PIMLC: Logic Compiler for Bit-serial Based PIM

Chenyu Tang*, Chen Nie*, Weikang Qian†, Zhezhi He*
*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
†UM-SJTU Joint Institute and MoE Key Lab of AI, Shanghai Jiao Tong University, Shanghai, China
{tangcy98, chen.nie, zhezhi.he}@sjtu.edu.cn; {qianwk2}@huawei.com

***Abstract*—Recently, the bit-serial-based processing-in-memory (PIM) has evolved as a promising solution to enhance the computing performance of data-intensive applications, due to its high performance and programmability. However, it is absent that a compiler can automatically convert an arbitrary Boolean function (generic workload) into PIM instructions, with optimized scheduling *w.r.t.* the varying hardware resource and specification. To fill the gap, we develop a logic compiler for bit-serial-based PIM (PIMLC). In PIMLC, we propose a workload-resource-aware scheduling to minimize the execution latency of a given parallel workload. Thanks to PIMLC, PIM can achieve 15.55× and 19.03× speedup (geo-mean) for SRAM- and ReRAM-PIM respectively, compared to the naive scheduling of prior work. PIMLC is publicly available at: https://github.com/Intelligent-Computing-Research-Group/PIMLC.**

I. Introduction

Processing-in-memory (PIM) is an emerging computing paradigm to minimize processor-memory communication, by empowering the computing capability within memory (Sec. II-A1). One of the most promising PIM architectures is known as the logic-in-memory that performs vectorized logic operations in great parallelism[1]. By leveraging the bit-serial computing technique [1] that serially conducts the aforementioned vector logic operations (Sec. II-A2), PIMs can perform complex Boolean functions [2], [3] in parallel.

Unfortunately, mapping and scheduling a high-level vector computing workload (Boolean function) upon the bit-serial PIM is complicated. These PIM targets diverge in terms of the internal organization (*e.g.*, #blocks, #mats/block and etc), storage medium (SRAM [4], DRAM [3] and ReRAM [5]), and supported bit-wise vector operations (*e.g.*, `and`, `or`, `majority` and etc.). Assume a vector operation where the Boolean function $\mathcal{F}$ is performed on each vector element, the existing compilation work of PIM [3], [6], [7] utilizes a directed acyclic graph intermediate representation (DAG-IR) to depict $\mathcal{F}$ and perform optimization, *e.g.*, graph rewriting. Then, the optimized DAG-IR is converted into instructions for the specific PIM platform. As we will discuss in Sec. II-B, all the prior works of compilation for PIM only treated PIM as a single-instruction-multiple-data (SIMD) engine, and did not fully exploit available PIM resource and take workload size into the consideration during the compiling(scheduling).

This work is partially supported by National Natural Science Foundation of China (No.62102257), National Key R&D Program of China (2022YFB4500200). Corresponding author: Zhezhi He.

[1]In this work, the PIM refers to the logic-in-memory based architecture, rather than the memory crossbar (vector-matrix multiplication engine).

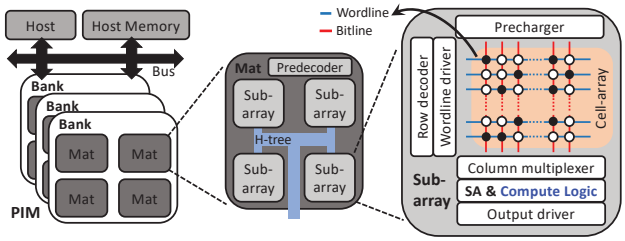


Fig. 1: Memory organization with PIM capability.

As the solution, we develop an ahead-of-time (AOT) logic compiler for the bit-serial-based PIM system, *aka. PIMLC*. As far as we know, PIMLC is the first work that supports workload-resource-aware scheduling. Our contributions are: **1) flexible support:** We develop *PIMLC* to compile high-level Boolean functions describing basic operators into the instructions of bit-serial PIM ahead of time. Our PIMLC supports Boolean functions taking different graph representations as its DAG IR, while the backend PIM system can be with SRAM- and ReRAM-based PIM micro-architecture. **2) scheduling optimization:** The core feature of PIMLC is scheduling optimization, where the compilation performs workload-resource aware scheduling (WRAS) to minimize the execution latency.

II. Preliminary

A. Bit-serial based PIM Architecture

1) PIM Architecture and ISA: PIM has been widely explored in prior works [3]–[8]. In this work, we focus on the PIMs inheriting the architecture of conventional random-access memory with minimum circuit-level modifications [4]. Its computation capability is embedded by inserting compute logic peripherals with sense amplifier (SA), as shown in Fig. 1. Besides, we take PIM as a co-processor, where data moves bidirectionally between the host memory and PIM. In Fig. 1, PIM is composed by *bank*, *mat* and *sub-array* [9], [10], which are connected through delicately designed multi-direction H-tree. The atomic computing unit of PIM is the sub-array. For each sub-array, it can perform the bulk bit-wise logic operation in parallel along its bit-lines (column of sub-array cells), which is known as *bit-line computing* [4]. Thus, a sub-array can be viewed as an atomic SIMD logic engine with intrinsic hardware synchronization, where the atomic vector length equals the number of bit-lines in one sub-array. In general, PIM operations can be simply divided into two categories, *i.e.*, *Computation* and *Communication*, as tabulated in Table I.

