ROC: DRAM-based Processing with Reduced Operation Cycles

Xin Xin
Electrical and Computer Engineering Department
University of Pittsburgh
xix59@pitt.edu

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

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

ABSTRACT

DRAM based memory-centric computing architectures are promising solutions to tackle the challenges of memory wall. In this paper, we develop a novel design of DRAM-based processing-in-memory (PIM) architecture which achieves lower cycles in every basic operation than prior arts. Our small yet fast in-memory computing units support basic logic operations including NOT, AND, and OR. Using those operations, along with shift and propagation, bitwise operations can be extended to word-wise operations, e.g. increment and comparison, with high efficiency. We also optimize the designs to exploit parallelism and data reuse to further improve the performance of compound operations. Compared with the most powerful state-of-the-art PIM architecture, we can achieve comparable or even better performance while consuming only 6% of its area overhead.

1 INTRODUCTION

Modern system performance is hindered by memory subsystem, known as "the memory wall", because of the high cost of data movement [1, 2]. Particularly, for big-data application, the limited bandwidth of the off-chip bus between memory and processor cannot meet the increasing demand of data, exacerbating the memory access latency. Even when the data is moved into the cache, the inefficient reuse of cached data can also aggravate data movement. In addition, a significant amount of power during data movement is consumed, leading to system energy inefficiency.

To tackle this problem, near-data processing (NDP) and processing-in-memory (PIM) based on DRAM technology have been proposed as promising alternative solutions. In NDP, logic units (LU) are built near DRAM cells, such as DRISA [3], Automata [4] and HMC [5, 6], which can achieve high performance due to the extreme processing speed in LU. However, LUs usually occupy large die area, which decreases the density of memory. For example, the adder LU in DRISA takes 51% area, routing matrix in Automata occupies about 30% of the chip [4]. Even HMC has an entire die for logic, the available area is limited for additional LUs, as memory controllers, I/Os etc. are already taking much die area. Besides, large area of LUs increase the complexity for DRAM technology, as LUs and DRAM cells are usually not compatible in technology process [7]. In PIM, logic functions are directly integrated into the DRAM arrays, hence the name processing *in* memory. Examples of PIM include RowClone [8] and Ambit [9]. RowClone performs bulk copy and initialization by defining a back-to-back activation commands with negligible area cost. Ambit implements basic logical operations based on the charge sharing mechanism with only two additional access transistors per column of cells. However, the performance is impeded by multi-commands in DRAM, as the latency of each command in DRAM is usually much longer than CPU operations. For example, an XOR operation requires 7 commands (or cycles), totaling ~350ns with a 49ns cycle time in Ambit. Meanwhile, RowClone and Ambit are more suitable for specific applications, such as bulk copy and bitwise logic operations, so they are not yet sufficiently general. In summary, NDP and PIM are complementary approaches. PIM embeds operations inside DRAM original structure, which saves real estate for more logic design in NDP. DRISA is such an example that takes advantage of Ambit's basic logic operation to build more functional DRAM-based accelerator.

To address the aforementioned latency challenges in PIM operations, a direct way is to down-size DRAM arrays. Previous work showed that reducing the size of subarray or shorting the bitlines can reduce DRAM access latency, but at a cost of memory density [3, 10] which is critical to DRAM vendors. Another approach is to increase bank-level parallelism by activating more banks at the same time. But this could exceed the maximal limit of current draw from a single chip, as defined by t_{FAW} parameter for DRAM. In this paper, we propose another improvement method by minimizing the cycles of a logic function, e.g. AND. We design bitwise logic operations with reduced operation cycles, termed ROC, in each DRAM subarray. Taking a further step, we enhance ROC to realize more compound functions, while minimizing operation cycles and area overhead. The contributions of this paper are summarized as follows:

- We proposed a new method to construct logic operations by using a diode-connected transistor inside a DRAM cell. This method requires fewer cycles for implementing basic logic functions, AND, OR and NOT than previous work.
- For more compound logic operations such as XOR and XNOR which are built up from basic functions by composition, we exploit parallel operating opportunities to speed up computing.
- We design two types of propagation operations, uni-directional and bi-directional propagation, to serve more application functions in enhanced ROC which can execute Shift, Substitute, Compare, and Increment operations in multiple propagating paths. Leveraging parallel operating opportunities, we also reduce the number of cycles in these operations.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317900

- We further optimize propagation-based operations by re-ordering and combining commands sequence, which further trim down the operation cycles. This optimization method can be used to improve the execution of more complex Boolean expressions.

