FlexiDRAM: A Flexible in-DRAM Framework to Enable Parallel General-Purpose Computation

Ranyang Zhou[†], Arman Roohi[‡], Durga Misra[†] and Shaahin Angizi[†]
[†]Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA
[‡]School of Computing, University of Nebraska–Lincoln, Lincoln, NE, USA
aroohi@unl.edu,shaahin.angizi@njit.edu

ABSTRACT

In this paper, we propose a Flexible processing-in-DRAM framework named FlexiDRAM that supports the efficient implementation of complex bulk bitwise operations. This framework is developed on top of a new reconfigurable in-DRAM accelerator that leverages the analog operation of DRAM sub-arrays and elevates it to implement XOR2-MAJ3 operations between operands stored in the same bit-line. FlexiDRAM first generates an efficient XOR-MAJ representation of the desired logic and then appropriately allocates DRAM rows to the operands to execute any in-DRAM computation. We develop ISA and software support required to compute in-DRAM operation. FlexiDRAM transforms current memory architecture to a massively parallel computational unit and can be leveraged to significantly reduce the latency and energy consumption of complex workloads. Our extensive circuit-to-architecture simulation results show that averaged across two well-known deep learning workloads, FlexiDRAM achieves ∼15× energy-saving and 13× speedup over the GPU outperforming recent processing-in-DRAM platforms.

CCS CONCEPTS

• **Hardware** → **Dynamic memory**; • **Computer systems organization** → *Reconfigurable computing*.

ACM Reference Format:
Ranyang Zhou[†], Arman Roohi[‡], Durga Misra[†] and Shaahin Angizi[†]. 2022. FlexiDRAM: A Flexible in-DRAM Framework to Enable Parallel General-Purpose Computation. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '22), August 1–3, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3531437.3539721

1 INTRODUCTION

Take the sight of the modern applications, the geometric growth of data imposes high energy consumption and high latency in the traditional von-Neumann computer architecture constrained by transferring a vast amount of data between separate memory and processing blocks [18]. To overcome these issues, Processing-in-Memory (PIM) mechanism has been widely exploited [2, 4, 12] to incorporate logic units within memory to process data internally without high-frequency access and long-distance transmission. PIM in the context of main memory (DRAM- [4, 12, 18]) has drawn much more attention in recent years mainly due to larger memory capacities and off-chip data transfer. Such processing-in-DRAM platforms show significantly high throughputs leveraging multi-row activation methods to perform bulk bit-wise operations by either modifying the DRAM cell and/or sense amplifier.

Ambit [18] presents a Triple-Row Activation (TRA) technique to carry out a majority gate (MAJ)-based AND/OR logic with negligible modification to SA, excelling NVIDIA GeForce GPU, and even HMC [16], respectively by 32.0×, and 2.4×. However, Ambit's throughput is limited when it comes to complex XOR-based logic implementations. For accelerating Convolutional Neural Networks (CNN), DRISA [12] presents two alternative 3T1C- and 1T1C-based PIM techniques and improves speedup and energy-efficiency by 7.7× and 15× against GPUs. The 3T1C-based computing performs intrinsic in-memory NOR logic by adding two more transistors to every cell. The 1T1C-based design does not modify the DRAM cell structure; however, it performs multi-cycle AND/OR logic by adding extra logic circuitry and latch after the sense amplifier. Some prior works have proposed PIM designs with five-row activation mechanisms that support more complex operations [2, 4] such as addition. Alternatively, pLUTo [9] presents DRAM lookup table-based operations supporting simultaneous querying and LUT operations. While there are many proposals to design processing-in-DRAM platforms, two major shortcomings avoid their further applicability. (1) They do not offer logic flexibility at the circuit-level and only support basic Boolean operations. This limits their performance for many applications requiring more complex logic to potentially benefit from PIM. (2) Most platforms do not provide ISA and software support to process the user-defined operations. Inspired by Ambit, a framework, SIMDRAM [10], has been designed to address the second challenge by providing the programming interface and parallel SIMD substrate that converts the operation to MAJ3-NOT trees. However, the framework's performance is limited by the Ambit's intrinsic constraints in performing logic operations.