979-8-3503-4859-0/DATE24/© 2024 EDAA

TABLE I: Simplified PIM instruction set architecture (ISA).

Category	Instruction Format	Opcode Set
Computation	`opcode,src1,src2,src3,dst`	{`and,or,maj,xor,and3,or3,xor3,inv`}
Communication	`opcode,src,dst`	{`copy`}
	`opcode,src,dst`	{`load,store`}

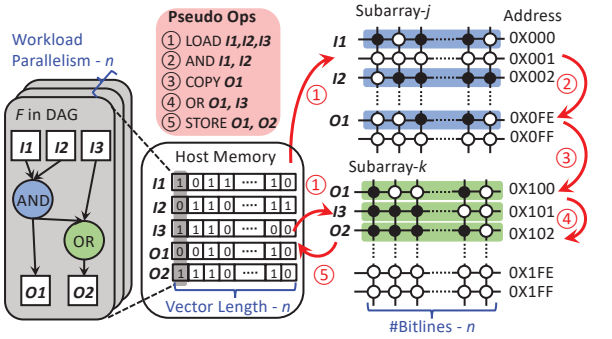


Fig. 2: Illustration of PIM processing the given workload $\mathcal{F}$.

Computation. Its instructions describe the on-chip intra-subarray operations, where data of single or multiple rows of sub-array act as inputs to perform specific bitwise logic, *e.g.*, `and`, `or`, `maj` (majority) and *etc*. `src1-3` are the source row addresses as inputs; `dst` is the destination row address for result write-back. While `and`, `or` and `xor` are 2-inputs, `and3`, `or3`, `xor3` and `maj` are 3-inputs (`inv` is 1-input).

Communication. Its instructions describe on-/off-chip data movements. For on-chip communication, we simply use `copy` to describe inter-subarray data movements from the source sub-arrays (`src`) to the destination ones (`dst`). The source and destination subarrays can either belong to the identical mat/bank or not. The off-chip communication relies on `load` and `store` to move data from and back to the host memory.

2) Bit-serial computing of PIM: Fig. 2 depicts an example of mapping a workload upon PIM. The workload is a Boolean function $\mathcal{F}$ expressed in a DAG-IR. It takes binary inputs $I1, I2, I3$ and output $O1, O2$, with a parallelism of n. As the DAG contains two operation nodes, *i.e.*, `and` and `or`, $\mathcal{F}$ can be conducted by executing them sequentially (*i.e.*, *bit-serial computing* [2]). Via bit-serial computing, computing $\mathcal{F}$ with parallelism-n is equivalent to performing element-wise logic with bit-vectors $\boldsymbol{I1}, \boldsymbol{I2}, \boldsymbol{I3}, \boldsymbol{O1}$, which takes five instructions ①-⑤ (marked by red arrow in Fig. 2). Note that, the prerequisite to perform bit-line computing is storing all the operands within identical sub-array (*sub-array locality*). Thus, as bit-vector $\boldsymbol{I3}$ is stored on subarray-k and to perform ④ `or` $\boldsymbol{O1}$, $\boldsymbol{I3}$, $\boldsymbol{O1}$ is copied from subarray-j to subarray-k. Furthermore, support for vector-scalar operations in [8] is in our plan while we only focus on vector-vector operations here.

B. Prior Compilers of PIM

Several works [3], [6], [7] have developed naive compiling methodologies for bit-serial PIM. [6] describes a ReRAM-based PIM micro-architecture that converts majority-inverter graphs (MIG) into operation sequences. Its successor [7] proposes an automatic compilation methodology for such conversion, along with optimization of both MIG rewiring and MIG-to-program conversion. [7] formulates the optimization problem w.r.t. MIG-based ReRAM PIM with dedicated considerations such as ReRAM endurance. SIMPLER MAGIC [11] focuses on memristor-aided-logic-based PIM and aims to maximize throughput via unique mapping principles. Based on MIG-based compilation flows, SIMDRAM [3] constructs a comprehensive system using DRAM technology. In summary, all the aforementioned works counter the two main drawbacks:

- They all treat the entire PIM as a SIMD engine, i.e., all sub-arrays only execute the same operation simultaneously, which may lead to under-utilized hardware resources if DAG size and n are small.
- Automated scheduling optimizations considering performance (e.g., throughput), available resource (e.g., #sub-array) and workload size (DAG size and n) are absent in prior instruction scheduling of PIM compilers.

