Max-PIM: Fast and Efficient Max/Min Searching in DRAM

Fan Zhang
School of Electrical, Computer and Energy Engineering
Arizona State University
Tempe, USA
fzhang95@asu.edu

Shaahin Angizi
School of Electrical, Computer and Energy Engineering
Arizona State University
Tempe, USA
sangizi@asu.edu

Deliang Fan
School of Electrical, Computer and Energy Engineering
Arizona State University
Tempe, USA
dfan@asu.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586096

***Abstract*—Recently, in-DRAM computing is becoming one promising technique to address the notorious 'memory-wall' issue for big data processing. In this work, for the first time, we propose a novel 'Min/Max-in-memory' algorithm based on iterative XNOR bit-wise comparison, which supports parallel in-memory searching for minimum and maximum of bulk data stored in DRAM as unsigned & signed integers, fixed-point and floating numbers. We then develop a new processing-in-DRAM architecture, called *Max-PIM*, that supports complete bit-wise Boolean logic and beyond. Differentiating from prior works, Max-PIM is optimized with one-cycle fast XNOR logic-in-DRAM operation and in-memory data transpose, which are heavily used and keys to accelerate the proposed Min/Max-in-memory algorithm efficiently. Extensive experiments of utilizing Max-PIM in big data sorting and graph processing applications show that it could speed up ~50X and~1000X than GPU and CPU, while only consuming 10% and 1% energy, respectively. Moreover, comparing with recent representative In-DRAM computing platforms, i.e., Ambit [1], DRISA [2], our design could speed up ~3X - 10X.**

***Index Terms*—In-DRAM Computing, IMC, PIM, Min/Max**

I. Introduction

In the era of big data, min/max searching from bulk data array is one of the most important and widely used fundamental operation in many data-intensive applications, including but not limited to sorting, ranking, bioinformatics, data mining, graph processing, route planning, etc. [3]–[5]. For example, online social and news media need real-time ranking to evaluate the hottest information to show on their website, which requires fast min/max searching from massive data. Another example: min/max searching is the most time-consuming computation (over 40%) in many large scale graph processing algorithms, such as popular Dijkstra's algorithm to find the shortest path, Prim's algorithm to find the minimum spanning tree, maximum flow, or to solve the famous traveling salesman problem. In those widely used "best-first algorithm", min/max searching operation is called for every node to find the minimum/maximum value in the large scale graph.

However, implementing fast and efficient min/max searching for big data faces significant challenges in conventional computer system from both memory architecture and computing algorithm. (1) From memory architecture, the well-known 'memory-wall' challenge is causing issues, like long off-chip memory access latency, data congestion due to limited memory bandwidth, two orders higher energy consumption in data movement than data processing, etc. [6]. (2) From computing algorithm, the min/max searching is in general comparison-based algorithm, where CPU needs to compare every element serially for colossal raw data. Such computing property makes it demand ultra-high computing resource and power.

Above challenges naturally motivate researchers to explore implementing fast and efficient min/max searching operations within memory where bulk data is stored, to greatly minimize power-hungry and low speed massive off-chip data communication, aligning with the emerging 'processing-in-memory'(PIM) concept. Among various types of PIM platforms, 'in-DRAM computing' platform is a natural choice for this problem due to its large memory capacity to store bulk data and off-chip data transfer reduction [1], [2], [7], [8]. However, if directly deploying min/max searching in the most popular existing in-DRAM computing platforms, e.g. Ambit [1] or DRISA [2], it faces two challenges. (1) First, unlike general-purpose CPU/GPU providing complex and complete computing instructions, those in-DRAM computing platform only supports bulk bit-wise Boolean logic and very limited instructions, which requires a new design of 'min/max-in-memory' algorithm to make it compatible with and fully leverage the in-DRAM computing hardware supported operations. (2) Second, from logic computing perspective, min/max searching function naturally relies on X(N)OR-based comparison operations. Although existing in-DRAM computing platforms could provide such function, their in-memory-logic designs are mainly depending on charge sharing based majority gate, which requires multiple cycles to implement X(N)OR logic [1], [2]. For example, Ambit [1] requires seven cycles to realize X(N)OR logic. It will take high cost in power and time due to large intermediate data write back, which reduces the benefits of in-memory computing.