In this work, we propose FlexiDRAM as a Flexible processing-in-DRAM framework for general-purpose computation to potentially address the aforementioned challenges. Our main contributions are as follows. (1) We design a high-throughput and energy-efficient XOR-MAJ Graph (XMG)-friendly processing-in-DRAM architecture based on a set of novel microarchitectural and circuit-level schemes to realize a data-parallel computational unit for various applications; (2) We exploit and modify effective methods to (i) generate an efficient XMG-based representation of the desired logic and (ii) appropriately allocate DRAM rows to the operands of the operation to execute any complex computation; (3) We develop ISA, software

ISLPED '22, August 1–3, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9354-6/22/08…$15.00
https://doi.org/10.1145/3531437.3539721

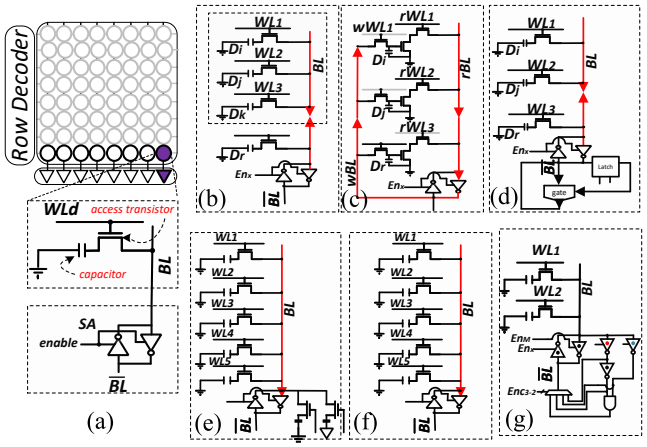

Figure 1: (a) DRAM sub-array organization, (b) Ambit's TRA [18], (c) DRISA's 3T1C [12], (d) DRISA's 1T1C [12], (e) GraphiDe's QRA [4], (f) QRA [2], (g) ReDRAM's DRA [5].

support, and the interface required to compute any user-defined in-DRAM operation; (4) We evaluate the performance of FlexiDRAM on data-intensive deep learning applications and compare our work with the GPU and the state-of-the-art in-DRAM computing designs.

2 BACKGROUND

At the architecture-level, the DRAM memory chip is spilt into several memory banks. Each bank comprises 2D sub-arrays of memory bit-cells that are virtually ordered in memory matrices (mats). As depicted in Fig. 1(a), the memory sub-array consists of (i) memory rows (normally 2^9 or 2^{10}) connected to DRAM cells, (ii) a Sense Amplifier (SA) row, and (iii) a memory row decoder connected to rows. A DRAM cell is composed of two modules, a storage module (capacitor) and an access module (Access Transistor-AT), as shown in Fig. 1(a). The gate and drain of DRAM's AT are connected to the Word-line (WL) and Bit-line (BL), respectively. DRAM cell stores the binary data by the charge of the capacitor. It encodes a fully-charged (V_{dd}) capacitor as logic '1' and no-charge capacitor as logic '0'. To realize DRAM read and write, both BL and $\overline{\text{BL}}$ are initially pulled to $\frac{V_{dd}}{2}$. Accessing data from a DRAM's sub-array after the initial state is accomplished with three commands [18] issued by the memory controller: 1) With the `ACTIVATE` command, a target row is activated, and row data is transferred to the SA row. The cell shares its charge value ($0/V_{dd}$) with the BL which slightly changes the initial BL 's voltage ($\frac{V_{dd}}{2} \pm \delta$). Then, the memory controller activates the *enable* signal that makes the SA amplify the δ towards the original value of the data through voltage amplification leveraging the switching threshold of SA's inverter [18]. 2) By a `WRITE/READ` command, the data can be then moved to/from SA from/to DRAM bus. 3) With a `PRECHARGE` command, both BL and $\overline{\text{BL}}$ precharge to the initial state.

RowClone-Fast Parallel Mode (FPM) [17] presents a very fast in-memory copy operation (<100ns) by issuing two back-to-back `ACTIVATE` commands (without `PRECHARGE` command in between) to the source and destination rows. Ambit [18] extends the RowClone idea to realize three-input majority gate-based operations (MAJ3) in DRAM sub-arrays through simultaneously issuing the `ACTIVATE` command to three rows with a `PRECHARGE` command afterwards (Fig. 1(b)). With one row as the control (D_k) row, initialized by '0'/'1', this method implements in-memory AND2/OR2 based on TRA mechanism via charge sharing among connected DRAM cells (D_k, D_i and D_j) and writes the result back on D_r cell. Moreover, Ambit leverages dual-contact cells to execute in-memory NOT operation and complementary operations. The DRISA-3T1C [12] (Fig. 1(c)) leverages the 3-transistor DRAM design [19]. Such cell consists of two separated write/read ATs, and one additional transistor for decoupling the capacitor from the read BL (rBL) that links the two input DRAM cells in a NOR style on the rBL to perform the Boolean-complete NOR2 function. DRISA-1T1C (Fig. 1(d)) incurs a large area overhead to perform in-memory operations via an upgraded SA consisting of a CMOS logic gate and a latch in multiple cycles. GraphiDe [4] and the design in [2] (Fig. 1(e-f)) extend Ambit's idea to realize a Quintuple-Row Activation (QRA) mechanism and leverage it along with TRA to accelerate addition-based applications. However, activating more than three DRAM rows adversely impacts the reliability of the operation [18]. ReDRAM [5] (Fig. 1(g)) explores a Dual-Row Activation (DRA) mechanism to implement a set of functions by adding ~14% overhead to DRAM chip area.