III. PIMLC Framework

A. Framework Overview

As shown in Fig. 3, PIMLC takes the inputs of *1) Workload*: target workload $\mathcal{F}$ with parallelism n; *2) PIM Configuration*: memory size and architecture setting. Then, PIMLC outputs optimized instruction streams for the target PIM. PIMLC contains three modules: ❶ PIM specification generator (PIM SpecGen), ❷ compiler and ❸ simulator.

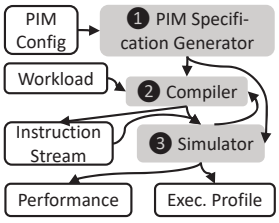


Fig. 3: PIMLC overview.

❶ PIM SpecGen provides detailed hardware specifications of target PIM to the compiler and simulator. Once inputting the memory size and PIM technology (e.g., SRAM/ReRAM), it optimizes the memory organization and then generates #banks, #mats per bank, #sub-arrays per mat, #bit-lines and #word-lines/sub-array, etc. Moreover, the energy and latency model of nvsim [10] is inherited to provide part of the circuit-level performance, e.g., energy and latency of instructions.

❷ Compiler first takes a workload $\mathcal{F}$ and converts it into a DAG-IR. The DAG-IR can be an and-or-inverter graph (AOIG), MIG, XMG [12], etc., but we focus on AOIG in this work. In this step, synthesis optimizations are taken to optimize DAG. Then, the compiler performs the *workload-resource aware scheduling* (WRAS) on DAG for optimized performance (discuss in Sec. IV), considering both workload characteristics and available PIM resources. Finally, the scheduled operations are translated into PIM instructions.

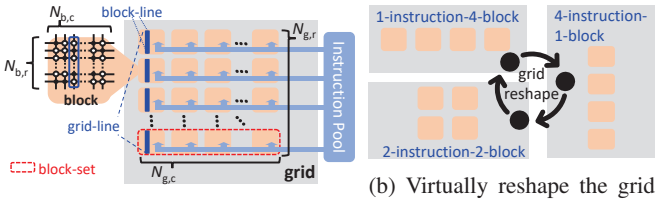
❸ Simulator is integrated to rapidly evaluate the performance of generated codes, via the embedded analytical energy and latency model. It mainly plays two roles: 1) end-to-end performance evaluation w.r.t the given workload and PIM hardware; 2) interacting with the compiler during the scheduling to evaluate the quality of the schedule on the fly.

B. PIM Hardware Abstraction

Hardware Abstraction. To mathematically model the compiling(scheduling) problem, we first introduce a hardware abstract of PIM. It includes the concepts of *block*, *block-set*,

TABLE II: Key Concepts of PIM Hardware Abstraction.

Definition	Description	Size (i.e., Data Footprint)
block	Abstraction of a sub-array (atomic SIMD engine).	$N_{\text{b,r}} \times N_{\text{b,c}}$
block-set	One row of the grid (extended SIMD engine)	$N_{\text{b,r}} \times (N_{\text{g,c}} \cdot N_{\text{b,c}})$
grid	Two dimensional block arrays (MIMD engine).	$(N_{\text{g,r}} \cdot N_{\text{b,r}}) \times (N_{\text{g,c}} \cdot N_{\text{b,c}})$
block-line	One column of the block (virtual thread)	$N_{\text{b,r}} \times 1$
grid-line	Stitched block-line (stitched virtual thread).	$(N_{\text{g,r}} \cdot N_{\text{b,r}}) \times 1$



(a) PIM hardware abstraction.

(b) Virtually reshape the grid for three computing patterns.

Fig. 4: Programming-oriented PIM hardware abstraction.

grid, *block-line* and *grid-line*, whose definitions are oriented from the data(memory)-layout organized virtually. The concepts are summarized in Table II with visual aids in Fig. 4a. As depicted in Fig. 4a, a block physically corresponds to a PIM sub-array (an atomic SIMD engine). The grid consists of available blocks as a stitched 2-dimensional array (total #blocks=$N_{\text{g,r}}N_{\text{g,c}}$). Observing a grid horizontally and vertically, each row (*block-set*) acts as an extended SIMD engine (sharing the identical instruction). Multiple block-sets compute asynchronously and interactively within a grid, making it a MIMD engine. In addition, each block comprises multiple block-lines, where a block-line physically corresponds to a column of bit-cells (a bitline) in sub-array. A grid-line is constructed by vertically stitched multiple block-lines, which serve the computation of one $\mathcal{F}$ at the same clock cycle.