2 BACKGROUND AND RELATED WORK

A DRAM chip has hierarchical architecture from banks to subarrays. Each bank can be operated by the memory controller independently. Multiple subarrays are connected by the global bitlines in a bank. Each subarray contains a matrix of DRAM cells which are built by an access transistor and a capacitor (1T1C). Multiple cells are connected by a local bitline with a sense amplifier, which also acts as a row buffer [9, 11].

The DRAM data read process can be divided into two stages: precharge and activate stage. In the precharge stage, bitlines and sense amplifiers are set to $1/2V_{dd}$. Once entering the activate stage, bitlines are first released while the target cells are accessed. Then, charge is shared between cell and bitline parasitic capacitance. After the charge sharing phase, sense amplify (SA) is enabled to sense the slight voltage variation caused by charge sharing. Finally, the sensed voltage is amplified by SA and restored to the target cells, known as the restore phase.

RowClone [8] implements data copy between different rows via simultaneously opening the target row when restoring data to the original row. This operation contains two back-to-back activations followed by the precharge stage, referred as Activate-Activate-Precharge (AAP) primitive in [8].

Ambit, shown in Figure 1(a), modifies the charge sharing phase by opening triple DRAM cells on the same bitline at the same time [9]. If at least two of the three cells, termed A, B, and C, are 1s, the voltage on the bitline will be above $1/2V_{dd}$ after charge sharing. Otherwise, the voltage will drop below $1/2V_{dd}$ when only one or none of them are 1s. The result after sensing in SA can be written as: R = AB + BC + AC. If we define C = 1 in advance, the operation will perform A OR B. Likewise, A AND B can be performed with C = 0. For completeness, the NOT operation is implemented with the help of one additional access transistor. Benefiting from the high parallel actions of DRAM cells in a row, logic computation of large bit-vectors could be significantly accelerated.

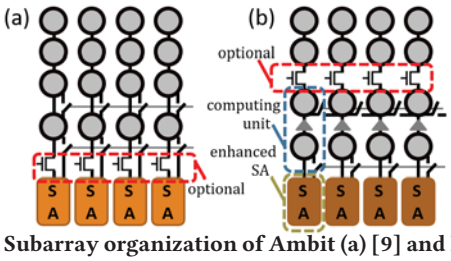

Figure 1: Subarray organization of Ambit (a) [9] and ROC (b)

The accuracy of charge sharing among triple cells is jeopardized by the initial charges remained in cells [12, 13]. Thus, to implement an accurate AND or OR logic, Ambit first copies two operands, A and B, to bottom rows near SA to keep them nearly fully charged. It also needs to write 1 or 0 to C as a preceding definition, followed by calculating the result via the charge sharing mechanism. This process involves four cycles in total. Although Ambit allocates 8 rows nearby SA for logic operations to reduce the preceding write of 1s or 0s, it is still not efficient to deal with a complex Boolean expression. For example, it carries out XOR and NXOR using 7 cycles.

Deng. etc. modified several subarray rows with extra transistors to improve the performance of Ambit based accelerator in CNN calculation [14]. However, the multiple cycles of an operation in processing still hinders the speed. For example, it needs 13 cycles to complete an addition operation, which amounts to $\sim 630ns$ with 49ns cycle time [9]. In addition, the additional transistors reduces area efficiency, which also increases design complexity for DRAM fabrication process.

3 DESIGN OF ROC

Instead of utilizing a charge sharing method, our proposed ROC, as shown in Figure 1(b), implements logic functions based on a proposed computing unit. This can achieve high speed and high area efficiency, as well as avoiding the inaccuracy caused by triple-cell charge sharing. Further, we improve our design with an enhanced SA to support for more complex functions such as compare and increment.