3 FLEXIDRAM

3.1 Circuit-level Exploration

We propose FlexiDRAM to realize the TRA mechanism with a new reconfigurable SA, as shown in Fig. 2(a), on top of existing DRAM circuitry. It consists of a regular inverter-based DRAM SA equipped with add-on circuits, including two inverters, a pair of NMOS and PMOS transistors, and one 2:1 MUX, controlled with four enable signals ($C_{nor}, C_{m/min}, C_{nand}, C_{mux}$). This design leverages the charge-sharing feature of DRAM cell and elevates it to implement `MAJ3` and `XOR2` logic between selected rows through static capacitive-NAND/NOR functions in a single cycle. To implement capacitor-based logics, we use two different inverters with shifted Voltage Transfer Characteristics (VTC), as shown in Fig. 2(b). In this way, a NAND/NOR logic can be readily carried out based on

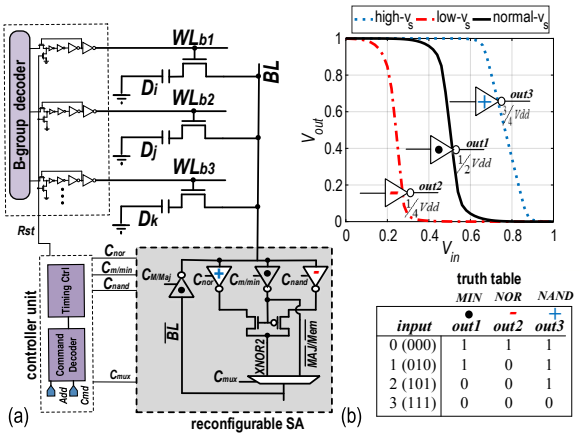

Figure 2: (a) The FlexiDRAM's sense amplifier supporting in-memory XOR and MAJ, (b) VTC of skewed inverters.

Table 1: FlexiDRAM's configuration bits.

Ops.	Activation	$C_{m/min}$	C_{nor}	C_{nand}	C_{mux}
Read	SRA*	1	0	0	0
MAJ3	TRA	1	0	0	0
XOR2	TRA	1	1	1	1

*Single Row Activation

high switching voltage (V_s)/low-V_s inverters with standard high-V_{th}/low-V_{th} NMOS and low-V_{th}/high-V_{th} PMOS transistors. It is worth mentioning that utilizing low/high-threshold voltage transistors along with normal-threshold transistors has been accomplished in the low-power application, and many circuits have enjoyed this technique in low-power design [3, 11, 14, 15]. Besides, the Minority3 function can be accomplished with a standard inverter based on the TRA mechanism. By slightly modifying the memory row decoder and activating three rows at the same time (WL_{b1}, WL_{b2}, and WL_{b3} in Fig. 2(a)), NOR3, Minority3, and NAND3 functions can be then realized through charge sharing between BL and cells' data. We then reformulated the Boolean function of XNOR3 to implement it in a single cycle. We observed that when the minority function of three inputs is '1', XNOR3 can be implemented by the NOR3 function, and when minority function is '0', XNOR3 can be achieved through the NAND3 function. This can be implemented by a multiplexer circuit as shown in Fig. 2(a). In this way, MAJ3 and XOR are readily computed after write-back inverter, and the result will drive the BL based on the configuration bits in Table 1.

$$XNOR3 = MIN(D_i, D_j, D_k).NOR(D_i, D_j, D_k)+ MAJ(D_i, D_j, D_k).NAND(D_i, D_j, D_k) \quad (1)$$

3.2 Architecture-level Exploration

FlexiDRAM framework is developed on top of the proposed XOR-MAJ-friendly processing-in-DRAM platform. It offers flexibility in bulk bit-wise operations in DRAM and flexibly supports user-defined operations by getting exposed to programmers and system-level libraries.

3.2.1 Sub-array Organization. Figure 3 gives an overview of the sub-array organization. FlexiDRAM divides DRAM's row space into three sub-groups: (1) The D-group containing 1018 original data rows, (2) The C-group (or C-row) initialized by all-'0' values, and (3) B-group containing five rows as the computation zone. Addresses of the D-group and C-group are decoded via a regular row decoder and

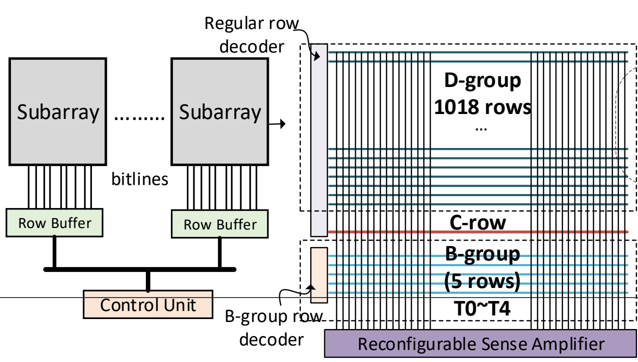

Figure 3: FlexiDRAM sub-array organization.