Grid Reshape. PIMLC can compile for PIM with grids in different shapes. For instance, assume the grid contains four blocks, the grid shape $\langle N_{\text{g,r}}, N_{\text{g,c}}\rangle$ can be $\langle 1,4\rangle$, $\langle 2,2\rangle$ or $\langle 4,1\rangle$ as in Fig. 4b. To describe the operating pattern of the grid, we introduce the terminology multiple-instruction-multiple-block (MIMB). Thus the grid is a $N_{\text{g,r}}$-instruction-$N_{\text{g,c}}$-block engine, where grid supports #$N_{\text{g,r}}$ instructions launched asynchronously and each instruction is executed by #$N_{\text{g,c}}$ blocks synchronously. Moreover, we constrain the #rows and #columns of grid in power of two ($N_{\text{g,r}}, N_{\text{g,c}} = 2^i, 2^j; i,j \in \{0,1,...\}$) for easier potential hardware support.

C. Mapping and Communication Strategy

To minimize the communication cost when executing the instructions generated by PIM, we specify the mapping strategy and communication protocol in PIMLC hereafter.

Mapping Strategy. In Sec. III-B, each PIM sub-array is abstracted as a block in a 2D grid with the shape of $\langle N_{\text{g,r}}, N_{\text{g,c}}\rangle$. As the communication cost is proportional to the physical distance between sub-arrays in the H-tree interconnect, the blocks with intensive `copy` should be mapped closer. As `copy` only occurs between the bits within the same grid-line, we map blocks one by one via the *column-major order*, as shown in Fig. 5.

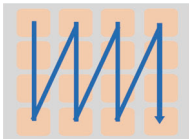


Fig. 5: illustration of column-major order mapping.

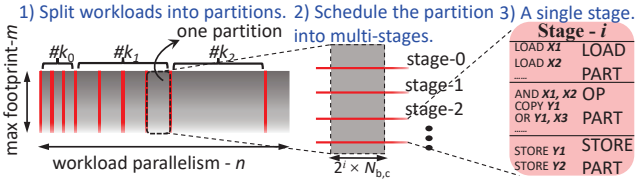


Fig. 6: Illustration of WRAS in PIMLC.

At mapping step-i, block-i is mapped to a sub-array closest to the sub-array mapped in step-$i-1$.

Communication Protocol. For on-chip communication, `copy` is between two selected block-sets, where `copy` should be synchronous (two blocks are occupied until `copy` is done) due to the PIM micro-architecture. Note that, such synchronous on-chip communication makes scheduling in PIMLC differ from the asynchronous scheduling in multiple-core processors or distributed systems. Thanks to our mapping strategy, on-chip communication between blocks can be parallel, if the H-tree forms a P2P connection. Otherwise, it can be serial or invalid, due to interconnect bandwidth, routing and grid shape.

IV. Workload-Resource Aware Scheduling

The objective of scheduling in PIMLC is minimizing the execution latency of a given workload.

A. Problem Statement and Our Solution

Assume a Boolean function $\mathcal{F}$ converts number of p primary inputs (PIs) into q primary outputs (POs), written as $\mathcal{F}: \{0,1\}^p \rightarrow \{0,1\}^q$, where each PI and PO are of binary value. $\mathcal{F}$ can be expressed by DAG, where each node corresponds to a computation instruction. To conduct bit-serial computing of $\mathcal{F}$ with parallelism of n, computation can be written as:

$$\mathcal{F}: \{0,1\}^{p\times n} \rightarrow \{0,1\}^{q\times n} \tag{1}$$

Ideal Scenario. We first assume one PIM block(sub-array) is sufficient to accommodate the processing of Eq. (1). From the perspective of memory footprint, the mapping and processing of Eq. (1) upon the PIM can be described as:

$$\mathcal{F}: \underbrace{\{0,1\}^{p\times n}}_{t_1\text{: Initialize (load)}} \rightarrow \underbrace{\{0,1\}^{m\times n}}_{t_i\text{: Max footprint}} \rightarrow \underbrace{\{0,1\}^{q\times n}}_{t_T\text{: Done (store)}} \tag{2}$$

where $t_i, i \in [0,1,\cdots,T]$ is the time-stamp of instruction, and T is the required #operations. Each instruction (from t_i to t_{i+1}) leads to a data update on PIM, where the latency depends on the category of instruction. The computation is initialized from t_0 (load from the host memory) and only data of PIs are mapped upon the blocks. After sequentially executing the vectorized Boolean operations, the computation achieves its maximum memory footprint of $\{0,1\}^{m\times n}$ at t_i, where the PIM accommodates the intermediate results, PIs and partial POs (*i.e.*, $m \geq q$). After t_T, the computation is done and only $\{0,1\}^{q\times n}$ (*i.e.*, POs) are stored to the host memory. Other data in the size of $\{0,1\}^{(m-q)\times n}$ are treated as "Do Not Care".

