PhGraph: A High-Performance ReRAM-Based Accelerator for Hypergraph Applications

Long Zheng, *Member, IEEE*, Ao Hu, Qinggang Wang, Yu Huang, Haoqin Huang, Pengcheng Yao, Shuyi Xiong, Xiaofei Liao, *Member, IEEE*, and Hai Jin

***Abstract*—Hypergraph processing has emerged as an effective approach to analyze complex multilateral relationships in real-world scenarios. Existing hypergraph processing solutions based on conventional architectures are severely bottlenecked by off-chip memory accesses. In this article, we propose the first processing-in-memory (PIM)-featured ReRAM-based hypergraph accelerator, dubbed PhGraph, which facilitates performance- and energy-efficient hypergraph processing. On the hardware level, PhGraph integrates analog memristor-based PIM (with high-matrix-grained parallelism) and digital memristor-based PIM (for high-bipartite-edge-grained efficiency) into one standalone solution. On the software level, an overlap-aware hypergraph partitioning mechanism is proposed to polarize hypergraph workloads into matrix-formatted dense and bipartite-edge-formatted sparse partitions for performance acceleration using analog memristor-based PIM and digital ones, respectively. In addition, PhGraph is equipped with load-balanced partition scheduling and algorithm mapping co-designs to boost hardware utilization and efficiency. Experimental results show that PhGraph outperforms the state-of-the-art CPU-, FPGA-, and ASIC-based solutions by up to 4,309.81×, 547.13×, and 166.76× in terms of performance, and 36,416.11×, 924.12×, and 41.44× in terms of energy-savings, respectively.**

***Index Terms*—Heterogeneous accelerator, hypergraph processing, processing-in-memory (PIM).**

I. Introduction

THE CONVENTIONAL graph is restricted to capturing the pairwise relations of objects. However, real-world scenarios are often much more complex with multilateral relationships [1], [2]. For example, a paper may be published

Manuscript received 13 April 2023; revised 4 October 2023; accepted 4 December 2023. Date of publication 14 December 2023; date of current version 23 April 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4503400; in part by the National Natural Science Foundation of China under Grant 62322205, Grant 62072195, and Grant 61825202; and in part by Zhejiang Lab under Grant 2022PI0AC02. This article was recommended by Associate Editor G. G. E. Gielen. *(Corresponding author: Qinggang Wang.)*

Long Zheng, Ao Hu, Qinggang Wang, Yu Huang, Haoqin Huang, Pengcheng Yao, and Shuyi Xiong are with the National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the Zhejiang-HUST Joint Research Center for Graph Processing, Zhejiang Lab, Hangzhou 311121, China (e-mail: longzh@hust.edu.cn; ahu@hust.edu.cn; qgwang@hust.edu.cn; yuh@hust.edu.cn; hqhuang@hust.edu.cn; pcyao@hust.edu.cn; syxiong@hust.edu.cn).

Xiaofei Liao and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: xfliao@hust.edu.cn; hjin@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2023.3343228

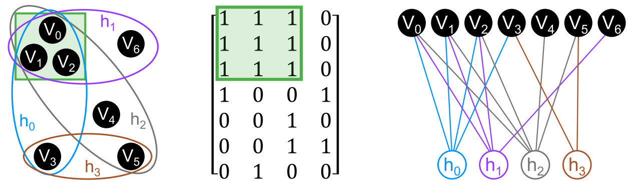


Fig. 1. Example hypergraph G, its incidence matrix M, and bipartite representation. (a) Hypergraph. (b) Incidence matrix. (c) Bipartite representation.

by more than two authors. Decomposing these polyadic relationships into pairwise ones may result in the loss of critical information [3]. Hence, hypergraph has emerged to naturally represent such multilateral relationships, where each hyperedge [e.g., h_0 in Fig. 1(a)] is able to connect any number of vertices [e.g., v_0, v_1, v_2, and v_3 in Fig. 1(a)]. Hypergraph processing can give rise to an expressive and efficient analysis of such relational data. Nowadays, hypergraph processing has been widely used in a large variety of domains, such as machine learning [4], drug discovery [5], and VLSI design [6].

Hypergraph processing typically relies on an iterative process with two basic kennels: 1) *hyperedge computation* and 2) *vertex computation*. The former uses active vertices to update the state of their incident hyperedges via an algorithm-specific hyperedge update function. Similarly, the latter utilizes the newly activated hyperedges to update vertices using a vertex update function. Previous studies [7], [8] have shown that hypergraph processing is typically memory-bound. Recent solutions, based on CPUs [1] and ASICs [7], [8], improve the locality of hypergraph processing by exploiting the intrinsic hypergraph structure overlap. However, they remain inadequate in addressing the memory bottleneck arising in hypergraph processing. Taking the state-of-the-art hypergraph processing accelerator XuLin [8] as a reference, especially for large hypergraphs, off-chip memory accesses can still be up to 80.19% of the total running time (as discussed in Section II-C).

Compared to the conventional von Neumann architecture, which relies on a separate computation-storage hierarchy, processing-in-memory (PIM) is a promising technology to improve memory-bound applications by integrating the processing elements within the memory. Past research [9], [10], [11], [12], [13] has demonstrated that resistive random access memory (ReRAM) can be effective in boosting ordinary graph processing. In consideration of the fact that an ordinary graph is a special case of a hypergraph where each hyperedge connects only two vertices, one feasible intuition for accelerating hypergraph processing is to directly reuse existing ReRAM-based graph accelerators to support the *hyperedge computation* and *vertex computation* kernels alternately. However, hypergraph processing exhibits unique *interkernel difference* and *overlap-induced irregularity* such that simply applying existing graph processing solutions to hypergraph applications cannot boost performance effectively and efficiently.

On the one hand, conventional graph applications only need to update the states of vertices with the same update operations in each iteration. In contrast, for hypergraph applications, the states of hyperedges and vertices are updated separately in two alternately executing kernels during an iteration. To support hypergraph processing, the existing ReRAM-based graph accelerator family has to handle the *hyperedge computation* and *vertex computation* kernels alternately in each iteration. However, the update logic of the two kernels is completely different, such that we have to remap kernel-specific data into crossbars once the executing kernel switches. Thus, this may incur significant extra resistance writing overhead, which can be up to 2× against the kernel execution time [14]. Even worse, this overhead can increase substantially as the number of iterations increases.

On the other hand, the existing ReRAM-based graph processing accelerator family, in order to harness the massive parallelism provided by the matrix-structured crossbar architecture, has to hold an assumption that crossbar cells are fully utilized [9], [10]. However, real-world hypergraph structure is often overlapped in the sense that the majority of vertices are associated with at least two hyperedges, and vice versa [7]. Thus, hypergraph processing exhibits overlap-induced workload irregularity. That is, heavy (light) overlapped topology substructure yields dense (sparse) regions in the hypergraph-induced incidence matrix [e.g., Fig. 1(b)]. Mapping these sparse submatrices into ReRAM crossbars leads to most of the cells being unused, incurring superfluous resistance writes and analog-signal conversion with limited performance and energy improvements. There have been some research efforts [10], [13], [15] for mitigating sparsity-induced performance degradation arising in the ReRAM-based architectures. However, few of them consider the unique overlapped feature of the hypergraph, leading to suboptimal improvements.

We observe that there is an opportunity to overcome the *interkernel difference* from the perspective of matrix-vector multiplication (MVM), in the sense that the *hyperedge computation* and *vertex computation* kernels can be uniformly formalized as a set of MVM operations, where only the input matrices are transposed to each other. Taking the PageRank algorithm as an example, its vertex and hyperedge update functions utilize the hypergraph-induced incidence matrix M [Fig. 1(b)] and its transposed one M^{T} for computation, respectively. Further, it is observed that there is no absolute winner for the row-grained digital memristor-based PIM (DPIM) and the matrix-grained analog memristor-based PIM (APIM) for processing all hypergraph workloads efficiently. By partitioning hypergraphs into dense and sparse workloads based on the overlap feature, APIM and DPIM can be leveraged to accelerate different (dense or sparse) workloads to maximize overall efficiency, further alleviating the *overlap-induced irregularity* issue. Thus, we are motivated to design a hybrid architecture that integrates APIM and DPIM technologies, which is capable of performing MVM-featured hypergraph processing with impressive performance and energy gains.

However, materializing the aforementioned idea remains challenging. First, the input matrix arising in the *hyperedge computation* kernel is transposed to the one arising in the *vertex computation* kernel. Alternately mapping the incidence matrix and its transposed one into ReRAM crossbars may incur superfluous resistance writes, thereby limiting gains that can be achieved. Second, to maximize the performance potential of the hybrid architecture, the most important imperative is to polarize the hypergraph-induced incidence matrix into either dense or sparse submatrices. Unfortunately, it is extremely difficult, if not impossible, to devise a fast yet efficient partitioning policy due to the complex intertwined connections between vertices and hyperedges. Third, after hypergraph partitioning, different tasks associated with the same vertex or hyperedge may be executed in distinct processing engines. This may produce a large number of intermediate results, incurring prohibitive reduction overhead.

In this article, we architect the first ReRAM-based hypergraph processing accelerator, dubbed PhGraph, which exploits a hybrid architecture with analog and digital memristor-based PIM technologies to boost overall efficiency. PhGraph features three novel designs. First, PhGraph is equipped with a transposed APIM crossbar architecture to support MVM and M^{T}VM operations directly without data remapping, thereby avoiding unnecessary analog-signal conversion and resistance writes. Second, we propose an overlap-aware HP strategy, which polarizes the hypergraph-induced incidence matrix into either sparse or dense submatrices to fully exploit the hardware potential of APIM- and DPIM-based processing engines. PhGraph also contains a hybrid partition scheduling scheme, which takes hyperedges and vertices into account equally. This enables maximizing data reuse for each processing engine and improves the load balance among all processing engines. Third, we also orchestrate a hierarchical reduction scheme to maximize the reduction efficiency of intermediate results through intratile local reduction and intertile global reduction.