Table 2: B-group address mapping to corresponding WL(s).

μReg.	WL(s)	μReg.	WL(s)	μReg.	WL(s)
B0	T0 addr	B5	T0,T1,T2 addr	B10	T0,T2,T4 addr
B1	T1 addr	B6	T0,T1,T3 addr	B11	T0,T3,T4 addr
B2	T2 addr	B7	T0,T2,T3 addr	B12	T1,T2,T4 addr
B3	T3 addr	B8	T1,T2,T3 addr	B13	T1,T3,T4 addr
B4	T4 addr	B9	T0,T1,T4 addr	B14	T2,T3,T4 addr

activated once at a time. Both C-Group and D-Group are dedicated to providing operands for subsequent operations in Group B to avoid data-overwritten. We reserve 15 registers as listed in Table 2 to store addresses used for computation in B-group, i.e., B0 - B14. These registers are split into three parts for the supported functions: (1) B0-B4 are dedicated single rows to data computation, (2) B5-B8 are rows for MAJ implementation, and (3) B9-B14 are rows for both MAJ and XOR implementations. Here, T4 is the specific row storing the value copying from C-row to realize the XOR2 function. This is because, in practice, FlexiDRAM activates three rows to generate XOR logic. We slightly modified the B-group's row decoder such that every input address can activate up to three output addresses, simultaneously.

3.2.2 XMG Implementation. FlexiDRAM implements bulk bit-wise operations based on XMG that requires fewer cycles to perform a given operation compared to state-of-the-art logic graphs. The framework then leverages ACTIVATE-PRECHARGE command sequence (a.k.a., AP primitives) to translate the XMG at install time to the μProgram and hardware instructions. The XMG implementation process is divided into two steps. In the *first step*, FlexiDRAM framework rewrites the original input circuit to the And-Or-Inverter Graph (AOIG) representation as shown in Fig. 4. AOIG represents the logic circuit with a tree graph, where each node represents an AND/OR gate, and the tree branch represents the dependency of the nodes. Each incoming node can be either regular or complemented, depending on whether it is connected to an inverter.

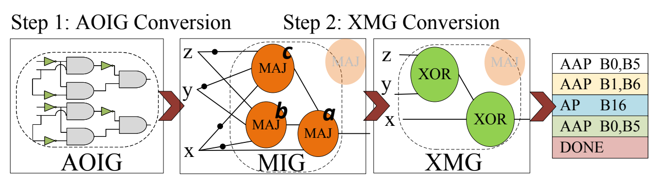

Figure 4: Overview of XMG implementation.

The *second step* takes AOIG as input and drives its optimized Majority-Inverter Graph (MIG) to minimize the complexity. Since the depth of the given MIG circuit naively represents the PIM computation cycles, we first apply logic optimization to minimize the number of logic primitives. The rewriting algorithm consists of two steps: (1) Replacing all the nodes with 3-input MAJs to optimize the graph, and (2) Using a greedy algorithm for multiple transformations and optimizing the graph by transformation rules. In this way, the AOIG turns into MIG with minimum MAJ primitives. In MIG, although the circuit depth of the meta-logic circuit is reduced [10, 20], FlexiDRAM can further optimize it by transforming MIG into XMG representation. MAJ distributes over XOR, which means we can combine MAJ and XOR operations, denoted as XOR-MAJ, and use XOR's Boolean algebra to identify a new distribution rule

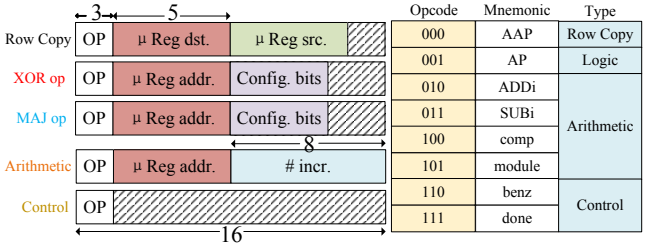

Figure 5: FlexiDRAM's μOps and their description.

[13]. In traditional synthesis, the XOR gate is not the best choice; however, in our design, it can be efficiently implemented in-memory by activating two operand rows and one row of constant '0' simultaneously. Since FlexiDRAM operates based on TRA, but the XOR has only two inputs, one input is set by a constant '0' during the synthesis. In addition, the use of XOR will effectively reduce the depth of the inverter. To detect XOR operations in MIG, we followed the following two principles: (i) if one node a is shared by two lower-level nodes b and c, as shown in step 2 in Fig. 4, x feeds the nodes b and c with the same polarity, while complimentary polarity feeds the node a, (ii) y and z feed the nodes b and c with complemented polarities. With this rule, FlexiDRAM can efficiently replace the MAJ with an XOR structure and further reduce the depth.