Practical Scenario. In practice, the block shape $\langle N_{\text{b,r}}, N_{\text{b,c}}\rangle$ maybe insufficient in both n and m dimensions for a specific

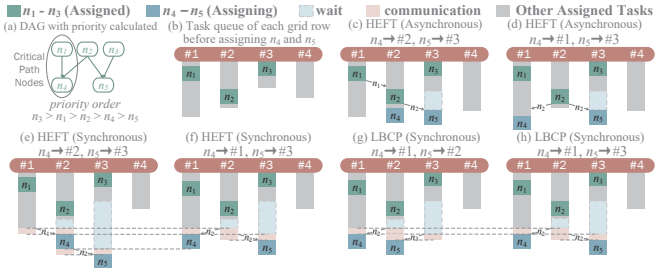


Fig. 7: Task placement of (a) a subgraph in a DAG; (b) task queues of 4 block-sets before assigning n_4, n_5; (c)(d) HEFT with asynchronous communication; (e)(f) HEFT with synchronous communication; (g)(h) LBCP with synchronous communication. Two schemes per scheduling algorithm are used to better illustrate the algorithm.

task. Thus it is mandatory to split the workload into multiple partitions, considering the limitation of PIM capacity and efficiency. It becomes even more complicated if the workload is extremely heavy (footprint of the entire grid is insufficient for $\{0,1\}^{m\times n}$), which will invoke multiple rounds of off-chip communications during the task execution life-cycle. As shown in Fig. 6, PIMLC conducts the scheduling *w.r.t.* the following strategies: **1) Split workload along dimension n.** We first allow the maximum footprint $\{0,1\}^{m\times n}$ to be split into multiple partitions along the dimension n. Each partition can be executed by the PIM grid in MIMB fashion with a specific grid shape (discussed in Sec. IV-B); **2) Schedule partition along dimension m.** If the memory capacity of the entire grid is insufficient, we can partition the task into multi-stages along m-dimension. As in Fig. 6, each stage may contain `loads` and `stores` at the beginning and the end.

B. Partitioning Workload via ILP

To realize the workload partition as described above, we model it as an integer linear programming (ILP) problem:

$$\arg\min_{\boldsymbol{k}} \sum_{i=0}^{\log_2^{N_{\text{block}}}} L_i k_i;\ \text{s.t.}\ \lceil \frac{n}{N_{\text{b,c}}} \rceil \leq \sum_{i=0}^{\log_2^{N_{\text{block}}}} 2^i k_i;\ 0 \leq k_i \quad (3)$$

where N_{block} is the total number of blocks in the grid. Due to the grid shape being "power of two", there exists $\#(\log_2^{N_{\text{block}}}+1)$ shapes for the grid. For the grid shape indexed by-i ($i \in \{0,...,\log_2^{N_{\text{block}}}\}$), there can be $\#k_i$ workload partitions to be executed by PIM in this specific shape. Furthermore, L_i is the latency of the potential workload partition using the i^{th} grid shape, which will be evaluated by PIMLC on the fly. Besides, the solution of $\boldsymbol{k}$ is bound by the first constraint of Eq. (3), which describes the padded workload parallelism ($\lceil \frac{n}{N_{\text{b,c}}} \rceil$) should not be greater than the accumulated workload parallelism of all partitions ($\sum_{i=0}^{\log_2 N_{\text{block}}} k_i \cdot 2^i$). The classic branch-and-cut is used to solve Eq. (3), which is not specified here due to its generality.

C. Operation Scheduling of Workload Partition

This subsection describes our scheduling generation w.r.t. the specified workload partition and grid shape $\langle N_{\text{g,r}}, N_{\text{g,c}} \rangle$.

Algorithm 1: Load Balancing based on Critical Path

Data: DAG $\mathcal{G}$ of Boolean function $\mathcal{F}$, sorted priority list Q, all block-sets $\mathbb{P}$, critical path CP
Result: scheduling scheme (i.e., PIM instructions)

Initialize the status S of all block-sets $p_i \in \mathbb{P}$;
while $Q \neq \emptyset$ **do**
  $v \leftarrow$ pop the first node in Q;
  if $v \in CP$ **then**
    $\text{LBS}(v, p_{\text{CP}}|S) = \infty$;
  else
    for *each* $p_j \in P$ **do**
      compute $\text{LBS}(v, p_j|S)$;
    end
  end
  Assign v to the p_j with $\arg\max_j \text{LBS}(v, p_j|S)$;
  Update S;
end

The schedule is designed to be before runtime as the compilation flow is AOT. We adopt and tweak the list-based scheduling [13] to determine operations and their orders. The primordial list-based method is used in a multi-core processor, where our PIM abstraction is similar to that but the on-chip communication of PIM is asynchronous. Thus, we decompose the list scheduling into two independent steps called *priority calculation* and *task placement* to be conducted sequentially.