This article makes the following contributions.

1) We develop the first ReRAM-based hypergraph accelerator, which is composed of a hybrid analog and digital PIM architecture to support hypergraph processing.
2) We propose the software-level hypergraph partition and scheduling co-designs to maximize the hardware efficiency of the underlying architecture.
3) PhGraph outperforms state-of-the-art hypergraph solutions by up to 4,309.81× (CPU), 547.13× (FPGA), and 166.76× (ASIC) in terms of performance, and 36 416.11× (CPU), 924.12× (FPGA), and 41.44× (ASIC) in terms of energy-savings, respectively.

The remainder of this article is organized as follows. Section II introduces the background and motivation. Section III describes the PhGraph architecture and the detailed designs. Section IV discusses the experimental results. Section V reviews the related works, and finally, Section VI concludes this article.

II. Background and Motivation

In this section, we first review the background for hypergraph processing and ReRAM basics. We then analyze the limitations of existing hypergraph processing solutions, finally motivating our approach.

A. Hypergraph Processing

Hypergraph: A hypergraph is composed of a set of vertices V and hyperedges H, denoted as $G =< V, H >$. A hyperedge can connect an arbitrary number of vertices. The number of vertices (hyperedges) in hypergraph G is donated as $|V|$ ($|H|$). We use $N(h)$ to indicate a set of vertices connected to the hyperedge h, and the degree of h is the size of $N(h)$. Two hyperedges h_i and h_j can be called *overlap* when they share some common vertices (i.e., $N(h_i) \cap N(h_j) \neq \phi$). Fig. 1(a) shows a hypergraph G with seven vertices and four hyperedges. The hyperedge h_0 connects v_0, v_1, v_2, and v_3, meaning $N(h_0) = \{v_0, v_1, v_2, v_3\}$. The hyperedges h_0 and h_2 is overlapped since $N(h_0) \cap N(h_2) = \{v_0, v_1, v_2\}$.

A hypergraph is typically represented in the *bipartite representation* format [2], [7], [8], which represents each hyperedge as a distinct vertex and connects the vertex with its incident vertices. Thus, as shown in Fig. 1(c), a hypergraph can be represented as a bipartite graph and stored with an extended CSR format [7]. In addition, a hypergraph can also be represented as an incidence matrix, as shown in Fig. 1(b), where rows and columns represent vertices and hyperedges, respectively. The (i, j)th nonzero element in matrix M means that the vertex i is associated with the hyperedge j.

Hypergraph Processing: Algorithm 1 shows the hypergraph processing procedure of the PageRank algorithm, performing *hyperedge computation* kernel and *vertex computation* kernel alternately in each iteration [2]. First, two algorithm-specific update functions `V_update_H` and `H_update_V` are defined (lines 1–4) for hyperedge computation and vertex computation, respectively. An `apply_V` function is also defined (lines 5 and 6) for an apply operation on each vertex after vertex computation. Similarly, there is an `apply_H` function for applying on hyperedges. The procedure first initializes the data value (*v_value* and *h_value*) and the active vertices and hyperedges (*FrontierV* and *FrontierH*) (lines 7–10). Then, the iterative computation starts (lines 10–23). For the hyperedge computation kernel, all bipartite edges associated with *FrontierV* are processed with `V_update_H` (lines 12 and 13). Once a hyperedge value is changed, this hyperedge will be activated and added into *FrontierH* (lines 14 and 15). Symmetric to the hyperedge computation, the vertex computation iterates all bipartite edges in *FrontierH* to perform the `H_update_V` operation and further generate a new *FrontierV* (lines 16–19). Finally, the `apply_V` function is applied on each active vertex. The hypergraph processing ends with no active data left or a maximum iteration count MAX_ITER reached.

Algorithm 1 PR Algorithm for Hypergraph

```
Input: Hypergraph $G =< V, H >$
Output: $v\_value$ and $h\_value$
1:  function V_UPDATE_H($v$, $h$)                         ▷ updating hyperedge
2:      $h\_value[h] \leftarrow h\_value[h] + \frac{v\_value[v]}{v.\texttt{getOutDeg}()}$
3:  function H_UPDATE_V($h$, $v$)                         ▷ updating vertex
4:      $v\_value[v] \leftarrow v\_value[v] + \frac{h\_value[h]}{h.\texttt{getOutDeg}()}$
5:  function APPLY_V($v$)                                 ▷ apply vertex
6:      $v\_value[v] \leftarrow \alpha \times v\_value[v] + \frac{1-\alpha}{|V|}$
7:  VertexInit($v\_value$)
8:  HyperedgeInit($h\_value$)
9:  $FrontierV$.init()
10: $FrontierH$.init()
11: for $iter \leftarrow 0$, MAX_ITER do
                                                          ▷ Hyperedge Computation
12:     for each $(v, h) \in G$ where $v \in FrontierV$ do
13:         V_UPDATE_H($v$, $h$)
14:         if $h$ is updated then
15:             $FrontierH.push(h)$
                                                          ▷ Vertex Computation
16:     for each $(h, v) \in G$ where $h \in FrontierH$ do
17:         H_UPDATE_V($h$, $v$)
18:         if $v$ is updated then
19:             $FrontierV.push(v)$
20:     for each $v \in FrontierV$ do
21:         APPLY_V($v$)
22:     if $FrontierV$ and $FrontierH$ are Empty then
23:         break
```

B. Crossbar-Based PIM Architectures

With in-situ processing, high parallelism, and low-energy consumption, ReRAM-based PIM architecture has been widely used to accelerate memory-bound applications. Typically, ReRAM cells are often organized as a crossbar architecture, which can be built in two forms: 1) analog crossbar-based PIM (APIM) and 2) digital crossbar-based PIM (DPIM). Fig. 2 shows their crossbar basics, respectively.

APIM Crossbar: Fig. 2(a) shows an example APIM crossbar. First, matrix data is mapped into the ReRAM crossbar in the form of conductance G, which is the reciprocal quantity of resistance. With digital-to-analog converters (DACs), the input vector is converted to voltage signals V, which will be further applied on word lines. According to Kirchhoff's law, current signals on bit lines can be calculated as $I = V \times G$. By converting currents into digital numbers via analog-to-digital converters (ADCs), the output vector stores the multiplication result of the preloaded matrix and the input vector. APIM crossbar can natively perform MVM operations in the $O(1)$ time complexity, assuming that the matrix can fit into the crossbar. In contrast, traditional architectures, such as CPUs, which leverage arithmetic or bit operations as basic operators, perform MVM in the $O(n^2)$ complexity with a large amount of data movement between memory and processing units. For applications based on MVM operations, such as hypergraph processing, the crossbar-based APIM architecture can dramatically improve the overall performance.

DPIM Crossbar: Different from APIM, which only utilizes the read feature of ReRAM cells, DPIM further takes advantage of the write feature of ReRAM cells, whose

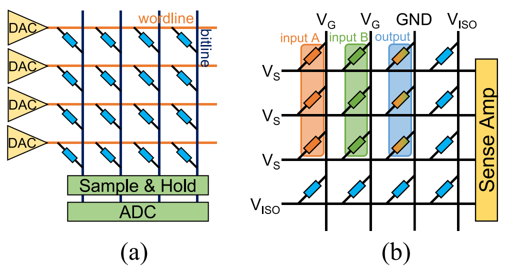


Fig. 2. ReRAM-based crossbar structures. (a) APIM crossbar. (b) DPIM crossbar.

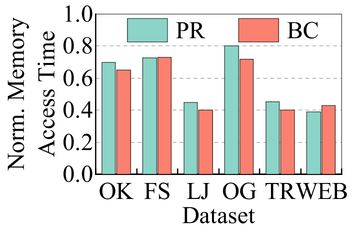


Fig. 3. Off-chip memory access time of XuLin normalized to total execution time for algorithms PR and BC.

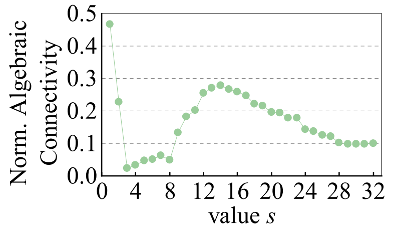


Fig. 4. Normalized algebraic connectivity of largest connected component of s-line graph for network com-DBLP [16] with various s values.

resistance can be modified by applying electrical current. In theory, when the current flowing into the ReRAM cell reaches a certain threshold, the resistance increases. When the current flowing out of the ReRAM cell reaches the threshold, the resistance decreases [17]. Following the above principles, DPIM can perform parallel bitwise NOR operations in practice [18].

Fig. 2(b) depicts the DPIM crossbar with the PIM capability of performing bitwise NOR operations. Two input data A and B are written into ReRAM cells with a single bit per cell pattern, where a high resistance (10 MΩ) represents the bit "0" while a low resistance (10 KΩ) denotes the bit "1," in the first and second columns, respectively. In the third column, the output result is initialized as 1 (i.e., low resistance). Then, the isolation voltage V_{ISO} is applied to unused rows and columns to isolate these cells. Meanwhile, the specific voltages V_G and GND are applied to the bit lines corresponding to the input and output columns, and V_s is used to select active word lines for computations. If an input cell in the activated word line is programmed to be a low-resistance state (i.e., 1), the high-current flows into the output cell whose resistance thus increases from the low to the high state (i.e., 1 $\rightarrow$ 0), based on Ohm's law. Similarly, if both input cells in the activated word line are programmed to be in the high-resistance state (i.e., 0), the current will be limited, and the resistance of the output cell remains unchanged (i.e., 1). As a result, the output data can be viewed as the formula A NOR B. Previous studies [19], [20] have demonstrated that almost all arithmetic operations, such as addition and multiplication, can be realized as a series of NOR operations. Consequently, the DPIM crossbar can support a wide range of applications and achieve impressive performance with fine-grained parallelism.

