XMG-GPPIC: Efficient and Robust General-Purpose Processing-in-Cache with XOR-Majority-Graph

Chen Nie*
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, China
chen.nie@sjtu.edu.cn

Xianjue Cai*
UM-SJTU Joint Institute
Shanghai Jiao Tong University
Shanghai, China
shashantao@sjtu.edu.cn

Chenyang Lv*
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, China
lvchenyang@sjtu.edu.cn

Chen Huang
UM-SJTU Joint Institute
Shanghai Jiao Tong University
Shanghai, China
kouchin@sjtu.edu.cn

Weikang Qian†
UM-SJTU Joint Institute and MoE Key Lab of AI
Shanghai Jiao Tong University
Shanghai, China
qianwk@sjtu.edu.cn

Zhezhi He†
School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, China
zhezhi.he@sjtu.edu.cn

ABSTRACT

Recent advances in processing-in-cache (PIC) have enabled general-purpose, high-performance computation with bit-serial computing techniques. Its outstanding performance relies on efficient hardware design, and also the software stack (i.e., Logic Compiler, LC) that converts a high-level function into compact PIC instructions to be executed. Since XOR-Majority-Graph (XMG) is one of the most efficient forms to represent a Boolean function, designing the PIC with XMG can further improve the performance. Thus, we propose an efficient and robust General-Purpose PIC using XMG, aka. XMG-GPPIC, with designs in both hardware and software. For the hardware part, we propose a micro-architecture of XMG-GPPIC supporting XMG operation. To improve computing efficiency and robustness against non-ideal effects, we highlight our novel designs of inversion fusion and temperature compensation. For the software part, we develop the XMG-LC for optimized compilation for GPPIC, which includes two main steps of synthesis and scheduling. In the synthesis, we propose a multi-line reinforcement learning agent to search the optimal synthesis flow for the best end-to-end GPPIC performance. In the scheduling, we minimize the memory footprint occupied by the computation and support inversion fusion for instruction reduction. Our design reduces the number of operations by 67.7% on average w.r.t a majority-inverter-graph-based prior work, and the average end-to-end energy-delay product is 50.2% and 13.1% lower than our XMG-based naïve and heuristically optimized baselines, respectively. At the system level, our design outperforms compute cache with and-inverter-graph-based LC by 77% and 64% in terms of throughput and efficiency, respectively.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Reconfigurable computing**.

KEYWORDS

In-Memory Computing; bit-serial; logic synthesis; SRAM

ACM Reference Format:
Chen Nie, Xianjue Cai, Chenyang Lv, Chen Huang, Weikang Qian, and Zhezhi He. 2023. XMG-GPPIC: Efficient and Robust General-Purpose Processing-in-Cache with XOR-Majority-Graph. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3583781.3590288

1 INTRODUCTION

Recently, the Processing-in-Cache (PIC) [1–3] has drawn great research interest as it prominently mitigates the expensive data communication between processor and cache, thus leading to a great boost in performance. As the initial PIC design [1] only provides bulk bit-wise logic (`AND`, `OR`, etc.), the followed design in [3] supports more complicated arithmetic operations (e.g., N-bits multiplication) via bit-serial technique. However, a framework supporting General-Purpose computing in PIC (aka. GPPIC) is still absent.

To achieve the above goal, further designs in both software- and hardware-end are required. From the software perspective, a compiler is necessary for GPPIC to convert a high-level function into hardware-supported instructions. Specifically, such a compiler can decompose an arbitrary Boolean function expressed as a logic netlist into the logic primitives with the least area or delay, then lowered into instructions. We name such compiler for GPPIC as *Logic Compiler* (LC). If the Boolean function is represented by a Directed Acyclic Graph (DAG), each node in the optimized DAG will be converted into an instruction correspondingly. Currently, the well-known DAGs include And-Inverter-Graph (AIG), Majority-Inverter-Graph (MIG), and XOR-Majority-Graph (XMG), whose

*These authors contributed equally to this research.
†Corresponding Authors: Weikang Qian and Zhezhi He.

GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06…$15.00
https://doi.org/10.1145/3583781.3590288

efficiencies are listed from low to high. Therefore, to minimize the number of compiled instructions (fewer execution cycles), efficient DAG with less number of nodes is preferred.