Priority Calculation. The priority value of the nodes determines their priority in task placement. PIMLC first traverses the DAG of $\mathcal{F}$ and computes the priority value of each node considering both the weights of edges and nodes. We adopt the priority calculation method in upward rank [14], where weights are all positive and a parent node always gets a higher rank than its children, thus the result always follows the topological order. In this work, the weights of nodes are the latency of the corresponding PIM instruction. The weight of an edge is initialized as the maximum potential communication latency (*i.e.*, longest physical communication path in specified PIM). The actual communication cost can be reduced or even completely eliminated by improving the locality.

Task Placement. In Fig. 7, with calculated priority values of all nodes, we first assign nodes/instructions (n_4-n_5) to proper block-sets (#1-#4) via modified heterogeneous earliest finish time (HEFT) [14]. The vanilla HEFT is a heuristic that always assigns tasks to the block-set that can complete the task earliest. Fig. 7 (c-d) presents two potential scenarios of the original HEFT where communication launches immediately once results are generated (*i.e.*, asynchronous). However, it is infeasible for PIM to support asynchronous `copy` due to the expensive architecture cost. Thus, simply applying the asynchronous communication-friendly HEFT upon synchronous PIM leads to poor performance. As shown in Fig. 7 (e-f), HEFT can assign n_4 to either block-set #2 or #1 due to the same priority value, but their performance varies.

Therefore, we introduce *load balancing strategy based on critical path* (LBCP), which still utilizes a list heuristic but does not determine task placement solely based on the earliest

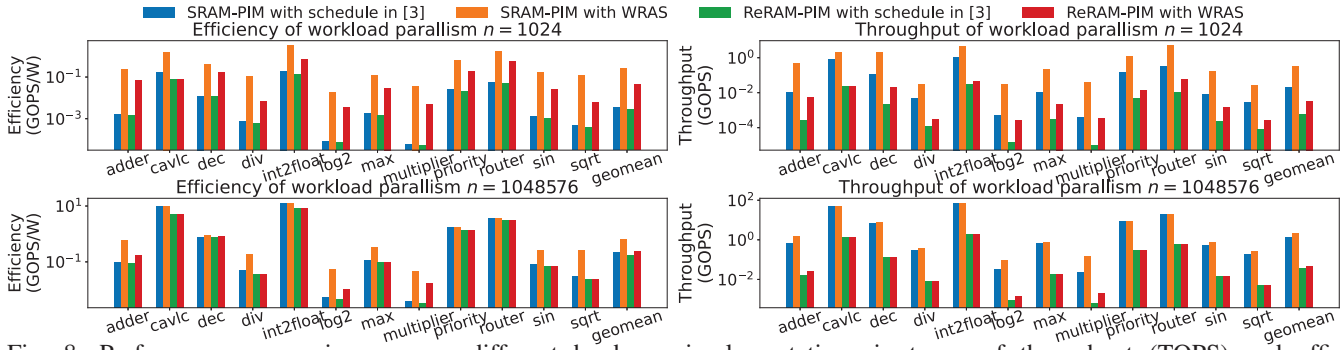


Fig. 8: Performance comparison among different hardware implementations in terms of throughput (TOPS) and efficiency (TOPS/W), on various Boolean functions in EPFL benchmark suite [15].

finish time. Instead, it takes into account both the *critical path (CP)* [16] and load balancing, to improve performance and resource utilization. CP in a weighted DAG is the set of nodes forming the longest directed path from a start node to an end node. We assign all the corresponding operations in the CP to a block-set and name it as *CP block-set*. Then, we intend to avoid any further communication from/to the CP block-set to minimize the overall execution latency. The correspondence of other nodes (not in CP) and block-sets are determined via conducting the scheduling algorithm (Algorithm 1). In the priority calculation step, one priority list Q is generated and sorted. The node corresponding to the highest priority value ($v_{Q_{[i=0]}}$) is popped to compute the load balancing score (LBS) when it is assigned to block-set p_j, i.e., LBS$(v_{Q_{[i=0]}}, p_j|S)$, where S is status record of prior node assignments. Then, the block-set with the highest LBS will be assigned to the current node. In Fig. 7(g-h), block-set $\#1$ is the CP block-set. n_4 is a node in CP which is assigned to block-set $\#1$ by default, while the assignment of n_5 is determined by our LBCP algorithm.

During the task placement process, to identify the best assignment, we monitor the status of each block-set. With operations assigned to each block-set, its memory footprint gradually filled up. Once no data can be overridden, we consider the current stage reaches completion followed by bulk data `store` to free the block-set for the successive stage, until the priority list is empty. Moreover, we reuse the data between stages to eliminate redundant off-chip data movements.

V. Evaluation and Discussion

A. Methodology

1) Benchmarks: To demonstrate the generality of PIMLC and the effectiveness of our proposed WRAS, we select twelve representative functions in EPFL combinational benchmarks suite [15] for evaluation. The selected functions include both arithmetic (adder, max, sin, sqrt, multiplier, log2, div) and control (router, int2float, dec, cavlc, priority) purpose. Besides, since those Boolean functions are highly customized and cannot be directly executed by the CPU, we additionally implement adder, multiplier, and divider with 8/16/32-bit operands, for fair comparison with modern CPU.

