HyGraph: Accelerating Graph Processing with Hybrid Memory-centric Computing

Minxuan Zhou, Muzhou Li, Mohsen Imani* and Tajana Rosing
Department of Computer Science and Engineering, University of California, San Diego
*Department of Computer Science, University of California, Irvine
miz087@ucsd.edu, mul023@ucsd.edu, m.imani@uci.edu, tajana@ucsd.edu

***Abstract*—Graph applications are challenging to run efficiently on conventional systems because of their large and irregular data. Several works have exploited near-data processing (NDP) based on emerging 3D-stacked memory to accelerate graph processing applications by offloading computations to massively parallel cores in the memory chip. Even though NDP can efficiently support parallel operations in a memory scalable way, it still requires data movement between memory and near-memory cores. Such data movement introduces large overhead because of the random data pattern in graph workloads. Furthermore, the parallelism provided by NDP systems is still insufficient for graph applications because of the limited number of processing cores. In this work, we tackle these challenges by integrating processing in-memory (PIM) technology in the NDP-based accelerator. We propose HyGraph, a software-hardware co-design for graph acceleration that exploits hybrid memory-centric computing technologies, including NDP and PIM. The design of HyGraph includes an optimization algorithm for hybrid memory layout, a run-time system combining both NDP and PIM processing flows, and customized hardware for efficiently enabling PIM functionality in NDP systems. Our experimental results show that HyGraph is up to 1.9× faster and 2.4× more energy-efficient than state-of-the-art memory-centric graph accelerators on several widely used graph algorithms with various real-world graphs.**

I. Introduction

Processing large real-world graph workloads may lead to long execution time because of limited parallelism, low data locality, and inefficient data communication [1], [5], [16]. Memory-centric processing is a promising technology to meet the requirements of parallelism, scalability, and data movement efficiency for graph processing applications [1], [12], [15]–[17]. One popular memory-centric technology is near-data processing (NDP), which utilizes emerging 3D-stacked memories (e.g., Hybrid Memory Cube [10]) with the support of massively parallel processing cores in the memory chip. Such NDP accelerators significantly improve the performance of parallel graph processing on extremely large data by simultaneously scaling the memory and computing parallelism [1]. However, NDP systems still separate the processing cores and the memory device, keeping the same data movements that may cause significant performance degradation because real-world graphs are usually random. In addition to the data movement issue, the parallelism of the NDP system is limited by the small number of NDP cores, which is insufficient for parallel graph applications.

Processing in-memory (PIM) is another memory-centric technology that directly processes data in the circuit level of various memory technologies including DRAM [4], SRAM [3], and non-volatile memory [6], [8], [9], [18]. PIM supports highly parallel operations by re-purposing the memory as a large SIMD processor, which processes a large volume of data appropriately placed in the memory. As compared to NDP, PIM can support a much higher degree of parallelism while reducing data movement between the memory and processing cores. However, PIM acceleration only achieves its full efficiency when processing large and uniform data, which randomly happens in graph applications. In this case, the previous PIM-based graph accelerators [15], [17] are highly specialized and have low memory utilization.

In this work, we propose HyGraph, a software-hardware co-design that exploits the emerging NDP-enabled hybrid memory cube (HMC) [10] and PIM-enabled DRAM technology [4] to accelerate graph processing applications by fully utilizing the advantage of each technology. HyGraph includes three key components, 1) a layout optimizer, 2) a run-time system, and 3) a custom hardware to enable an efficient acceleration in such hybrid memory-centric computing. Specifically, the layout optimizer finds an efficient data layout and the corresponding processing strategy, either NDP or PIM, for different parts of graph applications. We propose a graph-aware dynamic programming algorithm that explores this large design space. The optimized data layout is then used by the run-time system, which executes graph processing applications through a hybrid processing scheme - adopting different processing flows for NDP and PIM operations. In addition to the software-level support, we propose several custom hardware components in NDP architectures to efficiently support PIM functionality, including the PIM command execution and PIM data transfer.