Prior works [4–6] have contributed to the development of LC on PIMs that use ReRAM or DRAM memory, with the less efficient MIG. Note that, to design the LC for GPPIC using XMG, simply replacing the MIG with XMG will not succeed, as the currently available PIC micro-architecture cannot efficiently and robustly compute the primitives of XMG (i.e., 3-inputs majority, XOR, inversion, etc.). The hardware design challenges of GPPIC supporting XMG are, **1) Massive inversions hamper the throughput**: According to our analysis (Table 3), inversion operations take a great portion of compiled instructions with our XMG-based LC, thus lowering the GPPIC throughput; **2) PVT-Vulnerability**: under the variation of process-voltage-temperature (PVT), prior PIC circuit designs (e.g., compute cache [1]) with more memory rows activated (triple-row activation as discussed in Section 3) may lead to erroneous computing, thus compromising the functionality.

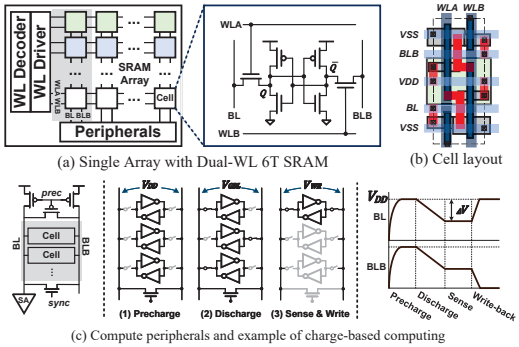
As a countermeasure, we propose a micro-architecture of GPPIC that can robustly and efficiently support the XMG primitives (aka. XMG-GPPIC). XMG-GPPIC can support efficient 3-input `XMG` operations in one cycle with fused inversion on any input and output bits. Standalone inversion is completely eliminated, which greatly reduces required instructions (cycles). Besides, we propose a duration-based temperature compensation method to counter the PVT vulnerability with negligible overhead. In addition to the contributions in hardware, we also develop an XMG-based LC as the software stack (Section 4) with two main steps of *synthesis* and *scheduling*. The multi-line reinforcement learning (RL)-based synthesis optimizes the XMG netlist, while the scheduling aims to reduce the runtime memory footprint and fuse all the inversions. Both of the above two features are designed for instruction reduction.

Overall, our contributions can be summarized as:

- **Hardware:** We propose a micro-architecture of General Purpose PIC supporting XMG operations (XMG-GPPIC). With our unique hardware design, standalone inversions can be completely fused into other operations to minimize the instructions and improve the throughput. Besides, our temperature compensation technique helps eliminate the computing error resulting from PVT variations.
- **Software:** We develop an XMG-LC to compile a high-level function into GPPIC instructions via synthesis and scheduling. In the synthesis, a multi-line RL agent is empowered to search the optimal synthesis flow with the best GPPIC performance. In the scheduling, we minimize the computation memory footprint and support the inversion fusion for instruction reduction.

2 PRELIMINARY

Bit-serial PIM. Bit-serial processing refers to computing in a bit-by-bit fashion, which is adopted in parallel processing system [7] with thousands of processing elements, back in the 1980s. The recent SRAM [1, 3] and DRAM [6] based processing-in-memory (PIMs) support bit-serial computing, with the parallelism along memory bit-lines. With bit-serial processing, PIM can implement various functions by serially executing only a few types of operations.



(a) Single Array with Dual-WL 6T SRAM (b) Cell layout

(c) Compute peripherals and example of charge-based computing

Figure 1: Micro-architecture of proposed XMG-GPPIC

Logic synthesis. Logic synthesis consists of two steps: technology-independent synthesis and technology mapping. The first step transforms a high-level design into a technology-independent DAG representation. Such representations include AIG, MIG, XMG, etc. [8]. For identical Boolean functions using various DAG, it takes less number of nodes from AIG to XMG. By far, there have been PIM designs [4–6] using MIGs, but works using the more efficient XMGs are absent due to hardware design challenges.

3 HARDWARE DESIGN OF XMG-GPPIC

3.1 Micro-architecture of GPPIC

As depicted in Fig. 1a, we use a dual Wordline SRAM [9] with separate access control (i.e., WLA and WLB for Q and $\bar{Q}$ respectively), where the bit can be directly accessed in its original or negated form. The decoder is customized to activate both WLs for memory read&write or a single WL for computing. The other peripherals (Fig. 1c) include precharger, BL synchronizer, sense amplifier, computing logic, and write driver (not shown for simplicity).