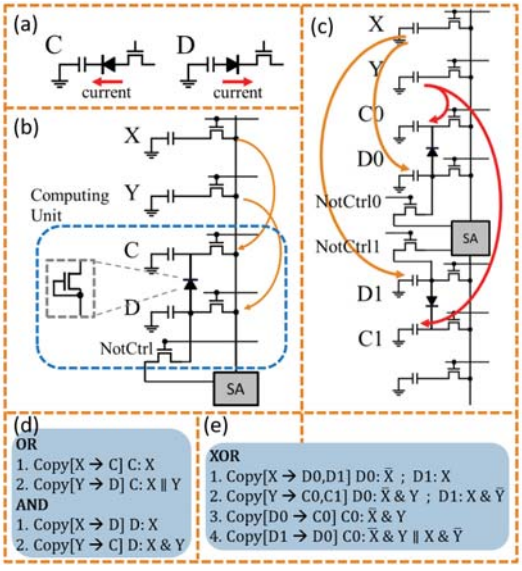

Figure 2: The design of ROC.

3.1 Basic Bitwise Logic Operations

The key idea of ROC is to leverage the character of a diode in implementing a logic operation. As shown in Figure 2(a), no matter what the initial state of cell C is, it will be charged to 1 once the transistor is connected to V_{dd}. Likewise, cell D will be 0 when the transistor accesses ground. Cell C and D behave like OR and AND logic respectively. That is, only 1 can be written into C, and only 0 can be written into D. Based on this observation, we add a diode-connected transistor between two DRAM cells, as shown in Figure 2(b), to implement the logic operations. For example, to perform logic X+Y, we first copy X to cell C, then copy Y to cell D. If Y is 0, C will remain the value of X. Otherwise C will be 1. Thus, C stores the OR result of X and Y. On the other hand, if X is first copied to cell D, followed by copying Y to cell C, then D

could be 1 only if both X and Y are 1, which effectively performs as an AND operation. Figure 2(d) shows the commands for OR and AND. The first word in the command indicates the command type. The "[]" part shows the operating source and destination, marked by an arrow. The rest part denotes the result of the command. In summary, for OR and AND operations, ROC takes only two copy commands to compute the result. This design is also area efficient, requiring only two cells during an operation

For completeness, we attach one access transistor (NotCtrl) at the bottom of a column, same design as in Ambit. We term cell C, D and NotCtrl as the computing unit (CU) in the following discussions. All further computing functions are built upon this unit.

3.2 Parallelizing Computing

As SA in DRAM is shared by bitline and its complement ($\overline{\text{bitline}}$), we build the same CU in $\overline{\text{bitline}}$ (Figure 2(c)). Hence, one SA connects to both CUs on the pair of bitlines. Because each CU can execute basic logic independently, we can parallelize AND, OR, and NOT operations in compound logic calculations, such as XOR and XNOR. This significantly saves operations cycles for them. Specifically, we can execute XOR in only 4 cycles, as shown in Figure 2(e). Note that X is copied to D0 and via NotCtrl0 and NotCtrl1, so that D0 = $\overline{X}$, D1 = X. XNOR has the similar operation steps, except copying X and $\overline{X}$ to D0 and D1 in the first cycle.

ROC is parallelizable due to the integration of a diode-connected transistor. Charge sharing method in previous design, however, occupies the bitline during the entire calculation of the logic, making it unsuitable for parallel operations. Our design opens up the potential to build more CUs per bitline to enhance parallelism. Due to the constraint of area overhead, we only exploit the dual CU design in this paper.

3.3 Enhanced ROC with Propagation and Shift

To extend bitwise Boolean operations into word-wise computing, such as comparison and increment, we introduce another primitive operation in ROC, **bit propagation**, which transmits a bit '1' to all bits to its left (or right) by changing their values into a '1'. We design both uni-directional and bi-directional propagation, and define that a word containing all '0's remain all '0's after any types of propagation. These propagation operations are implemented by connecting bitlines along the row direction and enhancing SAs to receive a propagated logic '1' in the restore phase. Figure 3(a) shows the enhanced CU with a new transistor MS that connects a cell to a neighbor bitline. It has two connection modes. One is turning on MD and MS to build the uni-directional connection with the help of diode-connected transistors, where logic '1' could only transmit in one direction, as shown in Figure 3(b). The direction can be defined by different layout of MS. We build CU connected to bitline with left-direction propagation, CU connected to $\overline{\text{bitline}}$ with right-direction propagation. The second mode is to construct a bi-directional connection by setting MC and MS on. Then logic '1' could be transmitted to both directions as shown in Figure 3(c). In addition, a new operation, **shift**, can also be done by activating MS to enable a cell accessing its neighbour bitline during the restore phase.