To address above two challenges, in this work, we follow a principle of software & hardware co-design to develop a parallel and efficient in-DRAM computing platform, called *Max-PIM*, where the main technical contributions are summarized below:

(1) We first propose a novel *Min/Max-in-memory* searching algorithm based on iterative XNOR bit-wise parallel comparison, which supports in-memory searching for minimum and maximum of bulk data stored in DRAM as unsigned & signed

This work is supported in part by the National Science Foundation under Grant No.2005209, No.2003749

integers, fixed-point and floating numbers.
(2) A new in-DRAM computing circuit and architecture, termed as *Max-PIM*, is then proposed to support complete bulk bit-wise Boolean logic and optimized for our proposed min/max-in-memory algorithm. Differentiating from prior works, the novelty comes from a new micro-architecture enabling an efficient in-memory data-transposing as well as a new in-DRAM-logic circuit to perform computationally-intensive XNOR logic operation in only one cycle.
(3) Detailed cross-layer experiments in a real-world dataset are conducted to demonstrate the efficiency of our design comparing with other state-of-the-art in-DRAM computing platforms (e.g. Ambit [1], DRISA [2]), CPU and GPU.
(4) Sorting and Dijkstra's algorithm in graph processing are used as study cases to demonstrate the improvement of Max-PIM.

II. Preliminary

The Rowclone [9], Ambit [1] and DRISA [2] are among the most recognized in-DRAM computing designs with bulk bit-wise in-DRAM-logic operations. We will discuss their design and compare them with ours in later experiments.

Rowclone [9] presents a high-speed ($< 100ns$) in-memory copy operation within DRAM sub-arrays as opposed to $\sim 1\mu s$ traditional operation in Von-Neumann computing architecture. RowClone-Fast Parallel Mode (FPM) [9] essentially indicates an in-memory copying technique that does not require to transmit data back and forth to the processing units. In this technique, a multi-kilo byte in-memory copy operation can be implemented by issuing two back-to-back `ACTIVATE` commands to the source and destination DRAM rows without `PRECHARGE` command in between [9].

Ambit [1] focuses on performing bit-wise logic operations in DRAM with minor modification. The basic AND, OR functions are implemented through a 3-input charge sharing based majority gates, called Triple Row Activation (TRA) in memory, by simultaneously sending the `ACTIVATE` command to three rows and then a `PRECHARGE` command. NOT function is also implemented considering two rows of Dual-Contact Cells (DCC). Ambit incurs 1% area over commodity DRAM chip [1]. But, it suffers from high-latency multi-cycle logic operations to realize XOR2/XNOR2 on top of TRA.

DRISA [2] is a re-configurable in-DRAM computing platform with different variations. DRISA not only has the ability to perform bit-wise logic operation, but also shows the ability to do arithmetical operations, such as ADD, MUL, and vector-wise inner-product. DRISA-3T1C triples the DRAM cell area to enable functionally complete NOR2 function on read bit-line. However, it also requires multi-cycle operations to implement XOR2/XNOR2 operation. DRISA-1T1C is designed to execute one particular logic function via upgrading the sense amplifier (SA) unit by adding a CMOS logic gate in conjunction with a latch. Therefore, a single function (here, XNOR) could be implemented in two consecutive cycles with add-on CMOS circuitry. So, to support such a great variety of operations, DRISA requires much more modifications on standard DRAM architecture than Ambit.

III. Proposed Max-PIM

A. Min/Max-in-Memory Algorithm

As discussed in [10], [11], finding the minimum/maximum (Min/Max) number equals to finding the last/first ranked number in a given list. To fully leverage the parallel bit-wise in-DRAM computing capability due to parallel sensing of multiple bit-lines [1], [2], [7], our core idea of developing min/max-in-memory algorithm is to convert traditional software-based sequential comparison operations to XNOR based bit-wise parallel comparison for all the data stored in the same memory array. Note that, the algorithm we discuss in this section refers to min/max function within one memory array. A large dataset will be first partitioned into multiple memory arrays to get memory-array level min/max in parallel, then an overall min/max will be identified similarly. Algorithm-1 shows the pseudo code of our proposed bit-wise parallel *Min/Max-in-memory* algorithm. It mainly leverages the property that each bit in the binary format has different significance, and parallel in-DRAM-XNOR logic based comparison operation could gradually exclude smaller (or larger) data from MSB (Most Significant Bit) to LSB (Least Significant Bit). Note that, our algorithm has a constant searching time determined by the bit length of numbers. It is compatible with unsigned and signed integers, fixed-point numbers and floating numbers, which will be discussed in separate sub-sections below.