APIM Versus DPIM: Both APIM and DPIM organize ReRAM cells in a crossbar architecture. However, they would realize two categories of base operators with distinct parallelism by architecting different peripheral circuits and applying different voltages. APIM is endowed with coarse-grained array-level parallelism, while DPIM is endowed with fine-grained row-level parallelism. Thus, APIM may achieve better performance than DPIM if crossbar cells can be fully utilized. Otherwise, DPIM would be superior since its fine-grained parallelism can avoid matrix-grained ineffectual computations.

C. Existing Efforts

1) Inefficiencies of Existing Hypergraph Solutions: Recently, many hypergraph processing frameworks developed on CPUs and ASICs have been proposed to improve parallelism [2], data locality [7], programming productivity [8], and communication overheads [1]. However, these earlier solutions are still inadequate in addressing the memory bottleneck arising in hypergraph processing. To demonstrate this, we conduct a set of experiments to count the normalized execution time of off-chip memory accesses for hypergraph processing over the state-of-the-art hypergraph accelerator XuLin [8].

Fig. 3 shows the normalized execution time arising from off-chip memory accesses by benchmarking PageRank (PR) and betweenness centrality (BC) on six real-world hypergraphs (i.e., com-Orkut (OK), Friendster (FS), LiveJournal (LJ), Orkut-Group (OG), Gottron-Trec (TR), and Web-trackers (WEB)). The benchmark and dataset details can be found in Section IV-A. We can see that off-chip memory accesses take about half of the total running time on average. For PR on the large hypergraph OG, this ratio can be as high as 80.19%, demonstrating that off-chip memory accesses still bottleneck hypergraph processing. The reason behind this is apparent. Although XuLin [8] has minimized the redundant off-chip memory accesses, there still exists a substantial amount of necessary data movement between processing elements and the memory under the traditional von Neumann architecture.

2) Limitations of ReRAM-Based Graph Accelerators: Previous studies [9], [21] have demonstrated that PIM is promising in tackling the memory bottleneck arising in the traditional von Neumann architecture. The recent breakthrough achieved by ReRAM-based graph processing acceleration [9], [10] further motivates us to accelerate hypergraph processing using ReRAM-based architecture. Considering a graph is a special case of a hypergraph, an intuition for realizing ReRAM-based hypergraph processing acceleration is to directly reuse existing ReRAM-based graph accelerator to support the *hyperedge computation* and *vertex computation* kernels alternatively. However, existing accelerators are not well suited to hypergraph processing due to the following two reasons.

1) *Interkernel Difference:* Conventional graph processing updates the vertex state using the same update function. In contrast, the hyperedge computation and vertex computation kernels perform different update logic to alternatively update the states of hyperedges and vertices in each iteration. Consequently, in order to handle hypergraph processing, existing accelerators have to remap kernel-specific data into the crossbar to implement kernel switching. It is clear that this will incur superfluous resistance writes with significant performance degradation. Taking the k-core algorithm running on OG as an example, it performs up to 2923 iterations, resulting in 5846 (2923×2) kernel switches. The resistance writing overhead arising from kernel switching is 13.89 times more than the kernel execution time, occupying 93.28% of the overall time. Even worse, frequent kernel switching will increase cell wear and reduce the lifetime of ReRAM cells.
2) *Overlap-Induced Irregularity*: Existing ReRAM-based graph accelerators enjoy the massive parallelism of the matrix-structured crossbar architecture, with an assumption that the crossbar cells can be fully utilized. However, real-world hypergraphs have a unique overlap structure, where many vertices (hyperedges) can be shared by at least two hyperedges (vertices), which induces an extremely sparse distribution of crossbar. Thus, using existing ReRAM-based graph accelerators to handle hypergraph applications will suffer significant performance degradation arising from ineffectual zero-valued data mapping. Although graph reordering [10], [15] techniques can be used to improve the sparsity-induced inefficiencies, these graph-oriented solutions are inadequate for hypergraph processing due to the intertwined relationships between hyperedges and vertices.

D. Overcoming Inefficiencies

In this work, we have the following two observations arising in hypergraph processing.

Observation 1: The hyperedge and vertex computation kernels in hypergraph processing exhibit an operational similarity, indicating that these two kernels can be uniformly formalized as a set of MVM operations, where only the input matrices are transposed to each other.

Algorithm 1 performs hypergraph processing from the perspective of hypergraph topology. In fact, this procedure can be expressed as the MVM-formatted computing paradigm in theory. Taking the PR algorithm as an example, the hyperedge computation and vertex computation kernels can be formalized into an MVM form as follows:

$$\boldsymbol{H}_i = M^{\mathrm{T}} \times (\boldsymbol{V}_i / V.degrees) \quad (1)$$

$$\boldsymbol{V}_{i+1} = \alpha \times (M \times (\boldsymbol{H}_i / H.degrees)) + (1-\alpha)/n_v \quad (2)$$

where M, V, and H denote the hypergraph-induced incidence matrix, the vertex property vector, and the hyperedge property vector, respectively. M^{T} indicates the transposed matrix of M. α is a damping factor, which is an algorithm-specific parameter. We can see that these two symmetric kernels perform similar MVM operations, except that their input matrices are transposed with each other (i.e., M^{T} and M). This observation provides an opportunity for us to use a uniform MVM kernel to eliminate the interkernel difference and further avoid costly repeated data mapping. We would like to note that almost all hypergraph algorithms can also be equivalently expressed as a set of MVM operations.

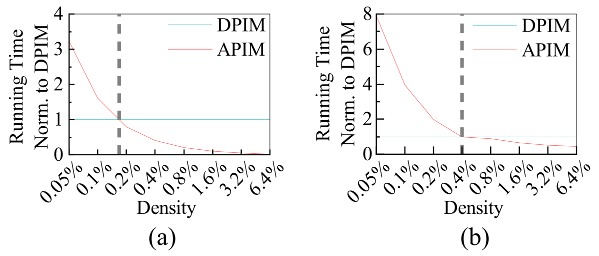


Fig. 5. Performance of APIM and DPIM for PR and BC algorithms on hypergraphs with various density. (a) PR. (b) BC.

Observation 2: The hypergraph-induced incidence matrix exhibits global sparsity and local denseness, which is closely related to the hypergraph overlap structure.

The hypergraph-induced incidence matrix is sparse in a global view since $[|B|/(|V| \times |H|)] \ll 1$, where $|B|$, $|V|$, and $|H|$ denote the number of bipartite edges, vertices, and hyperedges, respectively. However, the incidence matrix can still contain dense parts. Let us consider the example hypergraph in Fig. 1(a). Its incidence matrix is sparse ($[15/(7 \times 4)] \ll 1$) [see Fig. 1(b)], but the 3×3 submatrix in the upper-left corner of this incidence matrix can be dense ($[9/(3 \times 3)] = 1$). This phenomenon is caused by the hypergraph overlap feature, where some vertices tend to be shared by multiple hyperedges. In Fig. 1(a), we observe that h_0, h_1, and h_2 share the common vertices v_0, v_1, and v_2, resulting in the 3×3 dense submatrix in Fig. 1(b). This phenomenon will be particularly true for real-world hypergraphs.

To further demonstrate Observation 2, we construct the s-line graph [22] of a hypergraph by abstracting hyperedges as new vertices and creating new edges for representing specified overlap relationships where the common vertex count is greater than or equal to s. The s-line graph retains the critical topological structure features of the original hypergraph while removing unimportant overlap relationships. Fig. 4 illustrates the normalized algebraic connectivity of the s-line graph with different s over a real-world hypergraph *com-DBLP* [16]. The initial descent of the curve indicates that the hypergraph is sparse in a global view. The subsequent ascent evidences the presence of local denseness, while the final descent indicates that the scale of local denseness is limited. This observation inspires us to overcome the overlap-induced irregularity through a *split and conquer* hypergraph partitioning strategy, which polarizes the incidence matrix into either dense or sparse submatrices according to the hypergraph overlap structure, yielding two types of workloads with enhanced regularity.

Nevertheless, on the hardware level, we observe that there is no absolute winner between APIM and DPIM technologies for hypergraph processing. To demonstrate this, we evaluate the execution time of APIM- and DPIM-based hypergraph

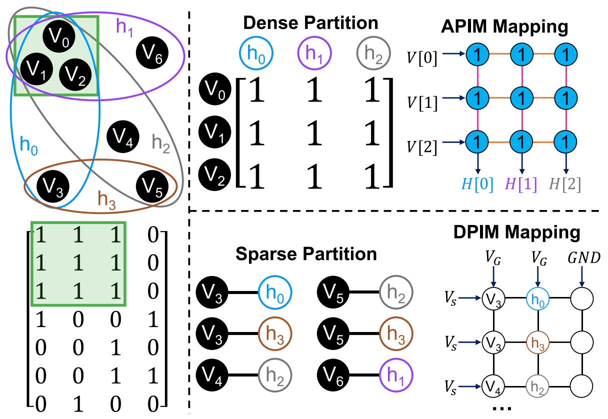


Fig. 6. Example for hybrid hypergraph mapping with example hypergraph and its incidence matrix, polarizing hypergraph into dense and sparse partitions, and mapping into APIM and DPIM respectively.