The enhanced SA is illustrated in Figure 3(d) in which two transistors, PsetL and PsetR, are added in p-latch pair. In a normal mode, PsetL and PsetR are both enabled. In a propagation mode, only one of them is enabled. Figure 3(e) shows the structure of SA when only PsetL is enabled. It cannot drive $\overline{\text{bitline}}$ to 1 when bitline is changed to 0. In other words, it holds a stable state when bitline is 1 and $\overline{\text{bitline}}$ is 0. Under an opposite condition, it has a changeable state. Therefore, logic 1 can be transmitted to a SA when its bitline is 0.

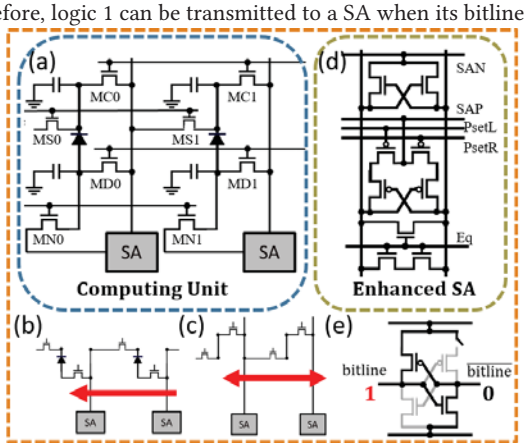

Figure 3: Structure of enhanced ROC

3.4 Compound Operations

With enhanced ROC, we develop new functions, such as **compare** and **increment** built on propagations. In interest of space, we elaborate only comparison and increment here. More functions such as **substitute** can be developed with proper operation sequences.

A comparison can be done by subtracting two words and examine the sign of the result. Another method is to make a comparison bit by bit from MSB to LSB, until the first mismatched bit is found. We adopt the second, more lightweight, way but avoid the bit by bit comparison and use our uni-directional propagation instead. As shown in the COMPARE part of Figure 4, we first calculate $X\overline{Y}$ and $\overline{Y}X$ in parallel. Then, we right propagate (PrgR) them to create blocks of '1's in each word. The larger or smaller value of X and Y is now indicated by the length of the '1'-block, represented by Tmp1 and Tmp2. If the '1'-block in Tmp1 is shorter, X is smaller and vice versa. Next, we compute Tmp3=$\overline{\text{Tmp1}}$×Tmp2. If X≥Y, Tmp3 contains all '0's. Otherwise, Tmp3 contains some '1's. By a bi-directional propagation (PrgD) on Tmp3, the result S contains either all '0's (Y is smaller) or all '1's (X is smaller). To restore the original value of X (or Y), we simply perform $XS+\overline{Y}\overline{S}$.

The comparison function above is completed in 6 cycles including 3 regular cycles and 3 propagation cycles, as shown in the "original command list" in the COMPARE part of Figure 4. The 6-command operation have already offered better performance than previous designs such as Drisa [3] and Dracc [14], but it can be further optimized to 4 commands, as discussed in section 3.5.

To design an increment function, we first make an observation that every increment just modifies the continuous 1s in LSBs of a word. For example, a word of ...0111 is turned into ...1000 after one increment. All leading bits remain unchanged. If the LSB is 0, the increment operation just changes it to 1. Based on this observation, we implement the increment function by identifying the

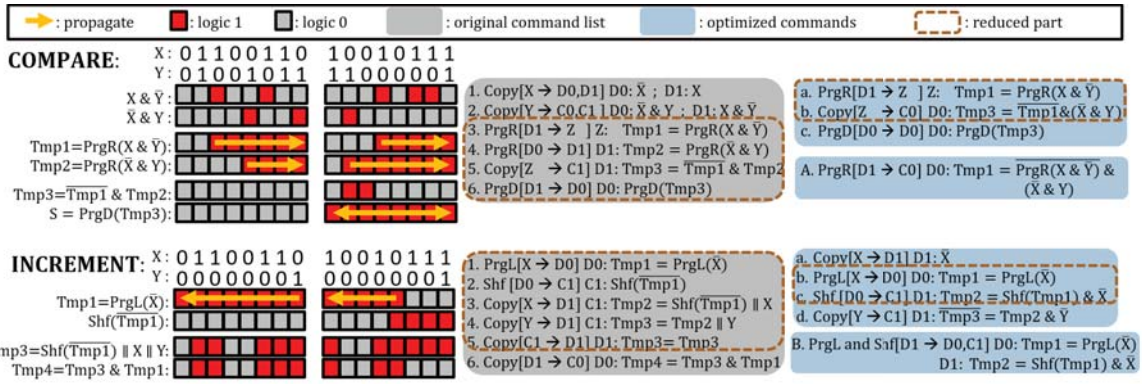

Figure 4: Compare and increment operation mechanism, original command list, and optimized commands. The up and down side is respectively the COMPARE and INCREMENT part with two examples.

