enhancements to enable DrACC: (1) we enhance the computation rows in the each DRAM subarray; (2) we enhance the memory controller to enable PIM processing; (3) we integrate a flag buffer for holding PIM instructions and their operand row addresses; (4) we integrate a shifter in the logic layer to support decomposed multiplication operations.

In the pre-conv step, the controller adds the shifted results from the post-conv step of its preceding layer, and then interprets the PIM instructions in the flag buffer. Depending on the ternary weights, either accumulations or subtractions are performed. In the activation step, there are two candidate functions — Rectified Linear Unit (ReLU) or Sigmoid. ReLU chooses the maximum between 0 and the input while Sigmoid uses a complicated non-linear function [10]. We implement ReLU in DrAcc, which sends 0 if the sign bit is '1', indicating a negative value. In max pooling step, we downsample the inputs by selecting the maximum data in every filter. The subtraction operation is processed in-memory. In post-conv step, the controller first determines the scaling factor and then send the scaling factor as well as the pooling results to the shifter in the logic layer. The results are sent back to the DRAM layer, which are then summed up at the beginning of the next layer.

Parallel optimization. DrAcc operates at the page level such that it demands massive concurrent computation to maximize resource utilization. Adopting the TWN algorithm helps to orchestrate the shift and accumulation operations. For further parallelism improvement, we reorder the input data and place the data with the same weight (but from different threads) in the same row. We keep their correlation to ensure correctness.

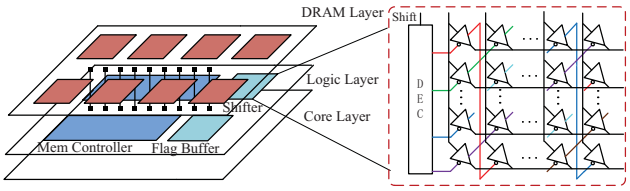


Figure 7: The hardware enhancements for DrAcc.

5 THE DIFFERENT WORK MODES IN DRACC

Since DrAcc exploits DRAM subarrays for both data movement and computation, maximizing the performance of one often hurts the performance of the other. In this section, we elaborate how to adopt data partition to balance the resource allocation and to tradeoff among accelerator throughput, single task performance, and power consumption.

5.1 Data Partition Choices

For each layer, there is a three-dimensional input matrix and multiple weight vectors. We differentiate two directions for the matrix — the Y axis direction, and the XZ plane. We next discuss three data partition choices.

- YP-partition. We partition both the input matrix and the weights along the Y direction, i.e. one thread is split to multiple sub-threads with each handling a portion of data along the Y direction. Each input data block corresponds to multiple pruned weight vectors. The number of pruned weight vectors is the same with the original size of weight vectors. YP-partition allocates more hardware resources to speed up thread computation. Given the sub-threads are to compute different columns, we need to conduct extra effort, e.g., data movements, to compute the final results.
- XZP-partition. We partition the input matrix along the XZ direction and let each partition form an independent thread. One thread in the XZ partition computes the final result. In each CNN layer, the stride of a kernel, which is the step in the weight matrix, is smaller than the dimension size of the weight matrix. Two adjacent threads share data in the initial input matrix. When

adopting XZP-partition, we need to copy the overlapped data multiple times to ensure thread independence. The additional data movement tends to degrade the throughput.

- WP-partition. We copy multiple input data samples with different weight vectors. A set of weight vectors with entire input data forms a thread. WP-partition avoids the data overlap issue in YP- and XZP- partitions. To mitigate weight inconsistence in one row (i.e., some are adds, some are subtractions, while others are no-ops), we use weight grouping to improve parallelism. For WP-partition, each thread needs the entire input, which increases the data movement overhead.

5.2 Working Mode Optimization

An embedded system may face different requirements when being deployed in different environments. We next elaborate how DrAcc can adjust its working mode to meet such requirements.

- High Throughput Mode. This mode keeps the DRAM-based adder busy and avoids data transfer latency, i.e. decreasing the data movement and improving the data reuse rate. For this purpose, restricting data partition helps to improve the system throughput, which is achieved with increased computing latency of each layer.
- Low Power Mode. This mode decreases the number of active pages to avoid partial computation. This is because the minimum computing unit in DrAcc is a DRAM page — all bits participate in computing even if there is only one operation. We first compare the power of the compute and the data movement. If the data transfer power is higher, XZP, WP and YP partitions can be applied to fulfill the partial active pages. Otherwise, we search for the best configuration. The order is decided by the power overhead of each partition.
- High Speed Mode. XZP, WP and YP partitions may be applied to increase the number of parallel threads. We achieve the best speedup when this number is large or we reach the resource limitation.

By mixing the above three modes, DrAcc can flexibly adjust its acceleration configuration to match the application's demand.

6 EVALUATION

We use Design Compiler and H-spice for the hardware simulation, and CACTI to simulate the DRAM structure. The neural network benchmarks are Mnist, Cifar10, Alexnet, VGG16 and VGG19. The baselines are DRAM_L, Neurocube, Eyeriss and ShiDianNao. DRAM_L is to simulate Drisa in our design flow. It adds basic logics in each DRAM subarray and uses the CSA algorithm for accumulation. Table 1 presents the configuration details.