3.2.3 μProgram Design. Every FlexiDRAM instruction is composed of a series of `AAPs/APs`, stored in memory. As shown in Fig. 4, after optimizing the XMG, FlexiDRAM uses it to generate the relevant μprograms. We considered two main aspects in generating a program: (i) rows allocation to the data to be calculated next and (ii) collecting the address of the row to be activated and allocating the corresponding register. The μprograms include a sequence of μOperations (μOps) that need to be decoded and executed in the memory control unit to sequence instructions. Figure 5 shows FlexiDRAM's μOp groups and instruction types. The length of each μOp is 16 bits with different organizations. FlexiDRAM supports four basic μOp types, (1) Row Copy based on RowClone [17], (2) Logic Ops (XOR2 and MAJ3), (3) Arithmetic Ops, and (4) Control. As depicted in Fig. 5, when opcode is 000, μOp performs row copy. Copy instruction `AAP(src., des.)` (i) `ACTIVATE`s a source address (src.), (ii) `ACTIVATE`s a destination address (des.), and (iii) `PRECHARGE`s to prepare the array for the next access. When the opcode is 001, μOp performs a 3-input operation. We reserve four bits corresponding to SA's configuration bits shown in Table 1 corresponding to C_{nor},$C_{m/min}$,C_{nand}, and C_{mux}. Logic instruction `AP(addr., config.)` (i) `ACTIVATE`s three DRAM rows for in-DRAM computation according to config. bits, and (ii) `PRECHARGE`s to prepare the array for the next access. Opcodes 010-101 respectively represent simple arithmetic operations: `ADDi`, `SUBi`, `comp`, and `module` to manage the computation address in DRAM. Lastly, we reserve two opcodes for loops and termination in the FlexiDRAM control flow (`bnez`, `done`).

3.2.4 μProgram Generation. As shown in Fig. 6, FlexiDRAM takes four steps to generate μprograms, i.e., rewrite, allocation, generation, and triggering. Every row in FlexiDRAM has an address stored in a set of registers located in the control unit. Thus to activate

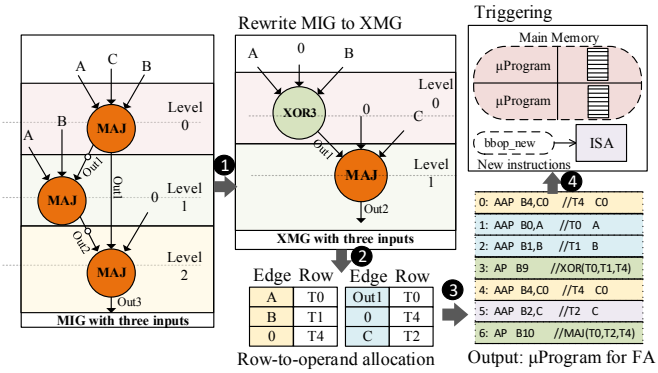

Figure 6: μProgram generation/execution steps in FlexiDRAM.

the row, the control unit specifies the address. One μOp executes only a 1-bit operation, so for n-bit operation, the framework needs to repeat this operation n times. For example, in CNN, the size of the filter is not constant. Therefore, FlexiDRAM cannot do operations on all columns directly. Therefore, n depends on the length of input data, and that is why the loop and termination opcodes are considered in μops.

Rewrite. FlexiDRAM generates the optimal XMG node consisting of MAJ3 and XOR2 with one constraint, i.e., using one specific logic μOp (MAJ3 or XOR2) in every logic level. This considerably increases the parallelism and significantly reduces the number of intermediate write-back operations. In Fig. 6 ❶, we observe that XMG reduces the redundant use of inputs by rewriting into XOR. Level 0 only contains XOR3, and level 1 consists of MAJ.

Allocation. This step aims to allocate address lines sequentially and optimize the command order to minimize execution time and complexity. All the computations are executed in B-group, which means FlexiDRAM has to copy the data enrolled for computation to B-group. The primary function of DRAM is to store data. If many rows are allocated for calculation, the loss will outweigh the gain. Therefore, we design an allocation algorithm to minimize the number of rows required for the B-group. Another essential factor is that we use TRA in operation, which will overwrite the three data lines involved in the process with output data. Referring to the dependencies between nodes in XMG, we can reuse some output rows directly as input to reduce row allocation operations while avoiding overwriting valid data as depicted in Fig. 6 ❷.