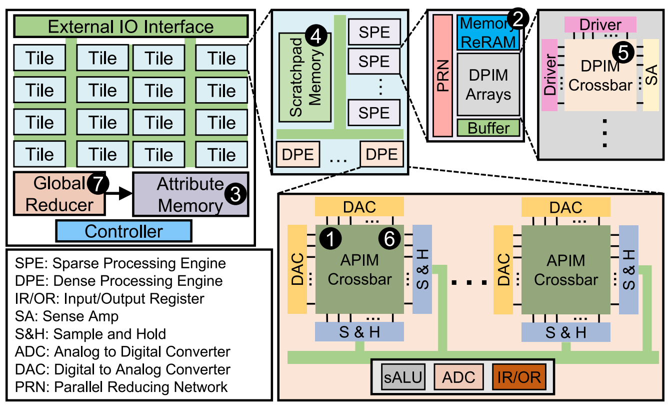


Fig. 7. PhGraph architecture.

processing for an all-active algorithm PR and a non-all-active algorithm BC using the generation hypergraph with varying density. Fig. 5 shows the results, indicating that APIM runs faster than DPIM when the hypergraph density is larger than a threshold (i.e., 0.18% for PR and 0.4% for BC). In contrast, when the density is less than this threshold, DPIM outperforms APIM. Consequently, we are motivated to reap the best of both worlds of APIM and DPIM with workload polarization to support efficient hypergraph processing.

Fig. 6 illustrates how the example hypergraph is mapped onto a hybrid APIM-DPIM architecture for computation. First, we polarize the hypergraph and further divide it into dense and sparse partitions, in which the dense ones are represented in matrix format, and the sparse ones are stored as the bipartite edge list. Second, dense submatrices are written into the APIM crossbar for computation, while each bipartite edge is mapped into a row of the DPIM crossbar for computation. Finally, the intermediate results from both APIM and DPIM will be reduced to obtain the final results.

However, materializing a hybrid APIM-DPIM accelerator for efficient hypergraph processing remains challenging. First, the APIM crossbar cannot be used directly to support both MVM and M^TVM operations without matrix remapping physically. Second, the aforementioned density thresholds vary from algorithm to algorithm, making hypergraph polarization complex. Third, substantial intermediate results arising from two isolated PIMs incur expensive reduction overheads.

III. PhGraph

This section first introduces the overall architecture of PhGraph and its workflow and then elaborates on the hardware- and software-level design in detail.

A. Architecture

Fig. 7 depicts the overall architecture of PhGraph. At a high level, PhGraph adopts a hierarchical architecture. A PhGraph chip consists of several Tiles, which are connected through an on-chip mesh interconnect network. Each tile comprises two types of processing engines: 1) dense processing engines (DPEs) and 2) sparse processing engines (SPEs). The former incorporates APIM arrays to process dense matrix-granularity workloads. To prevent the repeated loading of the incidence matrix and its transposed matrix into the crossbar, each APIM crossbar features a transposed crossbar design (discussed in Section III-C). The latter is responsible for processing sparse bipartite-edge-granularity workloads using DPIM arrays. The input bipartite edges are temporarily stored in the *memory ReRAM* to overlap data fetching and processing partially, thereby reducing the latency of data mapping to DPIM arrays. Since different bipartite edges executing in parallel may have common hyperedges or vertices, the output results of DPIM arrays need to be passed to the parallel reducing network (PRN), where several intermediate values of the same vertex or hyperedge are finally reduced into a final result. The *Buffer* is responsible for caching the final results. The vertex and hyperedge property values are initially stored in the *Attribute Memory*. At runtime, these property values are distributed to different tiles and cached in the *Scratchpad Memory*. They are then sent to the respective crossbars for computation. At the end of each iteration, the global reducer (GR) reduces and synchronizes property values among different tiles. The *Controller* is responsible for managing data interactions, including data fetching and communication among different components.

B. Workflow

The workflow of running a hypergraph application with PhGraph consists of several steps. Initially, a hypergraph needs to be polarized into dense and sparse partitions. The dense partitions are stored in a dense matrix format, which can be mapped into the APIM crossbars offline in advance (❶). The sparse partitions are stored in the *Memory ReRAM* in a bipartite edge list format (❷). Meanwhile, vertex and hyperedge property values are initialized and loaded into the *Attribute Memory* (❸). Then, PhGraph starts to perform hyperedge and vertex computation kernels alternately in each iteration. Taking the hyperedge computation kernel as an example, where active vertices update their incident hyperedges using an algorithm-specific update function, the property values are distributed to all tiles and stored in the *Scratchpad Memory* (❹). Afterward, both DPEs and SPEs start working simultaneously. In SPEs, bipartite edges with active vertices are mapped to each row of DPIM crossbars to perform algorithm-specific operations

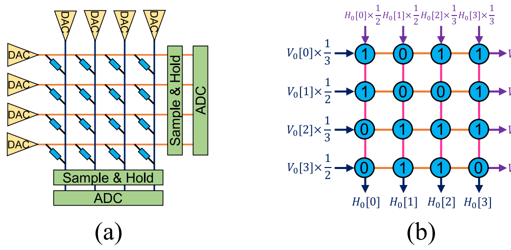

DAC
DAC
DAC
DAC
DAC
DAC
DAC
DAC
Sample & Hold
ADC
Sample & Hold
ADC
(a)
$H_0[0]\times\frac{1}{2}$ $H_0[1]\times\frac{1}{2}$ $H_0[2]\times\frac{1}{3}$ $H_0[3]\times\frac{1}{3}$
$V_0[0]\times\frac{1}{3}$
$V_0[1]\times\frac{1}{2}$
$V_0[2]\times\frac{1}{3}$
$V_0[3]\times\frac{1}{2}$
1 0 1 1
1 0 0 1
0 1 1 1
0 1 1 0
$V_1[0]'$
$V_1[1]'$
$V_1[2]'$
$V_1[3]'$
$H_0[0]$ $H_0[1]$ $H_0[2]$ $H_0[3]$
(b)

Fig. 8. (a) Transposed memory crossbar. (b) Single bit implementation

in parallel with a series of bitwise NOR operations (❺). The computation results are passed to PRN to reduce intermediate values into the final results, which will be temporarily stored in the *Buffer*. In DPEs, APIM crossbars are driven to perform MVM operations (❻). The output results are reduced by peripheral simple algorithmic and logic units (sALUs), and reduced results are stored in the output registers (OR). Finally, these newly computed results in the local buffer of different tiles are synchronized through the GR (❼).

C. Hardware Designs

Transposed Crossbar Architecture: As depicted in Fig. 2(a), performing an MVM operation with a conventional APIM crossbar involves rowwise input and columnwise output. However, hypergraph processing requires iterative hyperedge and vertex computation kernels, and their input matrices are transposed with each other. In the conventional crossbar context, the incidence matrix and its transposed one have to be loaded into the crossbar alternately and iteratively, along with two computation kernels. This inevitably introduces superfluous resistance writes and analog-signal conversions, leading to additional performance and energy overheads.

By attaching simple peripheral circuits to the conventional APIM crossbar as shown in Fig. 8(a), a lightweight transposed memory crossbar architecture can be built to cope with the aforementioned issue [23]. This architecture enables voltages to be applied on both vertical and horizontal directions, and the input voltages are coupled with the corresponding output currents in the orthogonal direction. Only a single set of orthogonal directions can be utilized during processing. The transposed crossbar can perform the MVM and M^TVM operation alternately by applying the input vector as voltages on the rows and columns of the crossbar, respectively.

In hypergraph processing, the incidence matrix can be mapped into the crossbar in advance, and the hyperedge computation kernel can be performed by inputting vertex values to row DACs and driving the crossbar in a row-input-column-output pattern. Intermediate hyperedge results can be obtained from column ADCs. Conversely, the vertex computation kernel can be performed in a column-input-row-output pattern without modifying the data stored in the crossbar. By this means, it circumvents the need for repeatedly loading both the matrix and its transposed one into the crossbar. The transposed crossbar architecture can introduce extra peripheral circuits (i.e., double number of DACs and ADCs), leading to power and area overheads. Fortunately, the row ADCs and column ADCs are never utilized simultaneously in practice, so that the ADCs are enabled to be shared between rows and columns of the transposed crossbar in a time-division manner using a simple multiplexer. This approach eliminates the need for extra ADCs, which typically occupy significant area and power in APIM designs, reducing the peripheral circuits overhead of the transposed crossbar architecture.

Hierarchical Reduction Design (HRD): As discussed in Section III-B, a significant number of intermediate results may be generated and stored in various buffers located in DPEs, SPEs, and tiles during hypergraph processing. At the end of each iteration, these intermediate results need to be reduced to obtain the final results, leading to costly reduction overhead and significant performance degradation. To address this issue, we present a two-level HRD, which contains both local and global reduction components, performing intratile and intertile reduction, respectively. Furthermore, this enables two-stage pipeline parallelism, allowing intermediate value reduction effectively and efficiently.

Considering distinct characteristics of intermediate results in various components, we exploit different specialized hardware designs to maximize the reduction efficiency. For DPEs, each row (column) of APIM crossbars produces intermediate results for a fixed vertex (hyperedge). Thus, the reduction can be directly performed by sALUs due to the regular and serial output format. In contrast, the output data distribution of DPIM crossbars in SPEs is more complex. DPIM performs rowwise parallel computation, and each row can process any bipartite edges, producing intermediate results with the $\langle id, value\rangle$ format, where multiple *values* with the same *id* need to be reduced. Fortunately, by assigning bipartite edge tasks in the vertex (hyperedge) index order, we can ensure that *ids* in the output sequence are ordered. Therefore, we can use a prefix sum network [24] as PRN to achieve parallel reduction. For tiles, intermediate results mix both features of DPEs and SPEs, leading to complex data distribution within each tile. However, benefiting from the coarse-grained task distribution among tiles, employing a binary reduction network as GR is sufficient to reduce intermediate results from tiles.

D. Software Designs

We propose three software co-designs to exploit the underlying hardware fully. In particular, PhGraph employs an overlap-aware hypergraph partitioning technique to polarize hypergraph tasks into dense and sparse workloads. In addition, PhGraph also customizes the partition scheduling strategy to keep the load balance between DPEs and SPEs. Finally, PhGraph improves the overall utilization of APIM crossbars with the single-bit fine-grained algorithm mapping.