Table 1: The Configuration of DrAcc

DRAM Architecture	Wide IO2	DRAM Tech.	25nm
Bank Size	32	Channel Size	8
Memory Capacity	8 Gb	Subarray Size	512x512

6.1 Performance Evaluation

Table 2 and 3 compare the performance between DrAcc and DRAM_L under two working modes. We compared both system throughputs (i.e., frame per second FPS) and the processing latency of a single frame (in ms). From the tables, DrAcc outperforms DRAM_L for both modes. The difference is small for deep neural network when adopting the high speed mode. That is because large neural networks, such as VGG19, have a large number of parallel threads that demand data movements. It is the limited data bandwidth that manifests the most critical performance bottleneck.

Table 2: Performance in High Throughput Mode

	DrAcc(FPS)	DRAM_L(FPS)	DrAcc(s)	DRAM_L(s)
MNIST	7697.4	4948.6	4.25	6.62
Cifar10	6008.4	3887	5.44	8.42
Alexnet	84.8	53.8	386.4	608.6
VGG16	4.8	3.08	6772.3	10627
VGG19	4.05	2.5	8074.5	12676

Table 3: Performance in High Speed Mode

	DrAcc(FPS)	DRAM_L(FPS)	DrAcc(ms)	DRAM_L(ms)
MNIST	142.8	121.4	7.0	8.239
Cifar10	120.5	104.2	8.3	9.69
Alexnet	3.63	3.55	275	281
VGG16	0.3	0.3	3282	3283
VGG19	0.25	0.25	3933	3933

In DrAcc, when allocating more hardware resources to accelerate one single frame, we shall have a large number of extra data movements or computing operations. There is a correlation between system throughput and single frame latency. For Alexnet in DrAcc, the highest throughput is 84.8 FPS, while its corresponding latency to process a single frame is 386.4s. When pushing the highest speed for a single frame to 275ms, we see a great degradation in throughput — the corresponding throughput is only 3.63 FPS. Therefore, it is important to choose an appropriate working mode to meet the application requirements. We leave the search of the best working mode as our future work.

6.2 Area Overhead

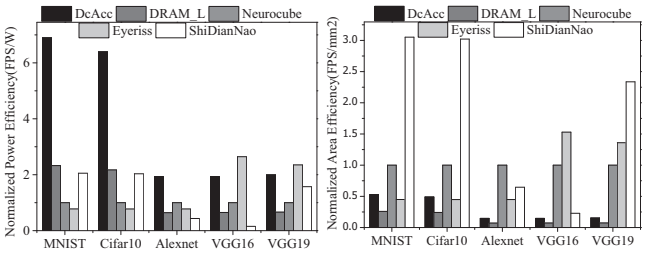
DrAcc is built on Ambit[18], of which the overhead is less than 1%. We change a wire connection in the DRAM cell layout and add two transistors per column. To support CLA adder, we have a simple modification of the subarray level SA, which adds four more transistors in each SA. The area of the enhanced SA in DrAcc is 36% larger than that of Ambit. Given SA occupies less than 15% DRAM die area, the overall area overhead of DrACC is around 2% over the commodity DRAM.

At the logic layer, DrAcc has a 5Kb weight buffer and 5K shifters. The shifter is implemented similarly as that in DRISA[14]. The total area is $0.01mm^2$, which is negligible in DRAM.

6.3 Power and Area Efficiency

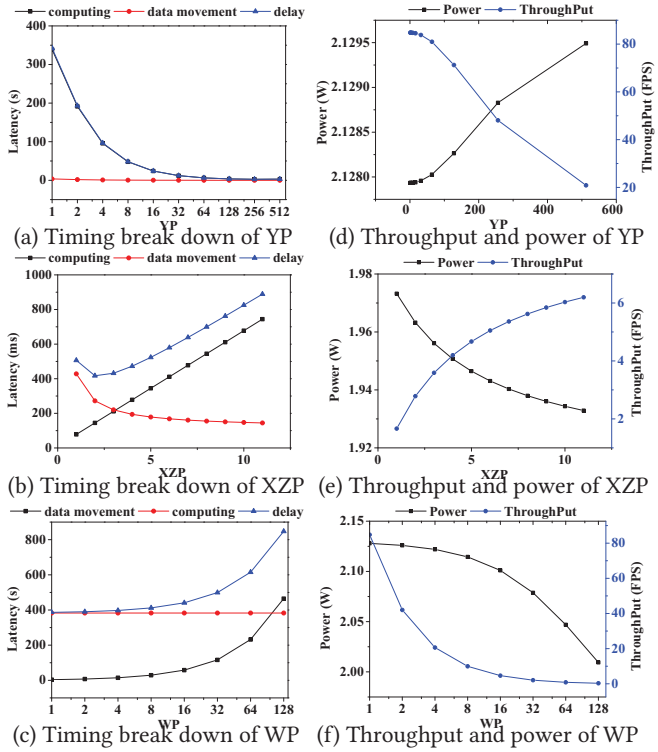
Figure 8(a) compares the power efficiency of different designs. The results are normalized to that of Neurocube. DrAcc achieves high power efficiency for small or large image recognitions. It achieves the best efficiency except for VGG 19 and VGG 16 where Eyeriss is slightly better. Note, Eyersiss is not a standalone accelerator. The power of additional hardware for enabling Eyersiss is not evaluated. On average, the power efficiency of DrAcc is 3.8× that of Neuocube and 2.9× that of DRAM_L.