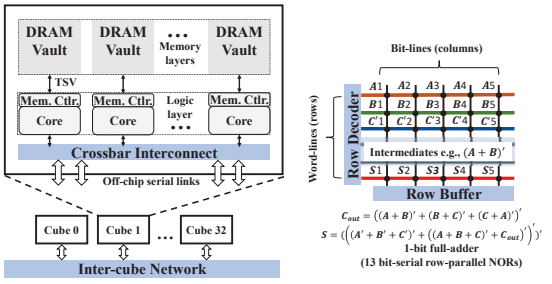
We evaluate the proposed ideas on three popular graph algorithms with seven real-world graph datasets. We compare HyGraph to several state-of-the-art memory-centric graph accelerators, including both NDP and PIM systems. Based on our experimental results, HyGraph can provide up to 1.9× speedup and 2.4× more energy efficiency. These results show that HyGraph can effectively utilize the advantages of NDP and PIM to accelerate a wide range of graph applications.

II. Background and Related Work

A. NDP Graph Processing

Figure 1 shows an NDP system, which is built upon a hybrid memory cube (HMC) system [1], [16]. The key component of the NDP system is the 3D-stacked memory cube, which has

978-3-9819263-5-4/DATE21/©2021 EDAA



(a) An HMC-based NDP system (b) PIM operation in memory array

Fig. 1. **The overview of memory-centric computing.**

Algorithm 1: One iteration in vertex program [16].

```
1 /* Process phase: Generate temporary property (update) for each vertex. */
2 for v : active_vertices do
3     for e : InComingEdges(v) do
4         process(&TempProp[v], v, e);
5 /* Apply phase: Update vertex properties. */
6 for v : V do
7     apply(&Prop[v], TempProp[v], v);
8     if Prop[v] is changed then
9         active_vertices.add(v);
```

a logic layer with multiple memory layers. The stack divides the memory vertically into several vaults and uses through-silicon-vias (TSVs) to transfer data. Each vault has a dedicated memory controller and a processing core. NDP cores can work independently to provide high degree parallelism.

Existing NDP graph accelerators [1], [16] handle graph applications based on a widely used general framework - vertex programming model, as shown in Algorithm 1. The vertex program model abstracts graph applications as an iterative process for updating application-specific vertex properties. Each iteration first generates temporary properties for vertices by an application-specific *process* function on a working set of vertices (*active_vertices*). Next, it uses temporary properties to updated vertex properties with an *apply* function and generates *active_vertices* for the next iteration. This framework can implement different graph algorithms by customizing *process* and *apply* functions.

To process a vertex program in NDP systems, vertices are allocated to different memory locations. Each processing core can independently handle computations of vertices allocated to the corresponding vault, providing massive parallelism which scales with the data. However, such NDP graph acceleration has several issues. First, computations still require time and energy-consuming data movement between the memory and processing cores. Furthermore, the number of processing cores (e.g., 32 per cube) is insufficient for the parallelism available in graph processing (e.g., millions of vertices).

B. PIM Acceleration

PIM is a promising technology to solve the above issues in NDP graph acceleration. In PIM, the memory supports in-memory computations by applying specialized signals (e.g., voltage) to directly change the states of memory cells. We can implement various bit logic operations, including AND,