ending block of '1's, termed Last_1s_Sequence. In the INCREMENT part of Figure 4, X is the number will be added with 1. We first use uni-directional propagation to extend the last logic '1' in $\overline{X}$ to MSB, obtaining Tmp1. The ending '0's in Tmp1 is the location of Last_1s_Sequence in X. In second step, we left shift $\overline{Tmp1}$ by one bit (indicated by Shf($\overline{Tmp1}$)), which acts as inverting the Last_0 in X to 1. The third step is to merge the leading unchanged bits by executing an OR operation between Shf($\overline{Tmp1}$) and X, obtaining Tmp3. In the final step, we implement an AND operation between Tmp3 and Tmp1 to turn the Last_1s_Sequence into 0s. However, in the case that LSB of X is 0, the shift operation cannot change the Last_0 to 1, as shown in the first example of INCREMENT. Hence, we perform another OR operation with Y in the third step to help change the LSB to 1.

The increment function above is completed in 6 commands, including 4 regular cycles, 1 shift cycle, and 1 propagation cycle, as shown in the INCREMENT part of Figure 4. The 6-command operation can be further optimized to 4 commands, as discussed next.

3.5 Optimizing Compound Operations

The compound operations can be further optimized in three ways. We already parallelized as many commands as possible in the initial design above. We further reduce the number of commands and trim down the total operation cycles as follows. First, we identify and remove redundant operations. As discussed above, the comparison operation requires 2 PrgR's to compute Tmp1 and Tmp2. However, if Tmp2 is replaced with $\overline{X}$Y, the final comparison result is still the same. As shown in the COMPARE part of Figure 4, the original commands 3-6 is reduced to a-c.

Second, we rearrange the commands sequence to optimize them for our hardware. Here, we first explain how to reuse data in ROC to minimize commands. Referring to Figure 2, if we calculate expression X+Y+Z, only three commands are necessary (copy X to C, copy Y to D, copy Z to D) and the result is in cell C. The intermediate result, X+Y, is naturally calculated and stored in C to facilitate the OR with Z. However, if we calculate expression (X+Y)Z with same number of logic operations, four commands are needed (copy X to C, copy Y to D, copy C to D, copy Z to C). The reason for the additional copy to move intermediate result X+Y from C to D is to calculate the AND in the last command. In INCREMENT, there is a (Shf($\overline{Tmp1}$)+X+Y)Tmp3 operation. To avoid extra move of intermediate result, we transform the expression into $\overline{(Shf(Tmp1)\ \overline{X}\ \overline{Y})}$Tmp3 with all AND operations to maximize data reuse in CU. As shown in the INCREMENT part of Figure 4, the original commands 1-5 is reduced to a-d.

Third, we merge different type of commands to reduce the command number. For example, command a, b in COMPARE can be merged to A. This is because the intermediate result is already in SA after the PrgR operation. The value can be restored to cell C0 directly. For the same reason, command b, c in INCREMENT can be merged to command B, as shown in Figure 4.

After these optimization, the final command sequence of COMPARE now is {1, 2, A, c}, which is of length 4, consisting of 2 regular cycles and 2 propagation cycles. The final command sequence of INCREMENT is {a, B, d, 6}, consisting of 3 regular cycles and 1 propagation cycle.

4 DISCUSSION

4.1 Latency of Different Command

For all the operations discussed above, ROC has 3 types of primitive commands: copy, propagation, and shift. The copy command has about the same latency as a regular command [9]. The shift command, based on the same mechanism (AAP) as copy command, also shares the same latency. The propagation command has an extra latency for transmitting bit '1'. From our Spice model evaluation, the average latency is ~0.72ns per bit, which is slightly higher than [14]. For 8, 16, and 32 bit word, the propagation latency increases by 12%, 23%, and 47% compared to the latency of regular command. Hence, the corresponding cycle can be regard as 1.12×, 1.23×, and 1.47× of a regular cycle. Here, we conservatively used 1.5× for 8 and 16 bit words, and 2× regular cycle for 32 bit words.