Figure 8 (b) compares the area efficiency of different schemes. The area efficiency of DrAcc is low, which is only higher than DRAM_L. This is because DrAcc is an in-memory accelerator. We adopt a conservative parallelism at the bank level, which limits the number of working pages. As a standalone accelerator, DrAcc



(a) Power Efficiency (b) Area Efficiency

Figure 8: The efficiency analysis.



(a) Timing break down of YP (d) Throughput and power of YP

(b) Timing break down of XZP (e) Throughput and power of XZP

(c) Timing break down of WP (f) Throughput and power of WP

Figure 9: The partition analysis.

avoids the memory constraint. Furthermore, the add operations in DrAcc only need local operations like activation and precharge. Therefore, DrAcc is compatible to more aggressive designs that exploit subarray level parallelism. Potentially, DrAcc can achieve 16× speedup, which is 2.5× of the best in Figure 8 (b).

6.4 Programmability of DrAcc

Figure 9 evaluates the programmability of DrAcc. By choosing different data paratition and working modes, DrAcc can be optimized for high throughput, low latency, or low power. We explore the design space in this evaluation. The x axis represents the partition number. Figure 9 (a) shows that YP helps to decrease the computing latency. There is a steady tail for delay because YP introduces additional computing operations. When the saved time is comparable to that to perform extra computation, there is no overall further improvement.

Figure 9 (b) evaluates the impact of XZP partition. The single frame processing latency can be reduced into several hundreds of milliseconds, as a comparison of several hundreds of seconds in Figure 9(a)(c). The minimal delay appears when the data movement and the computation can be fully overlapped. Figure 9 (c) evaluates the impact with WP partition. The computing latency is steady with different WP partitions. The latency of data movement increases with increasing partition numbers.

Figure 9(d)(e)(f) show that there is a tradeoff between the power and the throughput. A higher throughput often leads to a lower power consumption. Another observation is that a high throughput cannot guarantee low latency. The latter is often achieved by allocating more resources, which tends to decrease throughput.

7 CONCLUSION

In this paper, we propose DrAcc, a DRAM-based accelerator for accuracy inference. A CLA adder is implemented in DRAM with a light modification. A data partition strategies is provided to balance system performance, single frame performance and power consumption. We achieve 84.8 FPS for Alexnet with 2W and average 2.9× power efficiency over the process-near-memory design.

8 ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation #1617071. The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Y. Chen, *et al.* Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE Journal of Solid-State Circuits*, 2017.
[2] M. Courbariaux, *et al.* Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.
[3] Z. Du, *et al.* Shidiannao: Shifting vision processing closer to the sensor. In *ISCA*, 2015.
[4] M. Gao, *et al.* Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.
[5] V. Gokhale, *et al.* A 240 g-ops/s mobile coprocessor for deep neural networks. In *CVPR Workshops*, pages 682–687, 2014.
[6] M. Gupta, *et al.* Binary neural networks. In *Static and Dynamic Neural Networks: From Fundamentals to Advanced Theory*, pages 507–577.
[7] K. He, *et al.* Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *arXiv:1510.00149*, 2015.
[8] K. He, *et al.* Deep residual learning for image recognition. In *CVPR*, 2016.
[9] L. Jiang, *et al.* XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs. In *ISLPED*, 2017.
[10] R. Jozefowicz, *et al.* An empirical exploration of recurrent network architectures. In *ICML*, 2015.
[11] D. Kim, *et al.* Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.
[12] Y. Kim, *et al.* A case for exploiting subarray-level parallelism (salp) in dram. *ISCA*, 2012.
[13] F. Li, *et al.* Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.
[14] S. Li, *et al.* Drisa: A dram-based reconfigurable in-situ accelerator. In *MICRO*, 2017.
[15] Y. Li, *et al.* A 7.663-TOPS 8.2-W Energy-efficient FPGA Accelerator for Binary Convolutional Neural Networks. In *FPGA*, 2017.
[16] M. Rastegari, *et al.* Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.
[17] V. Seshadri, *et al.* Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization. In *MICRO*, 2013.
[18] V. Seshadri, *et al.* Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *MICRO*, 2017.
[19] A. Shafiee, *et al.* ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *IEEE Press*, 2016.
[20] C. Zhu, *et al.* Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.
[21] Z, Liu, *et al.* Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2017.
[22] Q, Lan, *et al.* High Performance Implementation of 3D Convolutional Neural Networks on a GPU. *Computational intelligence and neuroscience*, 2017.