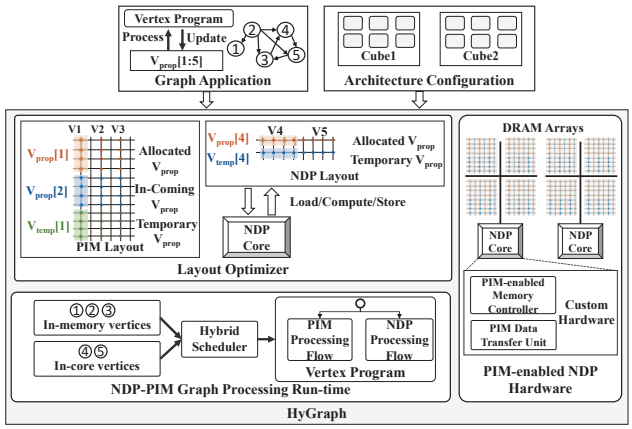


Fig. 2. **The overview of HyGraph.**

XOR, and NOR, which can be further used to realize custom functions (e.g., addition) by executing serials of bit logic operations (*bit-serial*), as shown in Figure 1(b). Such PIM operations can achieve high degree parallelism by processing all bit-lines in multiple rows simultaneously (*row-parallel*). In Figure 1, we can process all five elements in the vector addition $S = A+B$ in parallel. This is also applicable to computations on multi-bit values by serializing computations for each bit. Such PIM functionality has been implemented in various memory technologies including SRAM [3], non-volatile memory [6], [7], and DRAM [4], [13].

A recent study has shown that we can transform computations of vertex properties into very long vector/matrix operations that can fit into PIM to achieve better performance and energy efficiency [15], [17]. However, such PIM accelerators adopt a highly specialized memory architecture that can only efficiently support PIM operations under heavy and costly hardware modification. In this work, we directly exploit the PIM capability of commodity DRAM technologies in the existing HMC architecture. Such design is more general and practical than previous PIM-based accelerators. Furthermore, this work exploits the advantages of NDP and PIM for accelerating different portions of the graph application and shows several significant benefits over the NDP-only or PIM-only solutions.

III. NDP-PIM Hybrid Graph Acceleration

This section introduces the proposed hybrid NDP-PIM design for graph processing. The target architecture is an HMC-based NDP system with the support of PIM functionality in the memory layers. Figure 2 shows the overall design of HyGraph, consisting of three key components: layout optimizer, NDP-PIM run-time, and PIM-enabled NDP hardware.

The layout optimizer determines the data layout strategies, either PIM or NDP layout, used for different vertices based on both software and hardware configurations. As shown in Figure 2, the PIM data layout places data of the vertex program in a *row-parallel* way, while the NDP data layout allocates contiguous memory space for data and utilizes the NDP cores to process computations. The layout optimizer adopts a graph-aware optimization algorithm based on dynamic programming to find an efficient data layout strategy. The data layout strategy

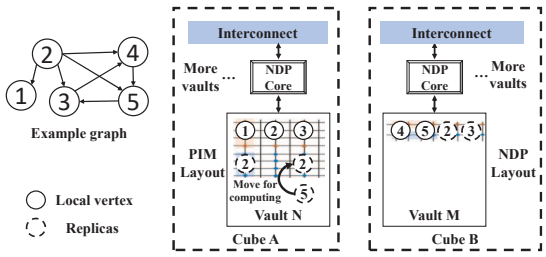


Fig. 3. **Hybrid source-cut vertex allocation.**

also determines the operations required for processing the vertex program. For example, vertex data using the PIM layout requires *bit-serial row-parallel* PIM operations. However, it is hard to efficiently process PIM operations using the NDP execution flow which only exploits the parallelism across NDP cores, ignoring the PIM parallelism inside each core. Therefore, we propose a run-time system using a hybrid scheduling process to handle PIM and NDP data in separate flows. Furthermore, to efficiently support PIM functionality, HyGraph adopts several customized hardware components, including a PIM-enabled memory controller with the support for PIM commands, and a PIM data transfer unit which uses a batching-based method for transferring PIM data. In the following sections, we introduce the detailed design for each key component in HyGraph.

A. Graph-aware Layout Optimization

As introduced in Section II, parallel PIM operations can only process data organized in a row-parallel way, which is different from the data layout in conventional memory. The HyGraph layout optimizer determines the data layout for all application data before the run-time of hybrid graph processing so that the system knows the correct data address to schedule both NDP and PIM operations.