Algorithm 1: Parallel Bit-wise Min/Max-in-Memory

```
   Input  : Array X has M elements, where each element contains N bits.
   Result: Returning the Min/Max value in given array.
1  Storing the input array into 2D bit-array(NxM size) where each element in X
     occupies one column;
2  Matching_Vector = ones(1,M);
3  if find min then
4  |   Matching_Sign_Bit=1 ; ▷ For signed number, the sign bit
   |     and the rest have different XNOR operands
5  |   Matching_Bit = 0;
6  else
7  |   Matching_Sign_Bit=0;
8  |   Matching_Bit = 1;
9  end
10 if unsigned number then
11 |   Matching_Sign_Bit = Matching_bit; ;   ▷ For unsigned bit, we
   |     do not need to distinguish sign bit and others.
12 end
13 while current_bit_position < N do
   |   ▷ Go through every bit from MSB to LSB.;
14 |   if current_bit_position == 0 then
15 |   |   Matching_Result = XNOR(current_bit,Matching_Sign_Bit);
16 |   else
17 |   |   Matching_Result = XNOR(current_bit,Matching_Bit);
18 |   end
19 |   if Matching_Result == All_Zeros then
20 |   |   Continue;
21 |   else
22 |   |   Matching_Vector = Matching_Result;
23 |   end
24 end
```

1) Unsigned Integer: Fig.1 gives an example to find the minimum value in a 4-bit unsigned *int* array. Since 4 bits are used to represent unsigned value, 4 iterations are required to compute the index of minimum value(s). First, it initializes a *matching vector* with all '1's, whose length equals to the number of data in the array, and this matching vector will be updated with the outputs of parallel XNOR logic in every

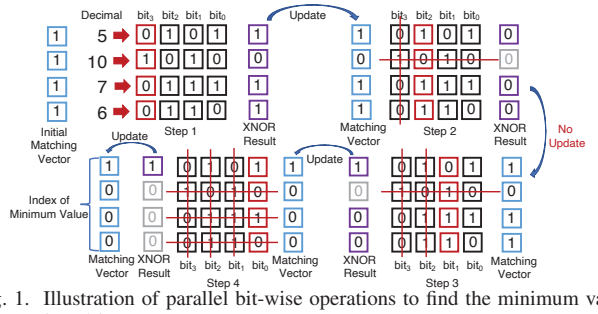

Fig. 1. Illustration of parallel bit-wise operations to find the minimum value of a unsigned integer array.

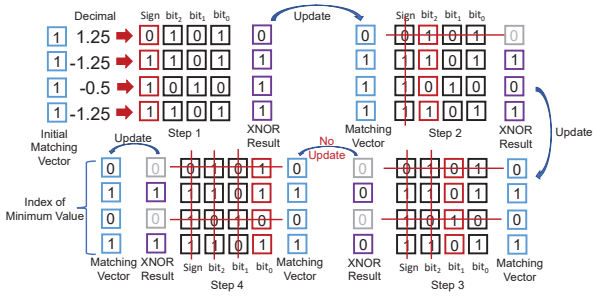

Fig. 2. Finding the minimum value from a signed fixed point number array.

iteration except when the parallel XNOR outputs are all '0's (line 18 to 22 in algorithm-1). Each element in the matching vector corresponds to one number in the array, where '1' indicates this number will be compared in the following iteration, while '0' indicates the corresponding number will be excluded. Starting from MSB, it applies parallel XNOR logic to MSB of every number with constant '0' for min function (XNOR with constant '1' for max function), corresponding to line 3 to 11 in algorithm-1. Next, the matching vector value of '1111' is updated with current iteration parallel XNOR output-'1011', since XNOR outputs are not all '0's (line 18 to 22). As shown in Fig.1, in the second iteration, the new matching vector value-'1011' indicates that the second row (marked as '0') should be excluded from XNOR based comparison operation, and this position will remain as '0' in all following iterations. The next iteration will compute parallel XNOR between the second MSBs of non-excluded numbers and constant '0' (line 16). In this example, in the second iteration, after computing XNOR between the rest bits-'111' with constant '0', the outputs are all '0'. Based on our algorithm-1 line-18, it will skip the update of matching vector of this iteration and continue to next iteration. Similar process will continue based on the same rule until LSB. As our example shown in Fig.1, when the comparison in LSBs is finished, the position(s) remained as '1' in the matching vector is(are) the identified minimum number(s).