3.2 Charge-based Computing and Instruction

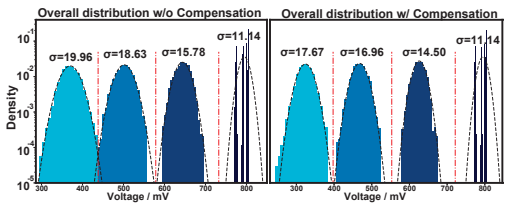
3.2.1 Computing mechanism. With the micro-architecture described in Section 3.1, multiple cells (rows) are simultaneously selected to conduct 3-input `XOR` or `MAJ` in one clock cycle. Such computing is charge-based and composed of the following four stages: *precharge*, *discharge*, *sense*, and *writeback*.

Precharge. The precharger pulls up both V_{BL} and V_{BLB} (i.e., V_{GBL}) to V_{DD}. Meanwhile, the BL synchronizer is turned on to keep the V_{BL} and V_{BLB} aligned by the end of the sense stage, where BL and BLB are wired as a global bit-line (GBL).

Discharge. We adopt triple-row activation (TRA) [10] to simultaneously activate three rows of SRAM cells. Note that, each cell is separately controlled by WLA and WLB, and only one access transistor will be switched ON. Depending on which access transistor is enabled, the node Q or $\bar{Q}$ of the cell (Fig. 1a) is routed to GBL to conditionally discharge the GBL. The activation of $\bar{Q}$ is equivalent to using the inverted data bit, without requiring extra INV operation. After the discharging, the GBL voltage is:

$$V_{\mathrm{GBL}} = \frac{C_{\mathrm{GBL}}V_{\mathrm{DD}} - \int_{t=0}^{t=\Delta t} n \cdot I_{\mathrm{disc}}(t)dt}{C_{\mathrm{GBL}}} = V_{\mathrm{DD}} - \frac{I_{\mathrm{disc}}\Delta t}{C_{\mathrm{GBL}}}n \quad (1)$$

where $n = \sum_{i=1}^{3} \mathbf{1}[g_i(x_i) = 0] \in \{0, 1, 2, 3\}$ is the number of activated cells discharging the parasitic capacitor of GBL C_{GBL}, where each contributes a discharging current I_{disc}. Each $x_i \in \{0, 1\}$

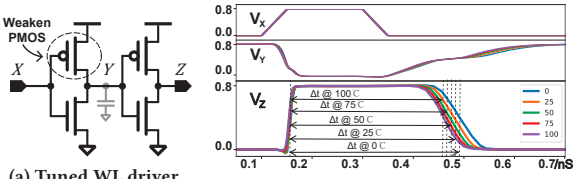


(a) Monte Carlo simulation

Func.	FR(%)
dec	0.05
cavlc	0.06
add	0.09
max	1.20
div	4.02
sin	5.67
sqrt	25.63
mult	35.03
log2	35.11

(b) Fail Ratio

Figure 2: (a) Worst case Monte Carlo simulation of V_{GBL} w/o and w/ temperature compensation. (b) Fail Ratio (probability of erroneous output) on EPFL benchmarks when w/o our temperature compensation.



(a) Tuned WL driver

(b) Transient at 0°C to 100°C

Figure 3: Duration-based temperature compensation by modulating the pulse width (activation time).

is a bit stored in a cell; g_i is the conditional inversion applied to the i-th bit; **1** is the logical indicator function. According to Eq. (1), with a given time window Δt, V_{GBL} can produce four voltage levels depending on the x_i's and g_i's.

Sense. The V_{GBL} is converted into digital output to realize the XMG operation. It is done by a 1.5-bit Flash ADC which generates a thermometer code to be further mapped into result values by subsequent logic. For example, the thermometer codes {000/001/011/111} are respectively mapped to {0/0/1/1} for MAJ and {0/1/0/1} for XOR.

Write-Back. The output is written back to a target cell by the write driver, like normal memory write. The write path is switched from the data bus to the compute outputs. Note that, the BL synchronizer is disabled in this stage to not disturb the write-back.

3.2.2 Instruction. The instruction of XMG-GPPIC can be written as ⟨opcode, inv_flags, in1, in2, in3, out⟩, where opcode ∈ {XOR, MAJ} specifies the operation type; inv_flags consists of four bits indicating the conditional inversion of the corresponding input/output; in1, in2, and in3 are the three input row addresses; out is the write-back row address.