2) Evaluation Methods: The comparison includes CPU and SRAM-/ReRAM-PIM with different scheduling methods,

TABLE III: Hardware specifications of evaluation platforms.

CPU	AMD EPYC 75F3 32-Core Processor, 256GB DDR4@3200MHz
SRAM-PIM	2MB, 1.20GHz, 128KB/bank, 32KB/mat, (256 × 256)-bits/subarray
ReRAM-PIM	8MB, 0.04GHz, 512KB/bank, 128KB/mat, (256 × 1024)-bits/subarray

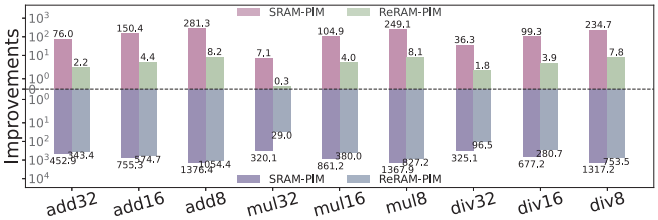


Fig. 9: The throughput (top half) and power efficiency (bottom half) improvements of PIM over CPU.

where the hardware specifications are tabulated in Table III. **CPU:** We implement C programs for the benchmarks and run multiple iterations to obtain the average results. The computing latency is measured by system in-built timers and tracking the program execution state. The power consumption is monitored by the Intel RAPL API, where only on-load consumption is considered (with non-load static power taken away). **PIM:** We leverage modified cacti [9] and nvsim [10] to evaluate the PIM single-cycle performance, using the area, clock, and energy specifications in [4] and [5]. The memory hierarchy is explicitly set as in Table III to model the architectural cost. As for the off-chip communications, we refer to [17] to estimate the energy cost of DDR4 (22pJ/bit) and HBM2 (4.0pJ/bit). The single-cycle performance and hardware specifications are then fed to our simulator to evaluate multi-cycle operations.

3) Verification: PIMLC verifies the execution of the instructions w.r.t the randomly generated operands and compares it with ground-truth computing results (generated by Verilator).

B. Performance Evaluation

1) WRAS Performance: We conduct evaluations on two workload parallelism ($n = \{1024, 1048576\}$), where the comparison results are shown in Fig. 8. For small n, SRAM- and ReRAM-PIM with WRAS boost the efficiency (throughput per watt) by 73.22 and 15.20, and the throughput by 15.55 and 5.72, compared to normal topological assigning order in [3]. For large n, the boost rates of efficiency are 2.80 and 1.30, and the boost rates of throughput are 1.52 and 1.18, for SRAM- and ReRAM-PIM respectively. We anticipate the

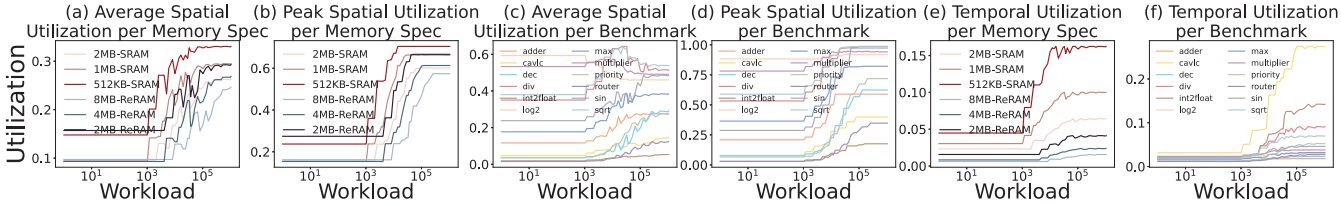


Fig. 10: Spatial and temporal utilization of PIM using PIMLC, with varying workload parallelism n and PIM resource.

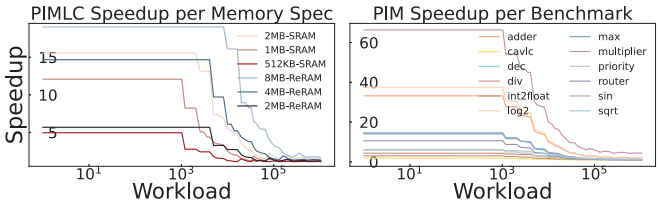


Fig. 11: Speedup *vs.* n and PIM resource. Curves in the left figure average over all benchmarks, and curves in the right figure average overall PIM configurations.

ReRAM-PIM has lower efficiency and throughput than SRAM PIM, as the energy and latency of writing ReRAM are much higher than SRAM, which has been discussed in [18] as well.