Theoretically, we could stop the process when there is only one '1' left in the matching vector (e.g. iteration-3 in our example) to save searching time. However, our target in this work is that all the operations in the algorithm-1 will be implemented by a corresponding circuit module in our proposed Max-PIM platform. Designing such early stop detection will incur extra overhead to the hardware. Thus, we choose to simplify the hardware design with no early stop, leading to a constant searching time determined by the bit representation length.

2) Signed Integer: For signed numbers in the min/max-in-memory algorithm, the basic procedure is the same as the above discussed unsigned numbers except sign bit. In signed number, the first bit represents the sign, where '1' and '0' indicate negative and positive, respectively. When for min (or max) function, the algorithm will first check the sign bit. If negative (or positive) values are found, it will exclude all positive (or negative) numbers for the following computation. To do so, as shown in line-3 to -8 in algorithm-1, a parallel XNOR between sign bits with constant-'0' (or '1') for max (or min) function is needed. The rest procedure will be similar.

3) Fixed-Point Number: Fig. 2 provides an example to implement min-in-memory function for a group of four 4-bit fix numbers. To start, matching vector will be initialized with all '1's. Then, XNOR logic is applied to all sign bits with '1', where the matching vectors will be updated based on XNOR outputs to exclude any positive number for next iterations. The rest 3 iterations are similar to the above discussed unsigned integer number process. In this specific example, it can be seen that if there are multiple minimum values in the array, our final matching vector will also indicate with multiple '1's.

4) Floating Number: Our algorithm is also compatible with floating numbers. For example, the IEEE754 [12] floating number representation is a popular floating-point arithmetic standard. It contains three parts: signed bit, exponent bits, and significand precision. According to the IEEE754, the binary floating number may take 16 bits to 256 bits to represent one floating number. Taking the 64 bit floating number as an example, the representation $(-1)^{sign} \times (1+\sum_{i=1}^{52} b_{52-i}2^{-i}) \times 2^{e-1023}$ can be written as $(-1)^{sign} \times 2^{e-1023} \times 1.fraction$. That means the bits in the exponent parts have higher significance than the bits in the fraction parts (significand precision). And the bits within exponent parts and significand precision obey the same rule as unsigned integer number, where the bit significance decreases along bit position. Thanks to the bit organization that exponent is on the left side of the fraction part in the endianness(little-endian) sequencing, our proposed Min/Max searching function can be directly applied to the floating number without modification.

B. Max-PIM Logic Circuit and Architecture

It can be summarized that several important functions are required to be implemented efficiently for in-DRAM computing platform to fully support above algorithm: 1) transpose data copy; 2) fast and parallel in-DRAM XNOR logic; 3) matching vector update; 4) identified min/max index decode.

The overall architecture diagram of our proposed Max-PIM is given in Fig. 3. It keeps the original memory hierarchy by dividing every DRAM chip into multiple banks that share I/O and buffers. Each bank contains multiple normal memory matrices (MATs) consisting of typical DRAM sub-arrays and a single computational array shown in blue. The computational array is developed similar as Ambit [1] [13], with all Ambit supported logic and instructions, but enhanced to support a new Dual Row Activation (DRA) mechanism to