3.3 Temperature Compensation

3.3.1 Temperature Variation. Transistors have varied ON-currents I_{ON} and leakage currents I_{leak} under different operating temperatures T (i.e., $I_{\text{disc}} = I_{\text{ON}} + I_{\text{leak}} \propto T$). Such variation can make the V_{GBL} levels deviate from their designated distributions and overlap each other (see Fig. 2a left), thus introducing errors. Our evaluation in Fig. 2b shows the impact of PVT by failure ratio, which becomes worse when the function becomes more complex (i.e., more nodes in DAG). Therefore, it is vital to preserve enough margin and ensure no overlaps between the V_{GBL} levels for error-free computing.

3.3.2 Duration-Based Compensation. Higher temperature T leads to larger I_{disc}, while $V_{\text{GBL}} \propto -I_{\text{disc}}\Delta t$ (Eq. (1)). To compensate for the variation of I_{disc}, we propose to modulate Δt to track the variation. Take a two-stage driver in Fig. 3a for example, we slightly tweak the driver to act as a temperature sensor that outputs a width-modulated pulse inversely proportional to T. The PMOS of the first stage shown in Fig. 3a is intentionally weakened while the other transistors remain unchanged. The transient curves in Fig. 3b illustrate the modulation w.r.t different temperatures. When a rising edge arrives at X, both Y and Z are instantly pulled down and up, as all the engaged transistors are of normal driving capabilities (unmodified sizing). However, when the falling edge arrives at X, Y is pulled up slowly by the weakened PMOS with an enlarged switching time. Thus, the impact of temperature variation will be amplified. Therefore, the transmission delay is almost fixed for rising edges but varies for falling edges, so the pulse width Δt is negatively correlated with temperature T, as shown in Fig. 3b.

Note that, memory read/write is not degraded as the rising edges determine the critical delay. Meanwhile, the Monte Carlo sampling in Fig. 2a (right) demonstrates the effectiveness of our solution, where overlapping of V_{GBL} levels is avoided with narrowed variance and enlarged margin. Such improvement in robustness empowers the error-free XMG-GPPIC.

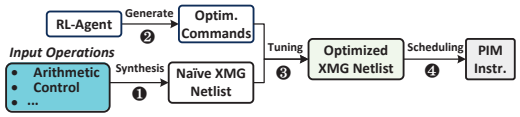


Figure 4: XMG-LC framework with synthesis and scheduling.

4 XMG-LC:XMG-BASED LOGIC COMPILATION

4.1 Framework Overview

We develop an XMG-based logic compiler framework for GPPIC synthesis and scheduling, as depicted in Fig. 4. The main steps in the frameworks can be specified as: ❶ A high-level input function is primarily synthesized into a naïve XMG netlist; ❷ The RL agent generates the optimal optimization commands that achieve the best hardware performance. ❸ The naïve XMG is tuned with the generated commands and obtains the optimized XMG. ❹ Scheduling is applied to obtain instructions with the best efficiency.

4.2 RL-framework for Optimized Synthesis

We leverage reinforcement learning to perform design space exploration and generate optimized XMG with the best end-to-end GPPIC performance. Our solution distinguishes from prior works [11, 12] in terms of parallel multi-worker space exploration and GPPIC-aware end-to-end optimization.

State space. We concatenate the extracted XMG topology with historical state and temporal information as the state vector $\boldsymbol{s} = [\boldsymbol{t}; \boldsymbol{r}; i; G_{\text{xmg}}]$, where $\boldsymbol{t} \in \{0,1\}^8$ is of one-hot format and indicates the past four actions, $\boldsymbol{r} = [r_{\text{cur}}, r_{\text{last}}]$ is the rewards (indicating GPPIC performance) of current and last step, i is the index in the command sequence, and G_{xmg} is the extracted topology.

Action space. Each action is an optimization command in Mockturtle [13] that optimizes the XMG without distorting its function.

Rewards. The reward is $R_t = ((r_{\text{last}} - r_{\text{cur}}) - r_{\text{base}})/r_{\text{init}}$, which includes the improvement of performance (i.e. energy, delay).

Algorithm 1 Monte Carlo policy gradient with state estimation.