Generation. After allocating the rows for computation, FlexiDRAM utilizes the addresses to generate μops. As shown in Fig. 6 ❸, μops are generated sequentially. First, finding the original inputs and node, the framework allocates the inputs and generates row copy operations. Then it looks down for the dependent node of the output. If there is, it will reserve the output row and allocate the new inputs. If the remaining unallocated rows are enough, they will continue to allocate by address. Otherwise, the new inputs will overwrite the previous rows. In FlexiDRAM, we save the smallest address in our design and put it into the next operation. Then, FlexiDRAM goes to the next node and generates the μops till the end of the XMG tree. Once going through all the inputs and the generation of μOp is complete, we need to put it into the loop to repeat n times. Here the FlexiDRAM generates the arithmetic and control μops. Arithmetic μops are for calculating the cycles in n-bit

Table 3: Performance comparison of processing-in-DRAM platforms (No. of gates after rewrite step/ in-DRAM cycles).

Benchmarks		ISCAS'85				EPFL				ITC'99		
DRAM Platform	base function	c17	c880	c2670	c3540	log2	mult	sqrt	square	ITC_b01	ITC_b05	ITC_b20
FlexiDRAM	XOR2-MAJ3	7/3	272/27	396/26	857/38	22107/321	16015/139	17368/7064	12804/156	15/5	548/46	9383/72
Ambit [18]/ SIMDRAM [10]	MAJ3-NOT	6/6	325/40	717/34	1038/67	32060/791	27062/534	24618/9155	18484/497	25/9	793/93	12186/124
DRISA-3T1C [12]	NOR2	7/4	320/38	697/34	1057/67	32156/824	27122/577	24695/9195	18516/518	25/9	688/84	12232/148
DRISA-1T1C [12]	NAND2	7/8	320/70	697/68	1057/145	32156/1648	27122/1154	24695/18,390	18516/1036	25/18	688/168	12232/296
[2]/ GraphiDe [4]	MAJ5-MAJ3	6/6	295/29	710/30	1038/67	31055/712	26813/490	24002/8934	18156/458	25/9	781/90	11957/98

computation. Finally, after generating the set of μops, FlexiDRAM packs them into a μprogram and stores it into memory and each μprogram can be triggered by ISA to implement the operations.

Triggering. Figure 6 ❹ shows that the memory can store multiple μprograms, and FlexiDRAM executes them by activating the B-group rows according to the decoded address to perform the corresponding operation. Here, the control unit plays a compiler's role, translating the programs to let the machine understand them. Meanwhile, the control unit is transparent to users. For example, when the user writes a program at a high level, the computer compiles it and sends the operations to the control unit. Then the control unit will access the register when receiving the location message and load a μprogram when receiving a request message. Users can make a try to optimize the run-time with their algorithms.

3.2.5 Programming Interface. When the μprograms are stored in the memory, the CPU will load and execute them. However, only one μprogram can be loaded at once, and it will cause significant latency if the control unit cannot load the correct μprogram. Therefore, we propose to extend the ISA to improve the performance. The primary purpose is to give the most efficient sequence to load the μprograms. There are three considerations for the CPU's ISA extension. (c1) Locate the following program. When a program is executing, the controller unit needs to predict the destinatio dress and locate it. (c2) Determining when to load the next prog When the loop in the previous program terminates, the control needs to load the next program immediately. (c3) Determinin weight of the programs. If two programs can be loaded simul ously, which one can be executed first is determined by the co unit. Designing a set for bulk bitwise operations (bbops) to int with the framework is left for future work.

4 EXPERIMENTAL RESULTS

We demonstrate the advantages of the FlexiDRAM through a c layer evaluation framework as shown in Fig. 7. We first devel FlexiDRAM's sub-array with new peripherals in Cadence Sp with 45nm NCSU PDK library [1] to verify the functionality achieve the performance parameters. The memory controlle cuits are designed and synthesized by Design Compiler with a 4 library. To efficiently realize the MIG and XMG representation extensively modified ABC Berkeley [7], Advanced Logic Synt and Optimization tool (ALSO) [8], and CirKit [13] to rewrit logic circuits and extracted the logic net-lists. We impleme FlexiDRAM using gem5 [6] and exported the memory stats performance into an in-house optimizer, taking the circuit-l parameters as the input. We ran CNN workloads and comp it with existing processing-in-DRAM platforms including A[18], DRISA [12], GraphiDe [4], and ReDRAM [5]. To have a fair comparison, we followed the system configuration in SIMDRAM [10] for all PIM platforms as follows: 1-core/ out-of-order/ 4GHz; L1 Cache: 32kB 8-way with 64B line; L2 Cache: 256kB 4-way with 64B line; Memory Controller: 8kB row size, DDR4-2400, 1 channel, 1 rank, 16 banks. The baseline von-Neumann computing platform is the NVIDIA GTX 1080Ti Pascal GPU with 3584 CUDA cores running at 1.5GHz (11TFLOPs peak performance).