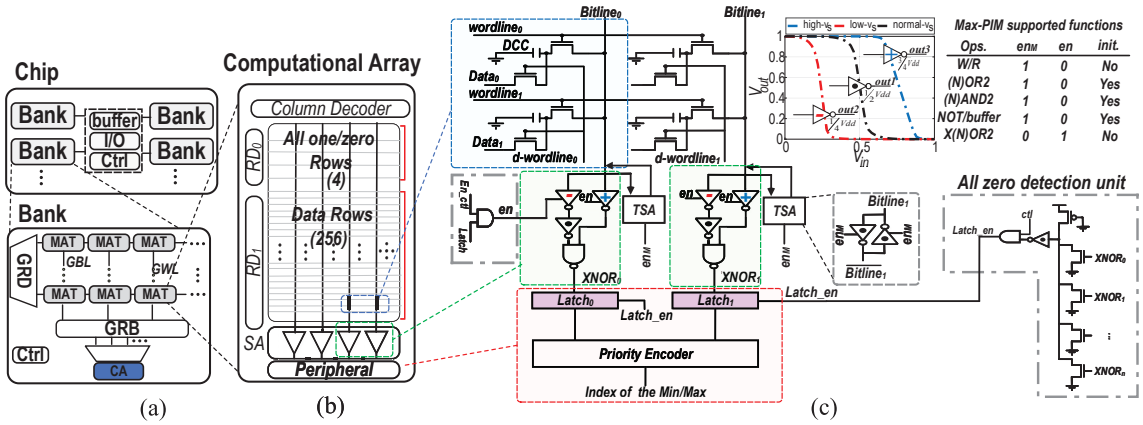

Fig. 3. (a) The architecture diagram of our proposed hardware design, (b) The computational sub-array with components, (c) The proposed dual-row activation based XNOR logic and peripheral circuits for other operations.

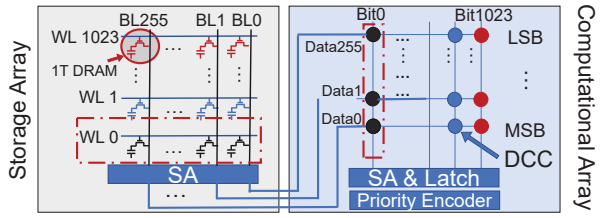

Fig. 4. Single-cycle transpose copy operation.

perform XNOR operation in addition to the typical Triple Row Activation (TRA). Each computational array consists of: two row decoders (one for data rows and one for all 1/0 rows), one column decoder (to enable transpose copy), modified logic sense-amplifier (to support XNOR logic) with Typical Sense Amplifier (TSA, for memory sensing and charge sharing based majority gate), one latch per bit-line (to store the matching vector and control the logic-SA), pseudo-OR gate (to detect all-zero XNOR outputs), one priority encoder (to return final index of identified min/max locations).

1) Transpose Copy: Due to DRAM's destructive read property, any in-DRAM computing needs to make a copy of operand data to perform in-DRAM-logic [1], [2], [7]. More importantly, based on our min/max-in-memory algorithm, one important operation of each iteration is that we need to implement parallel XNOR logic between the bits in the same location of all numbers with constant-'1'/'0', which brings two requirements for in-DRAM computing hardware: 1) for single XNOR logic, both operands (i.e. one specific bit from one number and constant-'1'/'0') need to be stored in the same bit-line based on the circuit design of in-DRAM-logic that will be explained in next paragraph; 2) to fully leverage the parallelism of in-DRAM computing, similar as prior works, multiple logic-SAs could be simultaneously activated, which is determined by the size of memory array. Therefore, to conclude, in the computational memory array, the binary data of a number needs to be stored along the bit-line direction, not along the traditional word-line direction, which requires a *transpose copy* from data memory to computational memory.

To support the transpose copy operation, we adopt the similar dual-contact cell (DCC) design (Fig.3) as [1]. In DCC, one transistor shares the gate with other transistors on the same row to form the normal row-wise word-line. Another transistor shares the gate with other transistors on the same column to form the column-wise write word-line, which is controlled by the column decoder. The drain and source of the second transistor connect to the cell capacitor and the other normal DRAM array, respectively (the Data port in Fig. 3(c)). Copying operand data from data storage array to computational array requires two steps: (1) reading entire word-line from the data storage array and (2) writing them to one bit-line (column) of the computational array with the help of DCC and column decoder. Fig. 4 gives a demonstration for this transpose copy.

2) Parallel XNOR logic in one cycle: Max-PIM supports all traditional charge sharing based majority gate logic and optimize heavily used XNOR logic with only one cycle. In our min/max-in-memory algorithm, since one operand of XNOR comes from one bit in specific location of one number in the array, and the other one is constant-'1' or '0' for min or max function, respectively, we reserve several rows in the computational array for storing such constant bits. Traditionally, due to the destructive read in DRAM, we have to wait for data refresh of all ones/zeros rows after every computation. Thanks to the DCC cell design, which separates write and read, we could only use 2 rows for all '1's and 2 rows for all '0's without introducing extra refresh time by refreshing one row while using the other row for XNOR logic. For the all '1/0' rows, we connect the data port of DCC cells to VDD/GND.