Input: Policy $\pi(a \mid s, \theta)$, State-value $v(s, w)$
Output: Updated $\pi(a \mid s, \theta)$, $v(s, w)$
1: **for all** epochs i **do**
2: Generate a with π, select the best one for the following update
3: **for all** time steps t in the action sequence **do**
4: $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$, $\delta \leftarrow G - v(s_t, w)$
5: $w \leftarrow w + \text{Adam}(\nabla_w \ln v(s_t, w))$
6: $\theta \leftarrow \theta + \text{Adam}(\gamma^t \delta \nabla_\theta \ln \pi(a_t, s_t \mid \theta))$

Multi-line merit-based update method. A parallel RL algorithm (Algorithm 1) is developed to speed up the exploration, named multi-line RL. In each epoch, copies of the initial environment are created for multiple independent agents. Each agent then individually samples and generates the actions with the same policy. With all the generated actions applied, the best one is selected to update both the state-value estimation model and the policy model.

4.3 Scheduling

We implement a priority-queue-based scheduling algorithm [5] to map the optimized XMG netlist into XMG-GPPIC instructions in a hardware-aware fashion. The algorithm ranks all the nodes in the XMG netlist and decides the execution order that minimizes the array-level memory footprint (avoid inter-array communication) and therefore improves the computation efficiency. In each time step, the most favored available node with the highest ranking among all the un-executed nodes is added to the instruction list. Note that, the node is considered available only when 1) it has not been computed; 2) all fan-ins have been computed. Besides, all the inversions are fused into other operations during the scheduling.

Table 1: Specifications of benchmarking platforms.

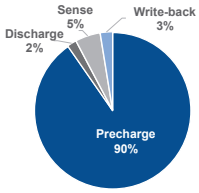
CPU	Intel Xeon® Silver 4210 @ 2.2GHz, 128GB DDR4 @ 2666MHz
ASIC	Design Compiler, TSMC-N28 technology, 32KB SRAM
PIC	1MB memory capacity, 32K computing parallelism 128KB per bank, 32KB per mat, 8KB per array

5 EVALUATION

5.1 Methodology

5.1.1 Benchmarks. The benchmarks (Table 2) consist of twelve representative functions in EPFL combinational benchmark suite [14] that are most probably applied in parallel scenarios. Both arithmetic and control functions are included to ensure generality.

5.1.2 Evaluation Methodology. We conduct in-depth performance evaluations to compare our design to CPU, ASIC, and prior works on PIC, with configurations listed in Table 1. **CPU:** The benchmarks are executed and verified using Verilator. The execution cost is evaluated following the method in [6]. Meanwhile, the average on-load CPU power consumption is monitored through the Intel RAPL API. **ASIC:** We use Design Compiler and TSMC 28nm standard cell library to synthesize and evaluate the area, delay, and power. Besides, a 32KB SRAM storing the input operands and final results is included as in [15], for a fair comparison. The performance is normalized according to the ratio of the ASIC area and the XMG-GPPIC area. **PIC:** The performance of our proposed GPPIC is assessed with an in-house cross-layer evaluator, based on circuit-level transient simulation, architectural formulaic estimation, and behavior-level cycle-accurate simulation. The memory cells and peripherals are both implemented with the TSMC 28nm PDK, and array-level post-simulation is conducted via Cadence Spectre to ensure functionalities. Besides, complete computations are simulated to acquire important specifications, such as per-cycle energy consumption and maximum delay. For the system level, we modify NVSim [16] to acquire memory read/write cost and area of common memory components. All these specifications are packed into our cycle-accurate simulator, which implements all the operations to evaluate performance, throughput, and energy efficiency. Besides, we compare with the compute cache [1], which is an SRAM-PIM and performs AIG-based logic operations via our LC.



(a) Energy breakdown

Component	Area (μm^2)
SRAM cells	9680
Decoder	1589
Control unit	840
Compute logic	402
Sense amplifier	186
Precharger	159
Total	12856

(b) Detailed area summary

Figure 5: Array energy breakdown and area summary.

5.2 Micro-Architecture Evaluation

5.2.1 Array-Level Performance. Obtained from circuit-level simulation, the average energy cost of each PIC operation (XOR/MAJ) is 46.93fJ. Besides, the four computing stages can be completed in less than 0.9ns and we use a 1GHz clock. The energy breakdown of each stage is depicted in Fig. 5a, where the precharge consumes the most percentages owing to the large parasitic capacitance on BL. Meanwhile, a detailed area summary is listed in Fig. 5b, where the additional computation components have negligible area overhead.