4.2 Latency Optimization

ROC is compatible with previous latency optimization methods. For example, to take advantage of the isolation method in [10], we can arrange isolation transistors in folded bitline DRAM as shown in Figure 1(a), to isolate our computing structure from the subarray cells. This can greatly reduce parasitic capacitance on bitlines and reduce the latency of CU. Other optimization strategies such as

trading subarray size for more banks [3, 14], can also be adopted in ROC. Note that our main contribution is to reduce computing cycles, and hence can directly benefit other compatible DRAM latency reduction techniques.

4.3 Efficiency of the Computing Unit

Unlike Ambit, ROC does not require extra empty cells to serve as buffers to assist computing. The executions of all optimized commands are carried inside the CUs. We remark that cells in a CU can also store data when they are not used for computing. To this end, both cells store identical data as they are connected by a diode. Hence, both wordlines are activated when the cells are storing data instead of computing.

5 EVALUATION

We use Design Compiler and H-spice for circuit-level simulation. The parameters are from CACTI-3DD [15] which provides power, delay, area, and cycle time for commodity DRAMs. Then we build an in-house simulator to calculate the latency and throughput of ROC on certain applications. We configure a regular DRAM module with 8 banks. The baselines are Ambit, Dracc, Drisa, and KabyLake CPU [16]. Here, we first discuss the area cost of ROC, then compare basic operation cycles with baselines, and finally show the performance improvement in two applications.

5.1 Area Overhead

ROC has two changes on the microarchitecture of DRAM. The first is the addition of dual CUs, which modifies the cell C and D with 3 transistors (Figure 2). The diode-connected transistor can be placed in the isolation region in substrate-plate DRAM. Its area overhead is thus negligible. The additional MS and MN transistors will occupy two rows, based on estimates from [9, 17]. Therefore, including MC and MD, the dual CU costs roughly 8 DRAM rows. This is comparable to Ambit, which also allocates 8 rows for computing. The second source of cost is the enhanced SA with 2 enable transistors in p-latch pair, which increases 14% area in SA. Including the area overhead of dual CU and taking the area ratio of SA (~ 15%), the overall area overhead of the enhanced ROC is around 3%, which is less than half of area overhead of Dracc with 6 additional transistors per side and 4 extra transistors in SA.

5.2 Basic Operation Cycles

Figure 5 compares the total cycles of basic operations in Ambit, Dracc, Drisa, and ROC. That is, how many regular cycles an operation takes until the result is available in DRAM. As Dracc's logic operation is built on Ambit, they have the same performance on bitwise operation. Drisa executes logic operations via extra latches and logic gates circuit. For Drisa_nor, only NOR gate is attached. For Drisa_adder, four types of logic gates and an adder are attached.

Figure 5(a) shows cycles of 7 kinds of basic logic operations. Cycles of ROC and Ambit depend on the complexity of a logic operation. For example, NOT is the simplest which takes only one cycle, while XOR takes more cycles because it is calculated from combination of NOT, AND and OR. Cycles of Drisa depend on whether it has the type of logic gate for the operation it executes. For Drisa_nor, a NOR just needs two cycles, but other operations need many more cycles. On average, Drisa_add and ROC use lowest number of cycles, 2.3 and 2.4 respectively. But Drisa_add has a full adder so ROC is more area efficient. Its area overhead is only 6% of Drisa_add.

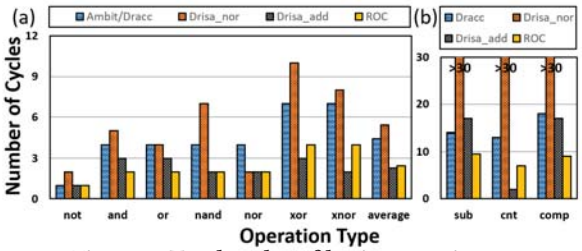

Figure 5: Total cycles of basic operations.