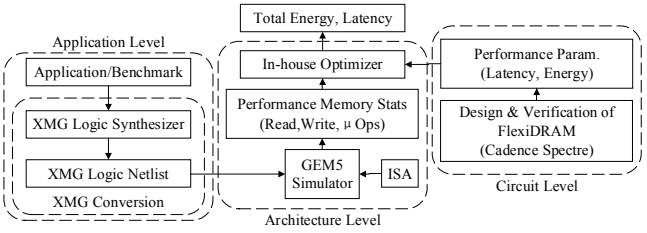

Figure 7: Evaluation framework.

Functionality Analysis: The transient simulation result of FlexiDRAM to implement single-cycle in-memory X(N)OR2 and MAJ3/MIN3 operations is depicted in Fig. 8. After setting the configuration bits, if two operands are stored in WL_{b1} and WL_{b2} at the B-group and the WL_{b3}'s capacitor (D_k) is initialized by

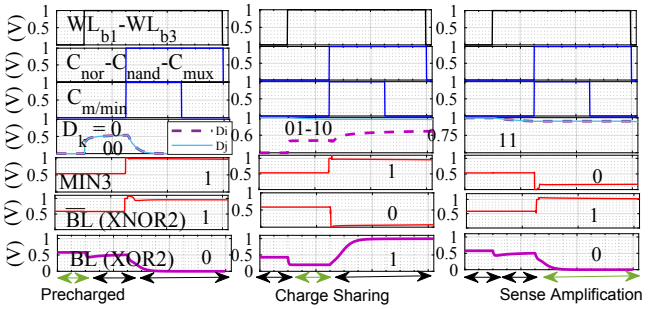

Figure 8: Transient simulation waveforms.

Memory Cycle Reduction: We compare the circuit complexity and performance of FlexiDRAM with other PIM designs by rewriting 11 various-size benchmarks extracted from ISCAS85, EPFL, and ITC99. For this experiment, we first configured the total capacity of DRAM as 16 MB with 1024 rows, 256 columns, and 64 sub-arrays. Table 3 reports the total number of the gates (i.e., base function) and the depth required to map each benchmark. For example, FlexiDRAM requires 7 XOR2/MAJ3 organized in 3 logical levels (3 compute. cycles) to implement c17. While FlexiDRAM's XMG

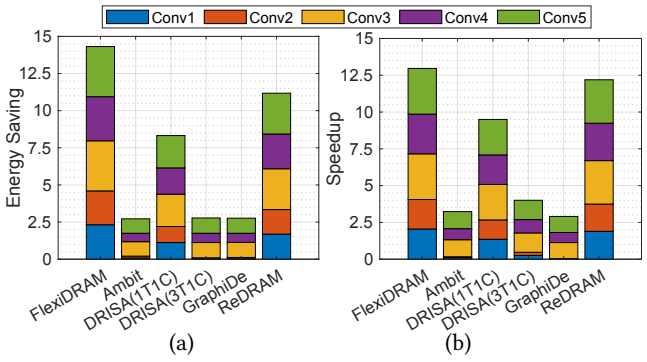

Figure 9: (a) Normalized energy saving and (b) Speedup over GPU running AlexNet kernels.

does not give a competitive performance in small-scale benchmarks such as c17 or c880, it will significantly outperform all counterpart designs when the size of the benchmark increases. For EPFL's log2, our platform can reduce the number gates and in-DRAM computation cycles by ~28.8% and ~55%, respectively, when compared to the fastest designs, i.e., QRA-based GraphiDe [4] and [2]. It is noteworthy that by simultaneously activating more than three cells, the deviation on the BL will be smaller. This not only lengthens the logic computation but also reduces the reliability of the operation. Thus, we limited the CNNs' performance and energy evaluations to the SRA to TRA.

Performance and Energy: Figure 9(a) and Fig. 10(a) report the energy saving of FlexiDRAM and various processing-in-DRAM platforms normalized to that of the GPU running two well-known binarized CNNs, i.e., AlexNET and GoogleNet, respectively. We observe the FlexiDRAM notably reduces the energy consumption for running X(N)OR-based operations compared with GPU and other PIM platforms. The GPU's energy consumption was measured with NVIDIA's system management interface and scaled-down by 50% to exclude cooling energy [12], etc. On average, FlexiDRAM achieves 15× energy saving over the GPU, while the most efficient design, i.e., ReDRAM, offers 11.2× energy reduction. Our experiment shows that FlexiDRAM consumes 0.9 nJ/KB to perform XOR2 operation, however, Ambit requires 5.5 nJ/KB.