Overlap-Aware Hypergraph Partitioning: As discussed in Section II-D, hypergraphs exhibit complex topology, where vertices and hyperedges are intertwined with each other, making it difficult and time-consuming to perform hypergraph polarization. Existing partitioning strategy [25], [26], [27] designed for graphs is less efficient for hypergraphs, due to the complex hypergraph topology, where vertices and hyperedges are intertwined with each other. Fortunately, the distribution of dense and sparse partitions in a hypergraph is closely

Algorithm 2 Overlap-Aware Hypergraph Partitioning

```
Input: Hypergraph $G = (V, H)$
Input: Overlap argument $s$ and Threshold $D$
Output: Dense partition $DP$ and Sparse partition $SP$
1:  procedure HYPERGRAPH_PARTITION
2:      $s\_line\_graph\_h \leftarrow$ GEN_S_LINE_GRAPH($G, s$)
3:      $s\_line\_graph\_v \leftarrow$ GEN_S_LINE_GRAPH($G^T, s$)
4:      $pre\_DP \leftarrow$ CONSTRUCT_HYPERGRAPH(
          hyperedges: vertices in $s\_line\_graph\_h$,
          vertices: vertices in $s\_line\_graph\_v$)
                                  ▷ specify left bipartite edges as sparse
5:      $SP.insert(G - pre\_DP)$
6:      for each $p \in$ GRID_PARTITION($pre\_DP$, $8 \times 8$) do
7:          if $Density(p) > D$ then
8:              $DP.insert(p)$                    ▷ $p$ is dense
9:          else
10:             $SP.insert(p)$                    ▷ $p$ is sparse
11:     return $DP$ and $SP$
```

related to the overlap feature, and the s-line graph of a hypergraph can reflect the most critical overlap relationships in the hypergraph. Taking advantage of this fact, we propose an overlap-aware partitioning technique, which follows a *split and conquer* ideology, to make a tradeoff between partition quality and efficiency, as depicted in Algorithm 2. First, we construct s-line graphs of the hypergraph for hyperedges and vertices, respectively (lines 2 and 3). These s-line graphs record the heaviest overlapped hyperedges (vertices), which share at least s common vertices (hyperedges) with each other. By selecting an appropriate value s, we can obtain a suitable number of hyperedges and vertices to build a preliminary dense subhypergraph (line 4). The remaining hyperedges and vertices are classified as sparse workloads (line 5). For the preliminary dense partition, we further employ a grid partition strategy [25] to divide the subhypergraph-induced incidence matrix into multiple submatrix blocks. If the density of a submatrix block is higher than a specified threshold D, it can be regarded as a dense workload. Otherwise, it is a sparse workload (lines 6–10).

The arguments s and D can jointly affect the efficiency and effectiveness of partitioning and further influence the overall performance of PhGraph. We first discuss the impact of the value s. On the one hand, selecting an appropriate s is critical for partitioning efficiency. For a specific hypergraph, constructing an s-line graph with a small s value can be more time-consuming than a larger s value. On the other hand, the number of dense partitions is determined by the value of s along with the hypergraph scale. A small s indicates producing a large s-line graph, which further generates many dense partitions. Making a tradeoff between these factors allows for a heuristic estimation of s, as follows:

$$s = 2^{\log_{10}(|V| \times |H|)}/64 \tag{3}$$

where $|V|$ and $|H|$ are the number of vertices and hyperedges, respectively. We take a multiplication between them to reflect the hypergraph scale.

We determine the value of D based on the global density of the hypergraph, with a larger threshold chosen for denser hypergraphs to achieve load balance between DPEs and SPEs. Additionally, the hypergraph algorithm characteristic can also affect the choice of D, as discussed in Section II-D. In practice, we obtain the value of D as follows:

$$D = \alpha \times \sqrt{GD} \tag{4}$$

where GD represents the global density of a hypergraph. α indicates whether the hypergraph algorithm is all-active (non-all-active), represented by the empirical number 1 (10).

Load Balanced Partition Scheduling: After the hypergraph is partitioned, all the partitions will be dispatched to tiles, and further to APIM and DPIM crossbars, for computations. One intuition solution is to apply a vertex-major scheduling strategy, which distributes all associated hyperedges to a specific crossbar according to the vertex. However, this can lead to load imbalance between crossbars due to the uneven vertex degrees of polarized dense or sparse workloads. Different parallelism of DPEs and SPEs can further exacerbate the load imbalance issue. Further, the vertex-major scheduling can lead to the under-utilization of data locality, especially for DPEs that preload all data from APIM crossbars. Symmetrically, a hyperedge-major scheduling strategy can raise similar issues in the orthogonal direction.

To address this issue, we propose a hybrid partition scheduling strategy, ensuring that each APIM (DPIM) crossbar contains a similar number of partitions (bipartite edges) and the numbers of vertices and hyperedges are similar for each crossbar. This strategy takes the limited local buffer size into account, ensuring load balance and maximizing data reuse.

Algorithm Mapping: The algorithm mapping on DPIM is straightforward due to the fine-grained parallelism. After partitioning the hypergraph into sparse and dense partitions, sparse partitions are organized as a bipartite edge list. Each bipartite edge will be mapped to a row of DPIM crossbars for parallel processing. Considering the hardware design requirements discussed in Section III-C, we map sparse bipartite edges to DPIM rows in the order of hyperedge (vertex) indices in the hyperedge (vertex) computation kernel, enabling SPEs to process sparse partitions efficiently.

In contrast, the algorithm mapping of dense partitions on APIM crossbars requires careful considerations, including data construction and mapping granularity. First, mapping which data into the crossbar is essential for algorithm mapping. Equation (1) as an example, the matrix $M^{\mathrm{T}}/\boldsymbol{V}.degrees$ is mapped into the crossbar in conventional designs, and $\boldsymbol{V}$ is used as input signals for computation. However, this approach is unsuitable for hypergraph processing due to the different coefficients of matrices for the hyperedge and vertex computation kernels. To resolve this issue, we only map the hypergraph topology to APIM crossbars with a single-bit pattern, as shown in Fig. 8(b). The bit 1 (0) indicates the (non-)existence of the connection between the hyperedge and vertex, and different coefficients are applied through peripheral circuits.

The mapping granularity is another crucial factor in AM on APIM crossbars. For the fine-grained mapping, if the crossbar is divided with small blocks (such as 2×2), the high

TABLE I
PhGraph Configurations

Component	Params.	Spec.	Power (mW)	Area (mm^2)
DPE Properties				
Crossbar	bits per cell	1 bit	1.2	0.0001
	size	128×128		
	number	8		
ADC	resolution	8 bit	16	0.0096
	number	8		
DAC	resolution	1 bit	8	0.00034
	number	8×128×2		
S&H	number	8×128×2	0.002	0.00008
sALU	number	8	6.512	0.0432
IR+OR	size	8KB+1KB	3.52	0.0081
SPE Properties				
Crossbar	bits per cell	1 bit	48	0.0276
	size	1024×1024		
	number	8		
Driver	number	8×1024×2	16	0.0007
SA	number	8×1024	0.003	0.0001
Buffer	size	64KB	17.66	0.0086
PRN	number	8	0.157	0.0341
Tile Properties (8 DPEs + 8 SPEs per Tile)				
DPE Total	number	8	281.872	0.4914
SPE Total	number	8	654.56	0.5688
Scratchpad Memory	size	1MB	165.6	0.664
PhGraph Accelerator Properties (16 Tiles)				
Tile Total	number	16	17632.5	27.587
Global Reducer	-	-	44.67	2.69
Total	-	-	17.677 W	30.3 mm^2

TABLE II
Read-World Hypergraph Datasets

Datasets	$\|V\|$	$\|H\|$	$\|B\|$	Type
com-Orkut(OK) [16]	2.32M	15.30M	107.08M	Social
Friendster(FS) [16]	7.94M	1.62M	23.48M	
LiveJournal(LJ) [28]	3.20M	7.49M	112.31M	
Orkut-Group(OG) [28]	2.78M	8.73M	327.04M	
Gottron-Trec(TR) [28]	0.55M	1.17M	83.63M	Text
Web-trackers(WEB) [28]	27.67M	12.76M	140.61M	Web

parallelism of APIM can hardly be exploited fully, resulting in low performance. In contrast, if the mapping is coarse-grained, many cells in the crossbar will have a value of 0, resulting in useless computation with limited efficiency. To be specific to hypergraph processing under the PhGraph architecture, since most overlapping occurs only on a few hyperedges and vertices, hypergraph partitions usually correspond to small-sized dense submatrix blocks, e.g., 8×8. However, APIM crossbars are typically sized of more than 128×128 with fewer peripheral circuits that can be shared to reduce overall area overhead [14]. To address this mismatch, we restrict the dense submatrix size to 8×8 and flatten multiple dense submatrices to fill a crossbar sized of 128×128 or more. This yields a nice sweet spot in three dimensions: 1) crossbar utilization; 2) execution efficiency; and 3) hardware cost.

IV. Evaluation

A. Experimental Setup

PhGraph Settings: We integrate a modified NeuroSim [29] with the cycle-accurate simulator ZSIM [30] to simulate the functionalities of PhGraph. We use the VTEAM [31] memristor model for DPIM design and refer to the APIM design as in [10]. Table I summarizes the PhGraph configurations. PhGraph is set with 16 tiles, each containing 8 DPEs and 8 SPEs. Each DPE consists of 8 crossbars sized 128×128 and configured as APIM arrays with the read and write latencies being 29.31ns and 50.88ns, respectively. Their read and write energy consumptions are 1.08pJ and 3.91nJ. Each SPE has 8 DPIM crossbars sized 1024×1024. As for the specific parameters of the DPIMs, we refer to the latest works [18], [32], [33] and set the state transition latency as 1ns. Furthermore, we widely research related works [20], [34] to determine the number of cycles required for various operators. The *Attribute Memory* is sized of 1GB, and the size of each *Memory ReRAM* is configured as 32 MB. Each *Buffer* has a size of 64 KB. We use CACTI 6.5 [35] to model the memories and buffers and estimate their area, power, and latency.