Figure 5(b) shows total cycles for several examples of word-wise operations. Substitution (sub) replaces a specific symbol in database with a new one, which is a common operation in data management. Pop counter (cnt) and comparator (comp) are based on the increment and compare functions discussed in section 3.4 and 3.5. The propagation cycles are transformed to regular cycles as discussed in section 4.1. In substitution and comparator, ROC achieves 32% and 50% improvement over Dracc, 44% and 47% over Drisa_add. Drisa_add only wins in pop counter applications, indicating the logic design in DRAM is quite application-specific. Base on the cycles of substitution, pop counter, and comparator, the raw throughput of ROC is 9.3×, 12.3× and 10.1× respectively above our baselined CPU. Operation cycles of Drisa_nor are much larger (much more than 30), because it only has the shift function to assist executing word-based operation bit by bit. Hence, in the following applications, we do not take Drisa_nor into account.

5.3 Case Study: Hamming Distance

Hamming distance is to calculate the number of different symbols between two strings. It can be used in wide range of applications, such as DNA recognition [18] and hyperdimensional computing [19]. For simplicity, we just calculate Hamming distance between binary vectors, which requires bitwise XOR operation and pop count operation. We perform Hamming distance calculation on the three accelerators, Dracc, Drisa_add, and ROC, with add or increment function.

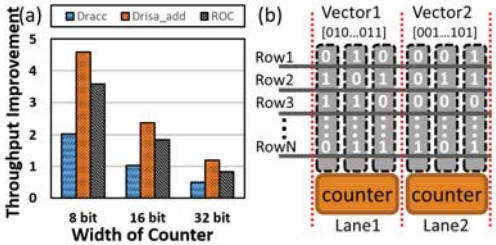

Figure 6: Throughput improvement offered by three DRAM accelerators for Hamming distance over our baseline CPU

In Figure 6(a), the performance improvement of the three accelerators decreases with increasing the bit width of counter, because the counter needs to increment each bit sequentially. As shown in Figure 6(b), the data is organized in vertical direction. Calculating a 16-bit vector with an 8-bit counter requires 2 XOR and 16 increment operations, thereby the throughput reduces with wider words. For 32-bit counter, the speed of Dracc and ROC is even lower than CPU.

In three accelerators, Drisa_add has the best performance, which benefits from the specific adder design but at a cost of 51% area overhead. On the other hand, ROC can achieve a high improvement close to Drisa (3.6× vs 4.6×, 1.8× vs 2.4× and 0.8× vs 1.2×) with only 3% area overhead. For Dracc, it has a larger area but lower throughput than ROC. Therefore, ROC achieves the best speedup over area.

5.4 Cast Study: Table Scan

Table scan is a common operation in a memory-based database management system. It sequentially reads the database and checks the columns for the validity of a predicate of a query. It usually takes many cycles to evaluate simple predicates. For example, a database query Q1 can be written as the following:

$Q1: SELECT\ \ COUNT(*)\ \ FROM\ \ R\ \ WHERE\ \ R.a < C1$

where R.a < C1 is a simple LESS THAN predicate. It involves a significant number of comparison and increment operations. Those operations will be done sequentially with in-memory processing. Creating parallelism among bulk operations will improve the throughput of queries. [20] proposes the BitWeaving method to parallelize comparisons for multiple words. It permutes each word to store it in a memory column. Hence, the same bit in multiple words can be compared in parallel. We found that BitWeaving can achieve better utilization of incrementers/counters in memory arrays and achieve higher throughput of operations. We evaluate the predicates in three ways for Dracc, Drisa_add and ROC, as reported in Figure 7. The first method is BitWeaving with increment performed in CPU ('BitWeaving'). Using CPU instead of in-memory increment/counting is because some designs incur too high latency in counting to be worthwhile. The second method is BitWeaving with in-memory increment/counting ('BitWeaving with inner cnt'). The third method is directly performing word-wise operations without BitWeaving ('Word-wise' Operation).