Regarding in-DRAM XNOR logic circuit, it is based on the popular Dual-Row Activation scheme [7]. First, the bit-lines will be pre-charged to $\frac{VDD}{2}$. Then, two word-lines storing the operand data are activated simultaneously, which leads to charge sharing between the two cells in the same bit-line and generates different sensing voltage at the associated logic-sense-amplifier (logic-SA) circuit depending on different values stored in these two cells. Then this sensed voltage will be compared with different reference voltages to implement different logic functions. In this work, we present a logic-SA circuit design that could implement XNOR logic in only one cycle as shown in Fig.3(c), where typically multiple cycles are needed in most prior works [1], [2]. In the circuit, the

plus sign marked inverter has higher threshold voltage, and the minus sign marked inverter has lower threshold voltage than normal inverter. It is designed as, for the high threshold inverter, the output goes to zero only when both activated cells have high voltage (storing '11'), implementing NAND function. The low threshold inverter works on the opposite situation that it only generates a high output when both cells have low voltage (storing '00'), implementing NOR function. Therefore, based on XNOR = NAND(OR, NAND), the XNOR logic is implemented by adding an inverter and NAND logic circuit following the two skewed inverter as shown in Fig.3(c).

3) Matching Vector Update: As described in our algorithm-1, matching vector needs to be updated in every searching iteration, except all-zero XNOR outputs. To avoid writing back the XNOR outputs to memory for matching vector update, we incorporate latches in the logic-SA circuits as shown in Fig.3(c) to store and update matching vector. Moreover, to exclude update in case of all-zero XNOR outputs, one all-zero detection unit circuit is designed based on a pseudo OR gate (dotted box of Fig.3(c)) to control the matching vector latch update.

4) Identified Min/Max decode: When the iterative XNOR comparison is done, the final matching vector indicates the indices/ location(s) of found min/max values. Max-PIM is able to return the identified min/max data address to user through a priority encoder.

IV. Experiments and Results

A. Experiment Setup

In our experiments, we set each bank has 16 MATs and each MAT consists of 2 sub-arrays with size of 1024×256. Each bank contains one computational array shared by all the sub-arrays within the same bank. The computational array size is 260×1024, where 256 rows are used to store data and 4 reserved rows store constant '0/1'. Thus, one bank stores ∼1MB data. Considering 1GB DRAM capacity, there are 1024 banks for the whole DRAM. As the competitors, we built Ambit and DRISA platforms with the same organization as Max-PIM, but with their own in-DRAM logic circuits. For a fair comparison, all designs use DCC cells for entire computational array to support transpose write.

To evaluate the performance, we use the similar cross-layer evaluation framework in [7], starting from circuit-level evaluation with TSMC 65nm technology. We use the same process node to re-implement all counterpart designs. The memory peripherals are simulated using the same technology library in Synopsys Design Compiler. We then feed the circuit simulation results into architecture-level tools. We then extensively modify CACTI7.0 [14] to extract the performance results. An in-house mapping algorithm is then developed on top of architecture level to parse the input vector coming from various data-sets and evaluate the performance.

B. Comparing with other in-DRAM computing platforms

We compare Max-PIM with two start-of-the-art in-DRAM computing platforms: Ambit [1] and DRISA [2]. To conduct

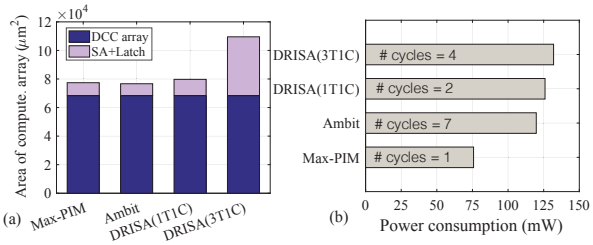

Fig. 5. (a) Area overhead. (b) Power consumption of a single computational array.