Datasets and Algorithms: Table II gives the six most widely used real-world hypergraphs from various domains, which exhibit varying scales and overall sparsity distribution. PhGraph is evaluated with five representative hypergraph algorithms, including betweenness centrality (BC), breadth first search (BFS), connected components (CC), *k*-core Decomposition (*k*-core), and PageRank (PR).

Baselines: We compare PhGraph with two state-of-the-art CPU-based hypergraph systems Hygra [2] and NWHy [36], an FPGA-based hypergraph accelerator XuLin-F [8], and two ASIC-based hypergraph accelerators ChGraph [7] and XuLin [8]. Both Hygra and NWHy run on a machine configured with two Intel Xeon Gold 6338 CPUs equipped with 1TB DDR4 memory, and XuLin-F is evaluated on a Xilinx Alveo U250 FPGA accelerator card running at 280 MHz. Since the open-source system NWHy only implements BFS and CC algorithms, and XuLin does not support the BFS algorithm, we only compare PhGraph against these baselines on their supported algorithms. Note that all performance and energy results are obtained by accounting for both computations and communications.

B. Overall Results

We compare PhGraph with baselines in terms of performance and energy savings, and analyze its power and area.

Performance: Fig. 9 depicts the total running time of PhGraph against state-of-the-art solutions, including Hygra (CPU-based), NWHy (CPU-based), XuLin-F (FPGA-based), ChGraph (ASIC-based), and XuLin (ASIC-based).

PhGraph Versus Hygra and NWHy: Overall, PhGraph outperforms Hygra and NWHy by 211.70× and 352.44× on average, respectively. The reasons are twofold. First, the highly parallel in-situ processing pattern adopted in PhGraph eliminates the massive off-chip memory accesses. Second, PhGraph employs specialized hardware designs, improving pipeline efficiency and avoiding instruction control overheads for CPUs.

Specifically, TR exhibits minimal performance improvement with 61.04× on average. This is because TR has the smallest scale among all six datasets, with 0.55M vertices and 1.17M hyperedges. Assuming that each vertex and hyperedge contains a 4-byte attribute value, the total size

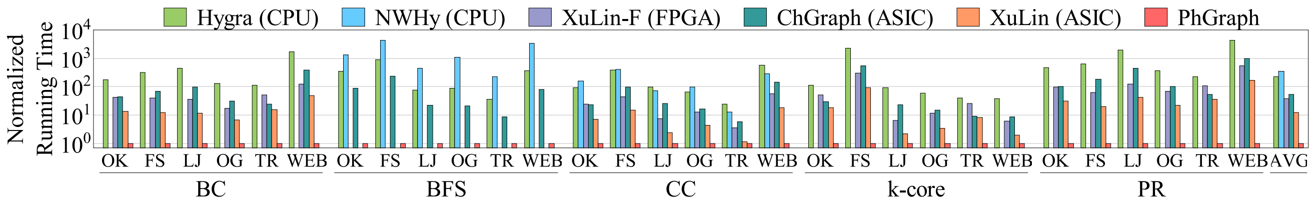


Fig. 9. Running time of PhGraph against Hygra, NWHy, XuLin-F, ChGraph, and XuLin (normalized to PhGraph).

of attribute values is much smaller than the LLC size of CPUs (i.e., (0.55M+1.17M) $\times$ 4 B $\ll$ 1.5 MB $\times$ 32 $\times$ 2). Thus, all attribute values can be cached in LLC until the algorithm execution is complete, implying fewer off-chip memory accesses that can be optimized by PhGraph.

PhGraph Versus ChGraph: PhGraph outperforms ChGraph by 54.07$\times$ on average. Despite the fact that ChGraph proposes chain-driven scheduling to improve the intrachain data locality, the existence of unexploited interchain locality and data conflicts still limits the efficiency. In contrast, PhGraph tackles these issues by eliminating memory access and employing highly efficient hierarchical reduction to avoid data races.

PhGraph Versus XuLin-F and XuLin: XuLin-F and XuLin are inferior to PhGraph by 37.72$\times$ and 12.44$\times$ on average, respectively. XuLin-F and XuLin are implemented on FPGA and ASIC platforms, employing the same data-centric execution model and hardware architecture. The major difference between them is that XuLin equips larger on-chip scratchpad memory and achieves higher frequency. Thus, XuLin performs better against XuLin-F. Taking XuLin as an example, there are two reasons for the performance improvement of PhGraph. First, although XuLin significantly optimizes the data locality by employing chunk merging and adaptive loading strategies to eliminate unnecessary data loads, it still suffers from data transfers between on-chip and off-chip memories, which does not exist in PhGraph. Second, the parallelism of XuLin suffers from the limited off-chip memory bandwidth, which prevents XuLin from scaling with the number of processing elements. In contrast, both APIM and DPIM crossbars in PhGraph can achieve high parallelism. Particularly, PR exhibits the most significant improvement compared to other algorithms, with a speedup of 38.97$\times$ against XuLin on average. The reasons are twofold. First, the processing logic of PR can perfectly match the APIM architecture. Second, all hyperedges and vertices for PR are active, and they can make full use of ReRAM crossbars with high-compute parallelism.

Energy Savings: Fig. 10 shows the energy consumption results. Since the BFS algorithm is supported only in existing Hygra, NWHy, and ChGraph, all of which integrate with high-power CPUs, we ignore BFS and select Hygra as a representative of CPU-based solutions for demonstrating better the energy superiority of PhGraph. Compared with Hygra, complex pipeline and ISA designs make CPUs consume over 22$\times$ power than PhGraph. The actual power of PhGraph is only 17.677 watts, while the actual power of two Intel Xeon Gold 6338 CPUs exceeds 400 watts.

Additionally, thanks to in-situ processing capability and customized hardware designs for hypergraph processing,

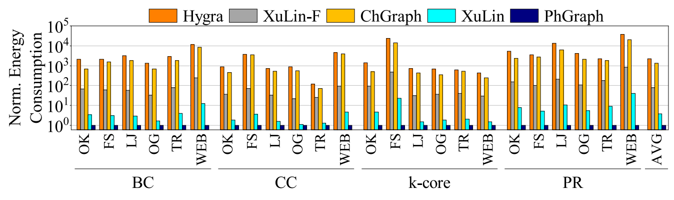


Fig. 10. Normalized energy consumption over PhGraph.

PhGraph consumes 2 247.83$\times$ less energy than Hygra on average. ChGraph designs a hardware prefetcher integrated with CPUs. Therefore, the energy consumption of ChGraph is still constrained by the high power of CPUs. Although ChGraph improves data locality and overall performance than Hygra, PhGraph saves 1 342.47$\times$ energy on average against ChGraph. XuLin-F and XuLin can make full use of cached data to reduce the off-chip memory accesses, but PhGraph still offers 83.15$\times$ less energy consumption against XuLin-F and 3.89$\times$ less against XuLin. The reasons are twofold. First, a substantial amount of data movement is eliminated by the in-situ processing. Second, the ReRAM crossbars can be utilized fully via sophisticated workload polarization.

Power and Area Breakdown: Table I shows that the total power and area of PhGraph are 17.677 watts and 30.3 mm^2, respectively. The APIM and DPIM crossbar arrays and their peripheral circuits in DPEs and SPEs take up 64.6% and 16.3% of chip power and area, respectively. Within DPEs and SPEs, the peripheral circuits, such as ADCs and DACs, consume most of the power (95.24%) and area (99.00%) in DPEs, while the crossbars occupy most of the power and area in SPEs at 75.00% (power) and 97.18% (area). Memory and buffers take up 30.3% and 42.1% of chip power and area. The hierarchy reducers occupy 5.1% of power and 41.6% of area.

C. Effectiveness

The high performance of PhGraph can be attributed to the high-parallel in-situ processing of ReRAM-based architecture. However, the effectiveness of the hardware and software designs is also critical to the performance benefits.

1) Hardware Effectiveness: There are two aspects for the consideration of hardware effectiveness: 1) the hybrid architecture design and 2) the HRD. As depicted in Fig. 11(a), we evaluate the performance of the APIM-only and DPIM-only architectures with all our software designs to demonstrate the effectiveness of the APIM-DPIM hybrid architecture design. The execution time for APIM-only and DPIM-only is broken down into dense and sparse partition execution time. Since

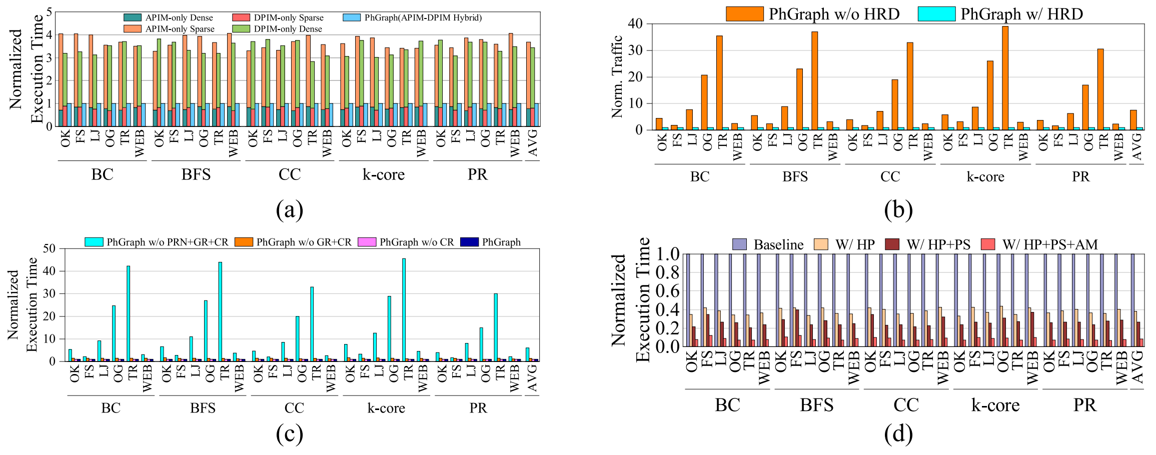


Fig. 11. Effectiveness of both hardware and software designs of PhGraph. (a) Hardware Effectiveness: Performance of APIM-only, DPIM-only, and APIM-DPIM hybrid PhGraph, breaking down by processing dense and sparse partitions. (b) Hardware Effectiveness: On-chip traffic of PhGraph with and without the HRD. All results are normalized to PhGraph w/HRD. (c) Hardware Effectiveness: Performance of PhGraph with and without PRN, GR, and controller (CR). (d) Software Effectiveness: Performance of PhGraph with and without HP, partitions scheduling (PS), and AM.

the hybrid architecture in PhGraph leverages the potential of both APIM and DPIM architectures, we can see that the performance of both APIM-only and DPIM-only is much lower than hybrid architecture in PhGraph. Processing sparse (dense) partitions dominates for APIM-only (DPIM-only) architecture, which confirms our analysis that APIM and DPIM are expected to accelerate workloads with different sparsity, as discussed in Section II-D.