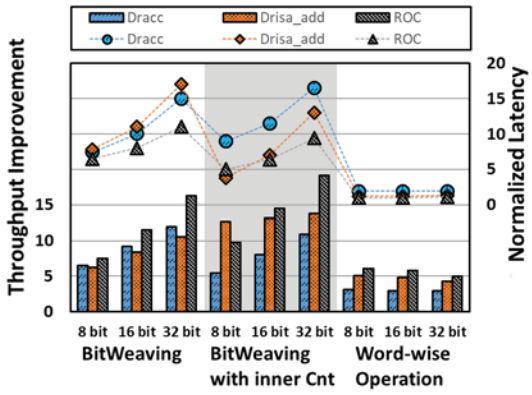

Figure 7: Throughput improvement over our baseline CPU and latency of a single operation in query offered by three DRAM accelerators for table scan

In Figure 7, bars are for throughput over our baseline CPU, and curves are single operation latency of Dracc, Drisa_add, and ROC normalized ROC in Word-wise operation. We draw several conclusions. First, BitWeaving can achieve higher throughput than word-wise operation, as the former can take full utilization of hardware inside CU, especially in incrementers/counters. Second, performing in-memory instead of in-CPU incrementing/counting is beneficial except for Dracc because its long-latency in-memory counters become an overkill. Third, the throughput of BitWeaving increases with word width because the proportion of time spent on sequential increment is reduced. Fourth, ROC outperforms Dracc and Drisa_add in almost all cases because of the low latency designs for every primitive and compound operation. Although Drisa_add also has low-latency operations, as indicated in Figure 5, our optimization on data reuse further greatly reduces latency in each operation while such optimization is hard to implement in Drisa_add.

6 CONCLUSIONS

In this paper, we develop a new PIM architecture to achieve higher performance in basic logic operations and frequent compound operations with lower area overhead than state-of-the-art. We found that it is critical to reduce the number of cycles in each operation in order to achieve high performance. Otherwise, in-memory operations may become even more expensive than in-CPU operations. Moreover, PIM operations should be well designed to maximize parallelism, data reuse, and hardware utilization to achieve operation throughput from the memory. Those will be essential to the overall performance of an application.

7 ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation #1422331, #1535755, #1617071, #1718080, #1725657, The authors thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] O. Villa, *et al.*, "Scaling the Power Wall: A Path to Exascale," in *SC*, 2014
[2] S. McKee, *et al.*, "Reflections on the Memory Wall," in *CF*, 2004
[3] S. Li, *et al.*, "DRISA: A DRAM-based Reconfigurable In-Situ Accelerator," in *MICRO*, 2017
[4] A. Subramaniyan, *et al.*, "Parallel Automata Processor," in *ISCA*, 2017
[5] B. Akin, *et al.*, "Data Reorganization in Memory Using 3D-stacked DRAM," in *ISCA*, 2015.
[6] H. Asghari-Moghaddam, *et al.*, "Chameleon: Versatile and Practical near-DRAM Acceleration Architecture for Large Memory Systems," in *MICRO*, 2016
[7] Y. Kim, *et al.*, "Assessing merged DRAM/logic technology," in *INTEGRATION, the VLSI journal*, 27, 2, 179-194, 1999
[8] V. Seshadri, *et al.*, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013
[9] V. Seshadri, *et al.*, "Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017
[10] D. Lee, *et al.*, "Tiered-latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013
[11] T. Zhang, *et al.*, "Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation," in *ISCA*, 2014
[12] X. Zhang, *et al.*, "Restore truncation for performance improvement in future DRAM systems," in *HPCA*, 2016
[13] P. Nair, *et al.*, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *ISCA*, 2013
[14] Q. Deng, *et al.*, "DrAcc: A DRAM Based Accelerator for Accurate CNN Inference," in *DAC*, 2018
[15] K. Chen, *et al.*, "CACTI-3DD: Architecture-level Modeling for 3D Die-stacked DRAM Main Memory," in *DATE*, 2012
[16] 7th Generation Intel Core Processor Family for S Platforms, Vol. 1, Datasheet. https://www.intel.com/content/www/us/en/processors/core/7th-gen-core-family-desktop-s-processor-lines-datasheet-vol-1.html
[17] S. Lu, *et al.*, "Improving DRAM Latency with Dynamic Asymmetric Subarray," in *MICRO*, 2015
[18] M. Mohammadi-Kambs, *et al.*, "Hamming Distance as a Concept in DNA Molecular Recognition," in *ACS omega*, 2, 4, 1302-1308, 2017
[19] A. Rahimi, *et al.*, "A Robust and Energy-Efficient Classifier Using Brain-Inspired Hyperdimensional Computing," in *ISLPED*, 2016
[20] Y. Li, *et al.*, "BitWeaving: Fast Scans for Main Memory Data Processing," in *SIGMOD*, 2013