2) PIM & CPU Comparison: Additional comparisons are conducted between PIM and CPU using 8,16,32-bits fixed-point adder, multiplier and divider, which are the most basic operators in various algorithms. According to the results in Fig. 9, both SRAM-PIM and ReRAM-PIM achieve remarkable improvements, especially in simple low-width operations.

C. PIMLC Analysis

Speedup versus n and PIM resource. As the compilation is AOT, the scheduling cost is not discussed. We study the execution speedup of SRAM- and ReRAM-PIM with WRAS, compared to the counterparts without WRAS, under different n and resources. For resources, we alter the PIM by size. As shown in Fig. 11, the PIM is prone to higher speedup for smaller n, since PIMLC optimizes grid shape to increase #block-sets, thus allowing the operations in DAG of workload $\mathcal{F}$ to be executed concurrently. The speedup gradually reduces while n grows larger, as the PIMLC splits the workload into partitions using grid shape of $\langle 1, N_{\text{block}} \rangle$. Moreover, the right sub-figure of Fig. 11 also depicts the speedup for each EPFL benchmark. The top-3 speedups are achieved by multiplier, log2 and adder, whose #nodes are 31934, 37269 and 1274. In general, higher speedup is achieved by benchmarks with more #nodes, but also affected by the DAG topology as well.

Spatial utilization versus n and PIM resource. To understand resource management, we first check the average and peak spatial utilization (*i.e.*, occupation rate of PIM memory cells). Fig. 10 (a-d) shows the spatial utilization increases with larger n most of the time, as PIMLC is prone to leverage more spatial resources to optimize the PIM performance. Moreover, the top-3 average utilization rates are achieved by sqrt, sin and multiplier, which slightly differs from the case in Fig. 11, but are still among the largest DAGs.

Temporal utilization versus n and PIM resource. We also check the temporal utilization (*i.e.*, the time ratio of PIM is computing) to examine whether the communications (`load`/`store`/`copy`) stall the computing. In Fig. 10 (e-f), the temporal utilization enhances with increased n, since more computation is introduced thus mitigating the domination of communication operations. Similar to spatial utilization, the average temporal utilization rate is even lower (the best result is around 25%). The utilization can be even worse if the off-chip memory cannot instantly provide the data.

VI. Conclusion

In this work, we develop PIMLC for function compilation of PIM. We anticipate PIMLC as a foundational and mandatory brick of software infrastructure for future bit-serial PIM research, from the front-end applications to back-end hardware. Researchers from diverse communities can benefit from the PIMLC to estimate the performance gain of their own designs.

References

[1] Denyer *et al.*, *VLSI signal processing; a bit-serial approach*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[2] Wang *et al.*, "A 28-nm compute sram with bit-serial logic/arithmetic operations for programmable in-memory vector computing," *JSSC*, 2019.

[3] Hajinazar *et al.*, "Simdram: a framework for bit-serial simd processing using dram," in *ASPLOS*, pp. 329–345, 2021.

[4] Aga *et al.*, "Compute caches," in *HPCA*, pp. 481–492, IEEE, 2017.

[5] Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016.

[6] Gaillardon *et al.*, "The programmable logic-in-memory (plim) computer," in *DATE*, pp. 427–432, IEEE, 2016.

[7] Soeken *et al.*, "An mig-based compiler for programmable logic-in-memory architectures," in *DAC*, pp. 1–6, IEEE, 2016.

[8] C. Nie, C. Tang, J. Lin, H. Hu, C. Lv, T. Cao, W. Zhang, L. Jiang, X. Liang, W. Qian, *et al.*, "Vspim: Sram processing-in-memory dnn acceleration via vector-scalar operations," *IEEE Transactions on Computers*, 2023.

[9] Thoziyoor *et al.*, "Cacti 5.1," tech. rep., Citeseer, 2008.

[10] Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, 2012.

[11] Ben-Hur *et al.*, "Simpler magic: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *TCAD*, 2019.

[12] C. Nie, X. Cai, C. Lv, C. Huang, W. Qian, and Z. He, "Xmg-gppic: Efficient and robust general-purpose processing-in-cache with xor-majority-graph," in *Proceedings of the Great Lakes Symposium on VLSI 2023*, pp. 183–187, 2023.

[13] Wang *et al.*, "List-scheduling versus cluster-scheduling," *TPDS*, 2018.

[14] Topcuoglu *et al.*, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE TPDS*, 2002.

[15] Amarú *et al.*, "The epfl combinational benchmark suite," in *International Workshop on Logic & Synthesis*, no. CONF, 2015.

[16] Kohler *et al.*, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE TC*, 1975.

[17] C. Walden *et al.*, "Monolithically integrating non-volatile main memory over the last-level cache," *ACM TACO*, vol. 18, no. 4, pp. 1–26, 2021.

[18] S. Angizi *et al.*, "Accelerating deep neural networks in processing-in-memory platforms: Analog or digital approach?," in *ISVLSI*, 2019.