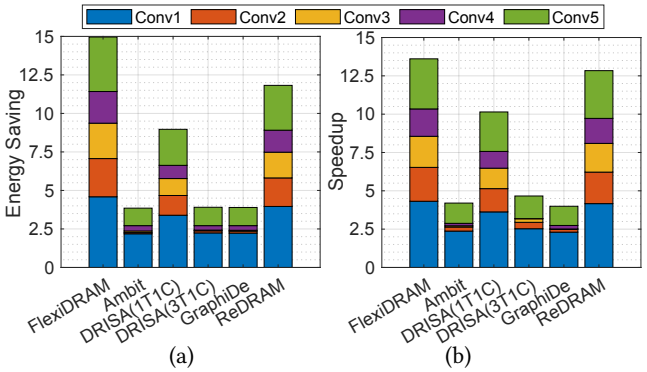

Figure 10: (a) Normalized energy saving and (b) Speedup over GPU running GoogleNet kernels.

Figure 9(b) and Fig. 10(b) show the performance of various processing-in-DRAM platforms, normalized to that of the GPU. We observe that FlexiDRAM outperforms the GPU, on average giving ~13× the performance of the GPU on both CNNs. Besides, it can be seen that FlexiDRAM outperforms counterpart PIM platforms, e.g., it achieves ~4.3× speedup when compared to Ambit [18] design. This mainly comes from 1) reduced-cycle and intrinsic XOR2 operation and 2) optimized address allocation in the framework.

5 CONCLUSION

In this paper, we presented the FlexiDRAM framework to support the efficient implementation of complex bulk bitwise operations in DRAM. FlexiDRAM generates an XOR-MAJ representation of the desired logic and appropriately allocates DRAM rows to the operands to execute any in-DRAM computation. The framework is supported by ISA and the interface required to compute in-DRAM operation. Our results demonstrate that averaged across two deep learning workloads, FlexiDRAM achieves ~15× energy-saving and 13× over the GPU, outperforming recent in-DRAM accelerators.

ACKNOWLEDGMENTS

This work is partially supported by a National Science Foundation grant (#ECCS-1710009).

REFERENCES

[1] 2011. *NCSU EDA FreePDK45.* http://www.eda.ncsu.edu/wiki/FreePDK45:Contents
[2] Mustafa F Ali et al. 2019. In-memory low-cost bit-serial addition using commodity DRAM technology. *IEEE TCAS I: Regular Papers* 67 (2019), 155–165.
[3] Mohamed W Allam et al. 2000. High-speed dynamic logic styles for scaled-down CMOS and MTCMOS technologies. In *ISLPED*. ACM, 155–160.
[4] Shaahin Angizi and Deliang Fan. 2019. Graphide: A graph processing accelerator leveraging in-dram-computing. In *GLSVLSI*. 45–50.
[5] Shaahin Angizi and Deliang Fan. 2019. Redram: A reconfigurable processing-in-dram platform for accelerating bulk bit-wise operations. In *ICCAD*. IEEE, 1–8.
[6] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39 (2011), 1–7.
[7] Robert Brayton and Alan Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*. Springer, 24–40.
[8] Zhufei Chu et al. 2019. Structural rewriting in XOR-majority graphs. In *ASP-DAC*. 663–668.
[9] João Dinis Ferreira, , et al. 2021. pluto: In-dram lookup tables to enable massively parallel general-purpose computation. *arXiv preprint arXiv:2104.07699* (2021).
[10] Nastaran Hajinazar et al. 2021. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *asplos*. 329–345.
[11] Tadahiro Kuroda et al. 1996. A 0.9-V, 150-MHz, 10-mW, 4 mm/sup 2/, 2-D discrete cosine transform core processor with variable threshold-voltage (VT) scheme. *IEEE JSSC* 31 (1996), 1770–1779.
[12] Shuangchen Li et al. 2017. Drisa: A dram-based reconfigurable in-situ accelerator. In *MICRO*. IEEE, 288–301.
[13] Giulia Meuli et al. 2022. Xor-And-Inverter Graphs for Quantum Compilation. *npj Quantum Information* 8, 1 (2022), 1–11.
[14] Shin'ichiro Mutoh et al. 1995. 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS. *IEEE JSSC* 30, 8 (1995), 847–854.
[15] Keivan Navi et al. 2009. A novel low-power full-adder cell with new technique in designing logical gates based on static CMOS inverter. *Microelectronics Journal* 40 (2009), 1441–1448.
[16] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot chips 23 symposium (HCS)*. IEEE, 1–24.
[17] Vivek Seshadri et al. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Micro*. 185–197.
[18] Vivek Seshadri et al. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *Micro*. ACM, 273–287.
[19] George Sideris. 1973. Intel 1103-MOS memory that defied cores. *Electronics* 46 (1973), 108–113.
[20] Mathias Soeken et al. 2017. Exact synthesis of majority-inverter graphs and its applications. *IEEE TCAD* 36 (2017), 1842–1855.