5.2.2 Cross-Hierarchy Movement Analysis. Oversized functions incur data movements across arrays, mats, or even banks to accommodate the input/output and intermediate results, leading to additional overhead. Our evaluation shows intra-mat, intra-bank, and intra-rank movements consume 87.6, 157.3, and 439.0 fJ of energy per bit, and take 637.4, 673.3, and 755.5 ps of time, respectively. The primary concern is energy consumption, where long-distance single-bit movement costs more than the operation itself. Therefore, our XMG-LC takes memory footprint minimization as an optimization objective to reduce cross-hierarchy communications.

5.2.3 Inversion Analysis. XMG-GPPIC can execute single-cycle XOR and MAJ operations with configurable inversion on each input and output, instead of using additional INV operations. Thus, the performance gain of configurable inversion relies on the occurrence of inverted operation inputs. According to our analysis in Table 3, the average inversion rate is 26.44%. Therefore, our inversion fusion can effectively reduce the delay and memory footprint overhead.

5.3 Framework and System Performance

5.3.1 RL-Assisted Synthesis. As tabulated in Table 2, we compare our RL-assisted XMG-based synthesis to a MIG-based work [5] as well as XMG-based counterparts, in terms of the number of

Table 2: Comparison of LC for GPPIC with different DAGs (MIG and XMG) and optimization techniques.

Benchmark	MIG		XMG												
	MIG-based [5]		Ours (naïve)				Ours (heuristic)				Ours (RL)				
	#C	#R	#C	#R	Energy	Delay	#C	#R	Energy	Delay	#C	#R	Energy	Delay	Imprv.
int2float	428	41	259	48	0.0131	0.2590	221	64	0.0112	0.2210	209	56	0.0106	0.2090	10.57%
dec	777	258	304	264	0.0162	0.3097	304	264	0.0162	0.3097	304	264	0.0162	0.3097	0.00%
router	401	64	207	85	0.0105	0.2070	208	110	0.0106	0.2080	197	101	0.0100	0.1970	10.30%
cavlc	1124	102	697	133	0.0354	0.6970	609	149	0.0309	0.6090	592	161	0.0301	0.5920	5.508%
adder	1911	259	764	627	0.0873	1.1171	256	384	0.0567	0.5811	256	384	0.0577	0.5811	-1.69%
priority	2147	149	976	246	0.0496	0.9760	574	248	0.0291	0.5740	535	240	0.0272	0.5350	13.13%
max	4996	579	2865	883	0.3428	4.3010	1985	860	0.3329	3.6020	1935	848	0.2838	3.2851	22.25%
sin	10223	402	4572	380	0.3131	5.1610	3814	397	0.2707	4.2489	3619	368	0.2477	4.0843	12.06%
sqrt	49782	323	31202	414	3.0014	41.515	9103	373	1.0414	13.318	9699	322	0.8877	12.575	19.52%
multiplier	56009	419	20160	1800	3.3366	33.421	14219	1847	3.0149	26.829	14251	1542	2.6112	25.190	18.68%
log2	60184	1256	24738	1238	2.7825	35.241	20276	1374	2.8604	32.318	21276	1315	2.5916	31.528	11.61%
div	147608	590	65533	731	6.4678	88.388	33355	602	3.2702	44.829	28379	619	3.1556	40.858	12.05%

(1) #C: number of clock cycles; #R: maximum number of memory footprint; Imprv.: EDP improvement over heuristic method
(2) Units of energy and delay are nJ and µs, respectively.

Table 3: Inversion rates of EPFL benchmarks.

Benchmark	% INV	Benchmark	% INV	Benchmark	% INV	Benchmark	% INV
adder	0*	max	41.27	log2	16.51	int2float	34.48
dec	14.12	sin	28.18	cavlc	34.87	priority	26.71
div	42.62	sqrt	33.69	multiplier	18.12	router	27.57

*adder is of regular ripple-carry type without inversion.