As for the effectiveness of the HRD, we first evaluate the traffic reduction introduced by HRD, as shown in Fig. 11(b). The experimental results indicate that on-chip communication traffic is reduced by 7.54× on average by applying HRD, which can bring significant overall performance improvement. Furthermore, we investigate the benefits breakdown for PRN, GR, and Controller (CR), evaluating the effectiveness of these components, as shown in Fig. 11(c). PRN contributes an average of 5.98× performance improvement over the baseline, while GR further improves performance by 1.39×, primarily due to the highly efficient reduction networks from local to global. CR is responsible for managing data interactions and has negligible impact on performance.

2) Software Effectiveness: Fig. 11(d) investigates the benefit breakdown for three software designs of PhGraph: 1) hypergraph partitioning (HP); 2) partition scheduling (PS); and 3) algorithm mapping (AM). The baseline represents the hardware implementation of PhGraph without any software designs.

HP: By polarizing hypergraph processing into sparse and dense workloads, the performance potential of DPIMs and APIMs can be fully utilized. Thus, we can see that HP contributes a speedup of 2.62× on average over the baseline, occupying 67.50% of overall performance.

PS: Based on the baseline with HP, PS further improves the overall performance by 1.43× on average, demonstrating the effectiveness of PS in improving load balance.

AM: Benefiting from the sophisticated AM, the crossbar utilization and efficiency can be improved. Therefore, AM further improves the total execution time by 3.18× on average, taking 19.97% of the overall benefit.

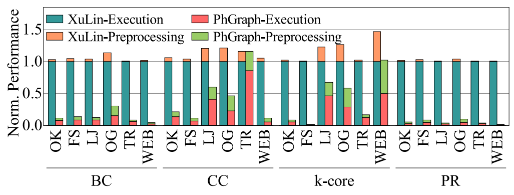


Fig. 12. End-to-end running time (including execution and preprocessing time) of PhGraph against XuLin, normalized to the execution time of XuLin.

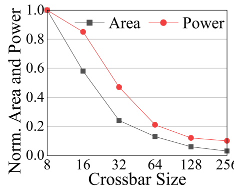


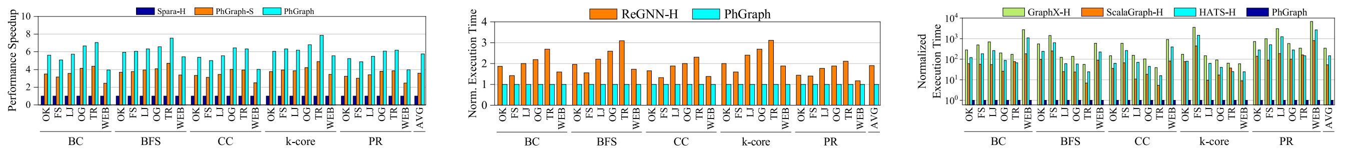
Fig. 13. Normalized area and power of APIM with various crossbar size.

D. Preprocessing

Both XuLin and PhGraph require extra preprocessing, such as hypergraph partitioning. We conduct an end-to-end performance evaluation of PhGraph against XuLin, as shown in Fig. 12, demonstrating that PhGraph still outperforms XuLin by 8.11× on average. We strike a balance between partitioning efficiency and effectiveness by using heuristics to select appropriate parameters, keeping the partitioning time acceptable. Furthermore, the most time-consuming procedure, line graph generation, is independent of hypergraph algorithms, which can be amortized by executing various algorithms on the same hypergraph.

E. Selection of Crossbar Size

The choice of crossbar size can significantly impact various aspects of the architecture, including utilization, power, and



(a) (b) (c)

Fig. 14. Performance comparation between PhGraph and existing graph accelerators enhanced for hypergraphs. (a) Performance comparison of PhGraph with the variant of Spara (Spara-H) and the variant of PhGraph (PhGraph-S). (b) Performance comparison of PhGraph with the variant of ReGNN (ReGNN-H). (c) Performance of GraphX-H, ScalaGraph-H, and HATS-H, against PhGraph.

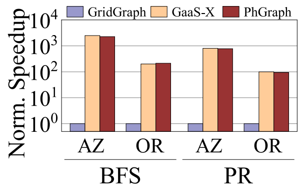


Fig. 15. Performance of PhGraph against GridGraph and GaaS-X for graph processing.

area. To choose a better crossbar size, we evaluate the normalized area and power of APIMs with various crossbar sizes from 8×8 to 256×256, as shown in Fig. 13. When the crossbar size exceeds 128, further increasing the crossbar size yields diminishing benefits. Conversely, due to the fixed computation granularity, this can lead to a reduction in overall parallelism. As a result, we extensively refer to existing works [18], [33], [37] and determine the crossbar size as 128×128.

F. Comparation With Graph Accelerators

Existing graph solutions based on CPUs [38], FPGAs [39], ASICs [40], and PIMs [15], [18] are not able to directly process hypergraphs. Even with some modifications to make them support hypergraphs, only suboptimal performance can be achieved. We enhance Spara [15] to enable hypergraph processing, denoted as Spara-H, and introduce the reordering method proposed by Spara to replace our partitioning algorithm, denoted as PhGraph-S. Fig. 14(a) shows the performance comparison between PhGraph and these two variants. PhGraph-S outperforms Spara-H by 3.58× on average, indicating the advanced architecture capabilities of PhGraph. PhGraph-S is inferior to PhGraph by 1.62× on average, further demonstrating the software design efficiency of PhGraph.

We also enhance the aggregation engine of ReGNN [18] to support hypergraph processing, denoted as ReGNN-H, and compare its performance with PhGraph, as depicted in Fig. 14(b). PhGraph outperforms ReGNN-H by 1.91× on average, which is attributed to not only the overlap-aware partitioning algorithm but also the delicate hardware design simultaneously supporting two-stage computation.

Furthermore, we make modifications to CPU-based GraphX [38], FPGA-based ScalaGraph [39], and ASIC-based HATS [40] to support hypergraph applications, donated as GraphX-H, ScalaGraph-H, and HATS-H, respectively, and compare the performance of PhGraph with these variants, as shown in Fig. 14(c). The experimental results demonstrate that PhGraph achieves performance improvements of 348.16×, 53.68×, and 147.73× compared to GraphX-H, ScalaGraph-H, and HATS-H, respectively. The speedup is attributed to reduced memory access and high-computation efficiency since the variants lack specialized optimizations for hypergraphs. ScalaGraph-H achieves efficient on-chip interconnection but considers the off-chip traffic less. HATS-H implements an efficient prefetcher, but the complexity of hypergraph structures leads to limited prefetching effectiveness.

G. Generality

PhGraph is capable of processing conventional graphs, although it is specifically designed for hypergraph applications. We implement graph algorithms BFS and PageRank on PhGraph and compare the performance against CPU-based graph system, GridGraph [25], and PIM-based graph accelerator, GaaS-X [13], on graph datasets *Amazon* (AZ) and *Orkut* (OR) [16], as shown in Fig. 15. Overall, PhGraph achieves similar performance over GaaS-X (0.92×∼1.05×) and 432.41× speedup over GridGraph on average.

H. Endurance Management

The wear-leveling policy is critical for the ReRAM endurance management due to the limited write endurance of ReRAM cells. For APIM, it is not necessary to consider wear-leveling because of the offline mapping strategy. For DPIM, PhGraph adopts a simple dynamic column address remapping strategy [20] to achieve wear leveling, improving the endurance by 9.92×.

V. Related Works

We would like to simplify this Section V Related Works to the following texts in LaTeX format.

Conventional Graph Processing: There have been several efforts for conventional graph processing in improving parallelism [41], mitigating data conflicts [43], reducing memory access overhead [45], [46], and optimizing for data sparsity [10], [12], [13], [15]. Graph reordering methods [10], [15] and heterogeneous architecture designs [12], [13] are most relevant to our software and hardware designs. However, all these solutions are designed for conventional graphs and not well-suited for hypergraph processing due to the unique hypergraph characteristics in both computation kernels and topology structure.

Hypergraph Processing Systems and Accelerators: There are serveral efforts for hypergraph processing in general-purpose systems for both in-memory [2] and distributed environments [1], [3]. Further, ChGraph [7] and XuLin [8] explores potential data locality in scheduling order and execution model, and propose dedicated accelerator designs. However, off-chip memory access remains massive and bottlenecks the performance. PhGraph is the first work to introduce PIM architecture into hypergraph, reducing data movement and achieving substantial benefits.