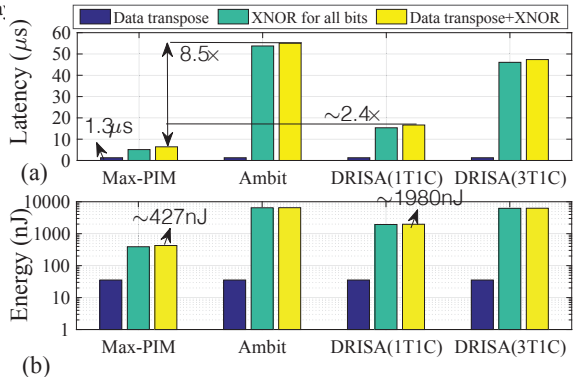

Fig. 6. (a) With the DCC and transpose write, data mapping is faster than the computation. Max-PIM is much faster than other counterparts in computing due to one-cycle XNOR design. (b) Max-PIM shows best energy efficiency as well in min/max searching.

fair comparison in min/max searching, we re-implement those designs with all required peripherals, including DCC array, column decoder, multiple row decoder, pseudo OR gate, latch (Matching vector), and priority encoder.

Fig. 5(a) shows the area overhead of a single computational array. All the peripheral circuits required to support min/max-in-memory algorithm is the same. The area difference comes from different logic-SA circuit designs. It shows Max-PIM has similar area overhead as Ambit and DRISA(1T1C) design, but DRISA(3T1C) design has much larger area overhead. Fig.5(b) provides the power comparison and cycles for XNOR logic. The DRISA(1T1C) uses XNOR gate at the bottom of every bit-line instead of leveraging the DRA to perform the XNOR, requiring two cycles. Both Ambit and DRISA(3T1C) leverage the multi-row activation to achieve the logic operations, such as AND, OR, and NOT. Ambit needs up to 7 cycles for XNOR. DRISA(3T1C) needs to do NOR 4 times and stores back the intermediate data. The Max-PIM could implement XNOR logic in only one cycle. It avoids intermediate data write back compared with other designs, and thus greatly reducing power and latency. Fig. 6 summarizes the distribution and comparison of latency and energy for Min/Max searching within one computational sub-array.

C. Min/Max searching in real word dataset

To evaluate the performance of our Max-PIM platform in min/max searching in a real world dataset, we adopt a popular data-set T10I4D100K [15] containing 1010228 numbers, where each number is represented as 256-bit unsigned integer numbers stored in DRAM. We report the execution time and energy consumption measurement in Fig.7 for different computing platforms including Ambit, DRISA, CPU, GPU. For

all in-DRAM computing platforms, including our Max-PIM, Ambit and DRISA, such large array cannot fit into a single computational array, requiring necessary data partitioning. The total 1010228 numbers are partitioned into 987 computational sub-arrays, where each sub-array will compute in parallel to return a local minimal number. Then, such local minimal numbers will be wrote into another computational array to get the global minimal number. For CPU&GPU evaluation, the reported time is total execution time, including data loading from main memory and processing. For energy estimation, similar as DRISA [2], we scale down 50% of CPU&GPU average power to exclude the power cost by cooling, voltage regulators, etc. Among in-DRAM computing platforms, our Max-PIM achieves the minimal latency and energy consumption. Max-PIM could achieve ∼50 -∼1000× speed improvement, and at least one order smaller energy consumption than GPU&CPU.

D. Applications in Sorting and Graph Processing

In this section, we utilize our Max-PIM in real-word applications of data sorting and Dijkstra's algorithm in graph processing to further evaluate the performance.

For the sorting evaluation, the dataset is the same as the one used in section IV-C. Leveraging the parallel min/max searching provided by Max-PIM, we adopt the min/max sorting algorithm, where each round will find the min or max to iteratively sort the data. We report the performance of different in-DRAM computing platforms and software implementation in CPU in table I. It can be seen that all in-DRAM computing platforms outperform software implementation in CPU by three orders mainly due to save of large amount of off-chip data movement and ultra-parallel processing capability. Aligning with prior experiments in single min/max searching, our Max-PIM achieves the best performance compared with other in-DRAM computing platforms due to its optimized XNOR and peripheral circuits.