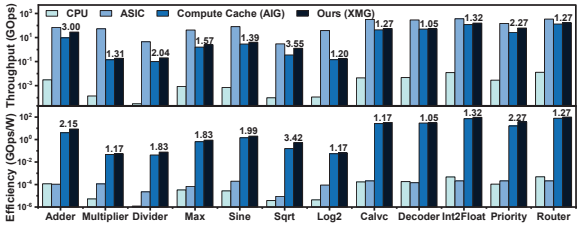


Figure 6: System-level performance comparison.

clock cycles (#C), the maximum number of memory rows occupied (#R), energy, and delay. Since [5] is implemented in ReRAM and does not report its energy or delay, we only report its #C and #R. Compared with [5], we achieve 67.7% fewer cycles (geometric mean). Moreover, our RL optimization achieves 50.2% and 13.1% lower energy-delay product (EDP) (geometric mean) than naïve and heuristically optimized XMG-based ones, respectively.

5.3.2 System Evaluation. We compare our system-level throughput and efficiency against CPU, ASIC, and prior PIC, as depicted in Fig. 6, with the improvement of ours w.r.t. compute cache [1] annotated above bars. According to the comparison, ASIC achieves the best throughput, due to the shortest delay. Both CPU and GPPIC are general-purpose platforms, but CPU is less efficient in processing a function in a bit-serial pattern. As for efficiency, XMG-GPPIC achieves orders-of-magnitude improvement over CPU and ASIC, due to the elimination of data movements and low-power mechanism. Our work also presents better throughput (77% on average) and efficiency (64% on average) than compute cache with AIG-based LC, as our design supports more efficient XMG-based computation and eliminates INV operations.

5.3.3 Scaling Analysis. We explore multiple array sizes to track the performance variation of the sqrt benchmark given fixed total memory capacity. As shown in Table 4, with larger arrays, both throughput and efficiency increase but then decrease sharply. The increase is caused by reduced cross-hierarchy data movements, as each array can accommodate more data. On the other hand, the decrease is caused by reduced parallelism, as there are fewer arrays.

Table 4: Design space exploration of array size.

Array size	64	128	256	512	1024
Throughput (GOps)	1.221	1.269	1.425	**1.688**	0.884
Efficiency (GOps/W)	0.103	0.319	0.536	**1.068**	0.548

6 CONCLUSION

In this work, we present the hardware and software stack of an XMG-based GPPIC, for higher efficiency and better robustness against PVT. With the improved performance, GPPIC can potentially execute more complex functions (i.e., netlist with more nodes) with lower energy and delay.

7 ACKNOWLEDGMENT

The authors acknowledge the support from National Natural Science Foundation of China (No.62102257), National Key R&D Program of China (2022YFB4500200 and 2020YFB2205501), and Lingang Laboratory Open Research Fund No.LG-QS-202202-11.

REFERENCES

[1] Shaizeen Aga et al. Compute caches. In *HPCA*. IEEE, 2017.
[2] Charles Eckert et al. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*, pages 383–396. IEEE, 2018.
[3] Wang et al. A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *JSSC*, 55(1):76–86, 2019.
[4] Pierre-Emmanuel Gaillardon et al. The programmable logic-in-memory (PLiM) computer. In *DATE*, pages 427–432. IEEE, 2016.
[5] Mathias Soeken et al. An MIG-based compiler for programmable logic-in-memory architectures. In *DAC*, pages 1–6. IEEE, 2016.
[6] Nastaran Hajinazar et al. SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM. In *ASPLOS*, 2021.
[7] Batcher et al. Bit-serial parallel processing systems. *TC*, 31(05):377–384, 1982.
[8] Testa et al. Logic synthesis for established and emerging computing. *Proceedings of the IEEE*, 107(1):165–184, 2018.
[9] Jeloka et al. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE JSSC*, 51(4):1009–1021, 2016.
[10] Vivek Seshadri et al. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *MICRO*, 2017.
[11] Keren Zhu et al. Exploring logic optimizations with reinforcement learning and graph convolutional network. In *MLCAD*. IEEE, 2020.
[12] Peruvemba et al. RL-guided runtime-constrained heuristic exploration for logic synthesis. In *ICCAD*, pages 1–9. IEEE, 2021.
[13] Soeken et al. The EPFL logic synthesis libraries. *arXiv:1805.05121*, 2018.
[14] Amarú et al. The EPFL combinational benchmark suite. In *IWLS*, 2015.
[15] Angizi et al. Mrima: An MRAM-based in-memory accelerator. *TCAD*, 39(5), 2019.
[16] Dong et al. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 31(7), 2012.