VI. Conclusion

In this article, we present PhGraph, the first PIM-featured hypergraph accelerator, incorporating two representative analog and digital PIM technologies for performance- and energy-efficient hypergraph processing. To boost hardware efficiency, PhGraph is equipped with three software-level co-designs. First, PhGraph employs an overlap-aware hypergraph partitioning mechanism to generate two levels of workload for ease of acceleration. Second, a partition scheduling strategy is adopted to improve hardware utilization. Third, PhGraph designs the algorithm mapping scheme elaborately to maximize execution efficiency. Experimental results show that PhGraph outperforms the state-of-the-art CPU-, FPGA-, and ASIC-based solutions significantly in terms of performance and energy savings.

References

[1] Y. Gu et al., "Distributed hypergraph processing using intersection graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 7, pp. 3182–3195, Jul. 2022.

[2] J. Shun, "Practical parallel hypergraph algorithms," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program*, 2020, pp. 232–249.

[3] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang, "HyperX: A scalable hypergraph framework," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 5, pp. 909–922, May 2019.

[4] X. Xia, H. Yin, J. Yu, Q. Wang, L. Cui, and X. Zhang, "Self-supervised hypergraph convolutional networks for session-based recommendation," in *Proc. Conf. Artif. Intell.*, 2021, pp. 4503–4511.

[5] H. Fan et al., "Heterogeneous hypergraph variational autoencoder for link prediction," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 8, pp. 4125–4138, Aug. 2022.

[6] A. Aghdaei, Z. Zhao, and Z. Feng, "HyperSF: Spectral hypergraph coarsening via flow-based local clustering," in *Proc. Int. Conf. Comput.-Aided Des.*, 2021, pp. 1–9.

[7] Q. Wang et al., "Hardware-accelerated hypergraph processing with chain-driven scheduling," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2022, pp. 184–198.

[8] Q. Wang et al., "A data-centric accelerator for high-performance hypergraph processing," in *Proc. Annu. Int. Symp. Microarchit.*, 2022, pp. 1326–1341.

[9] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 531–543.

[10] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "GraphSAR: A sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs," in *Proc. Asia South Pac. Des. Autom. Conf.*, 2019, pp. 120–126.

[11] G. Dai et al., "GraphH: A processing-in-memory architecture for large-scale graph processing," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 640–653, Apr. 2019.

[12] Y. Huang et al., "A heterogeneous PIM hardware-software co-design for energy-efficient graph processing," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2020, pp. 684–695.

[13] N. Challapalle et al., "GaaS-X: Graph Analytics accelerator supporting sparse data representation using crossbar architectures," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2020, pp. 433–445.

[14] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Design of cross-point metal-oxide ReRAM emphasizing reliability and cost," in *Proc. Int. Conf. Comput.-Aided Des.*, 2013, pp. 17–23.

[15] L. Zheng et al., "Spara: An energy-efficient ReRAM-based accelerator for sparse graph analytics applications," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2020, pp. 696–707.

[16] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." 2014. [Online]. Available: http://snap.stanford.edu/data

[17] S. Kvatinsky et al., "MAGIC—Memristor-aided logic," *IEEE Trans. Circuits Syst. II, Exp. Briefs,* vol. 61, no. 11, pp. 895–899, Nov. 2014.

[18] C. Liu et al., "ReGNN: A ReRAM-based heterogeneous architecture for general graph neural networks," in *Proc. Annu. Des. Autom. Conf.*, 2022, pp. 469–474.

[19] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided loGIC (MAGIC)," *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, Jul. 2016.

[20] H. Jin et al., "ReHy: A ReRAM-based digital/analog hybrid PIM architecture for accelerating CNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2872–2884, Nov. 2022.

[21] X. Qian, "Graph processing and machine learning architectures with emerging memory technologies: A survey," *Sci. China Inf. Sci.*, vol. 64, no. 6, pp. 1–25, 2021.

[22] X. T. Liu et al., "Parallel algorithms for efficient computation of high-order line graphs of hypergraphs," in *Proc. Int. Conf. High Perform. Comput., Data, Anal.*, 2021, pp. 312–321.

[23] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan, "X-MANN: A crossbar based architecture for memory augmented neural networks," in *Proc. Annu. Des. Autom. Conf.*, 2019, pp. 1–6.

[24] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He, "An efficient graph accelerator with parallel data conflict management," in *Proc. Int. Conf. Parallel Archit. Compilat. Techn.*, 2018, pp. 1–12.

[25] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[26] S. Mei, "A framework combines supervised learning and dense subgraphs discovery to predict protein complexes," *Front. Comput. Sci.*, vol. 16, no. 1, 2022, Art. no. 161901.

[27] P. Fang et al., "An efficient memory data organization strategy for application-characteristic graph processing," *Front. Comput. Sci.*, vol. 16, no. 1, 2022, Art. no. 161607.

[28] J. Kunegis, "KONECT: The Koblenz network collection," in *Proc. Int. Conf. World Wide Web*, 2013, pp. 1343–1350.

[29] P.-Y. Chen, X. Peng, and S. Yu, "NeuroSim: A circuit-level macro model for benchmarking Neuro-inspired architectures in online learning," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 12, pp. 3067–3080, Dec. 2018.

[30] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2013, pp. 475–486.

[31] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A general model for voltage-controlled memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs,* vol. 62, no. 8, pp. 786–790, Aug. 2015.

[32] Z. Zou et al., "BioHD: An efficient genome sequence search platform using HyperDimensional memorization," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 656–669.

[33] T. Yang et al., "PIMGCN: A ReRAM-based PIM design for graph convolutional network acceleration," in *Proc. 58th Annu. Des. Autom. Conf.*, 2021, pp. 583–588.

[34] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM: In-memory acceleration of deep neural network training with high precision," in *Proc. 46th Annu. Int. Symp. Comput. Archit.*, 2019, pp. 802–815.

[35] H. Labs. "CACTI." 2014. [Online]. Available: https://www.hpl.hp.com/research/cacti/

[36] X. T. Liu, J. Firoz, A. H. Gebremedhin, and A. Lumsdaine, "NWHy: A framework for hypergraph analytics: Representations, data structures, and algorithms," in *Proc. Int. Parallel Distrib. Process. Symp. Workshops*, 2022, pp. 275–284.

[37] D. Choudhury, L. Xiang, A. Rajam, A. Kalyanaraman, and P. P. Pande, "Accelerating graph computations on 3D NoC-enabled PIM architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 28, no. 3, pp. 1–16, Mar. 2023.

[38] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2014, pp. 599–613.

[39] P. Yao et al., "ScalaGraph: A scalable accelerator for massively parallel graph processing," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2022, pp. 199–212.

[40] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proc. 51st Annu. Int. Symp. Microarchit.*, 2018, pp. 1–14.

[41] H. Jin, P. Yao, and X. Liao, "Towards dataflow based graph processing," *Sci. China Inf. Sci.*, vol. 60, no. 12, 2017, Art no. 126102.

[42] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program*, 2013, pp. 135–146.

[43] Q. Wang, L. Zheng, J. Zhao, X. Liao, H. Jin, and J. Xue, "A conflict-free scheduler for high-performance graph processing on multi-pipeline FPGAs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, pp. 1–26, May 2020.

[44] D. Chen et al., "GraphFly: Efficient asynchronous streaming graphs processing via dependency-flow," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1–14.

[45] Q. Wang et al., "GraSU: A fast graph update library for FPGA-based dynamic graph processing," in *Proc. Symp. Field-Program. Gate Arrays*, 2021, pp. 149–159.

[46] D. Chen et al., "MetaNMP: Leveraging cartesian-like product to accelerate HGNNs with near-memory processing," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, 2023, pp. 1–13.

[47] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, "Directed hypergraphs and applications," *Discr. Appl. Math.*, vol. 42, no. 2, pp. 177–201, 1993.

Long Zheng (Member, IEEE) received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016.

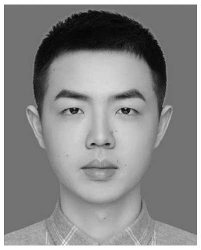
He is an Associate Professor with the School of Computer Science and Technology, HUST. His current research interests include program analysis, runtime systems, and heterogeneous computing with a particular focus on graph processing.

Ao Hu received the B.S. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2021, where he is currently pursuing the M.S. degree with the School of Computer Science and Technology.

His research interests focus on processing-in-memory architecture and hypergraph processing.

Qinggang Wang received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2023.

He is currently a Postdoctoral Fellow with Zhejiang Lab, Hangzhou, China. His current research interests include graph processing and reconfigurable computing.



Yu Huang received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2022.

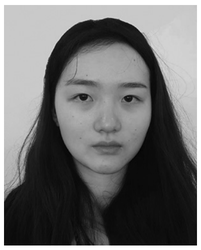
He is currently the Postdoctoral Fellow with Zhejiang Lab, Hangzhou, China. His research interests focus on processing-in-memory and graph processing.

Haoqin Huang is currently pursuing the master's degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

Her research interests focus on sparse graph processing accelerators and graph partitioning.

Pengcheng Yao received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2022.

He is currently the Postdoctoral Fellow with Zhejiang Lab, Hangzhou, China. His research interests focus on graph processing and domain specific accelerator.



Shuyi Xiong is currently pursuing the master's degree with the Huazhong University of Science and Technology, Wuhan, China.

Her research interests include program analysis and hypergraph computing.

Xiaofei Liao (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

He is currently a Professor with the School of Computer Science and Technology, HUST. His research interests are in the areas of system virtualization, system software, and cloud computing.

Hai Jin received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994.

He is a Chair Professor of Computer Science and Engineering with HUST. He was with The University of Hong Kong, Hong Kong, from 1998 to 2000, and as a Visiting Scholar with the University of Southern California, Los Angeles, CA, USA, from 1999 to 2000. He has coauthored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.

Dr. Jin was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz in Germany, in 1996. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a Fellow of CCF and a Life Member of the ACM.