Dijkstra's algorithm [16] is a popular and widely used algorithm in graph processing to find the shortest path in large graph, where min/max searching dominates overall computation. Three different dataset are used here: geom has 7343 nodes, foldoc has 13356 nodes and EAT_SR has 23219 nodes [17]. Table II reports the Min/Max searching speed improvement over CPU for different in-DRAM computing platforms. Similar performance improvement trend could be observed. In general, in-DRAM computing platform could all achieve one to two orders speed up than CPU and our Max-PIM still outperforms all other in-DRAM computing platforms significantly. Meanwhile, it is also observed that larger speed up is achieved for larger dataset, clearly proving again memory-wall becomes the bottleneck when dealing with larger dataset.

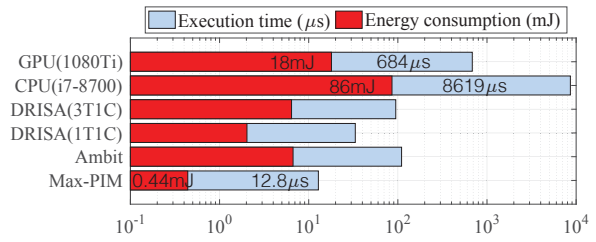

Fig. 7. Min/Max searching performance of real dataset

TABLE I
SORTING PERFORMANCE

Name	Max-PIM	Ambit	DRISA (1T1C)	DRISA (3T1C)	CPU
Time(sec)	6.49	55.6	16.83	47.86	9888.05

TABLE II
MIN/MAX SEARCHING SPEEDUP OVER CPU IN DIJKSTRA'S ALGORITHM

Name	geom(7343 nodes)	foldoc(13356)	EAT_SR(23219)
Max-PIM	91X	167.74X	466.29X
Ambit	9.78X	18.68X	53.49X
DRISA(1T1C)	34.62X	64.03X	179.07X
DRISA(3T1C)	11.53X	21.84X	62.32X

V. CONCLUSION

In this work, we first propose a min/max-in-memory algorithm to find the minimum/maximum of an array stored in main memory. A DRAM based in-memory computing design, named Max-PIM, is presented to support this algorithm with massive parallelism and minimized data movement. Comparing to other DRAM based in-memory computing designs, CPU, and GPU platforms, Max-PIM's software&hardware co-design requires the shortest time and smallest energy for an identical task. We have demonstrated that such proposed Min/Max searching algorithm and hardware have great potential to be applied in many applications, such as sorting, ranking, graph processing, etc.

REFERENCES

[1] V. Seshadri *et al.*, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 MICRO*, 2017, pp. 273–287.

[2] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 MICRO*, 2017, pp. 288–301.

[3] R. V. Lebo, "Chromosome sorting and dna sequence localization," *Cytometry: The Journal of the International Society for Analytical Cytology*, vol. 3, pp. 145–154, 1982.

[4] M. A. Ismail *et al.*, "Multidimensional data clustering utilizing hybrid search strategies," *Pattern recognition*, vol. 22, pp. 75–89, 1989.

[5] L. Page *et al.*, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[6] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," *SIGPLAN Not.*, vol. 53, no. 2, p. 316–331, Mar. 2018.

[7] S. Angizi *et al.*, "Redram: A reconfigurable processing-in-dram platform for accelerating bulk bit-wise operations," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[8] Q. Deng *et al.*, "Dracc: a dram based accelerator for accurate cnn inference," in *2018 DAC*, 2018, pp. 1–6.

[9] V. Seshadri *et al.*, "Rowclone: fast and energy-efficient in-dram bulk data copy and initialization," in *Micro*, 2013, pp. 185–197.

[10] P.-E. Danielsson, "Getting the median faster," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 71 – 78, 1981.

[11] M. Vacca *et al.*, "Logic-in-memory architecture for min/max search," in *2018 ICECS*, 2018, pp. 853–856.

[12] Wikipedia contributors, "Ieee 754 — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=IEEE_754&oldid=976446634, 2020, [Online; accessed 5-September-2020].

[13] S. Angizi and D. Fan, "Accelerating bulk bit-wise X(N)OR operation in processing-in-dram platform," *CoRR*, vol. abs/1904.05782, 2019.

[14] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, Jun. 2017.

[15] *T10I4D100K dataset*. [Online]. Available: http://fimi.cs.helsinki.fi

[16] D. B. Johnson, "A note on dijkstra's shortest path algorithm," *J. ACM*, vol. 20, no. 3, p. 385–388, Jul. 1973.

[17] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.