1) ***Hybrid Layout for Source-cut Allocation****:* The optimization works with the source-cut vertex allocation, which has been used in the state-of-the-art NDP graph processing system to improve the inter-cube communication efficiency [16]. With the source-cut strategy, vertex properties are partitioned across different memory cubes based on source vertices of all edges. Each cube stores properties of local vertices (vertices allocated to this cube) and a replica for the property of each remote source vertex connecting to a local vertex through an edge. Unlike the original source-cut algorithm, which focuses on the cube-level vertex allocation, the source-cut allocation used in this work considers the vault-level allocation to explore the fine-grained hybrid layout inside each cube.

Figure 3 shows details of the vault-level source-cut hybrid layout. Each vault has a specific type of layout, either in-core or in-memory. Each in-core vault simply stores properties of all local vertices and replicas of remote vertices with conventional memory allocation. For an in-memory vault, we align the properties of all local vertices in the *row-parallel* layout to exploit PIM functionality for processing these properties in parallel. For each local vertex, we allocate memory rows for computing with vertices connecting to it. If a remote vertex connects to multiple local vertices in a vault, we allocate multiple replicas in this vault to enable parallel PIM computations.

Algorithm 2: Dynamic programming based optimization framework for hybrid layout.

```
1 for m ← 1 to #vaults do
2     for i ← 1 to |V| do
3         for j ← 1 to i do
4             if f[m][i] > f[m − 1][j − 1] + CoreCost(j, i) and
               CoreLayout(j, i) fit in one vault then
5                 f[m][i] = f[m − 1][j − 1] + CoreCost(j, i));
6                 decision[m][i] = InCore(j, i);
7             if f[m][i] > f[m − 1][j − 1] + MemCost(j, i)) and
               MemLayout(j, i) fit in one vault then
8                 f[m][i] = f[m − 1][j − 1] + MemCost(j, i));
9                 decision[m][i] = InMemory(j, i);
```

The computation efficiency of each in-memory vault depends on the maximum number of in-coming active vertices among all local vertices. As shown in Figure 3, vault N needs to sequentially issue 2 PIM operations because vertex 3 has 2 in-coming vertices (vertex 2 and vertex 4). However, each of the other vertices in this vault only has one incoming message, resulting in imbalanced computations. Because PIM operation takes much more cycles than in-core computation, the PIM layout may not be efficient to process only a few vertexes with high in-degrees. Instead, we can process such vertices in the NDP cores or simultaneously process many high in-degree vertices for PIM operations to achieve better performance.

2) ***Graph-aware Layout Optimization****:* The key problem of the hybrid layout is to determine an efficient strategy that utilizes NDP and PIM to accelerate different portions of vertex programming. We propose a graph-aware optimization that holistically optimizes this problem. Algorithm 2 shows the algorithm of proposed optimization, which is based on dynamic programming. The dynamic programming algorithm explores the design space of allocating different groups of vertices in memory vaults with either in-core or in-memory layout. The algorithm iterates over the number of vaults used for vertex allocation. For each iteration m, the algorithm checks the costs of allocating each possible group of continuous vertices in the m^{th} vault based on the minimum costs using $m-1$ vaults.

To allocate a group of vertices in a vault, the algorithm compares the costs of in-core and in-memory layouts and whether this vertex group can fit in a vault with each layout strategy. Based on this scheme, the algorithm can find the allocation and layout strategy with minimum cost for allocating different numbers of vertices in different numbers of vaults.

The efficiency of the optimization algorithm significantly depends on the accuracy of cost estimation based on the operations required for applications and the hardware characteristics of different operations. We can accurately estimate the hardware characteristics of different operations using validated hardware simulation and publicly available product specifications. Section IV introduces the detailed simulation infrastructure used in this work. However, it is hard to know the required operations during application run-time because the layout phase happens in the offline stage. Therefore, we can only estimate the required operations based on the static graph structure and graph algorithm. We adopt an upper-bound estimation that uses

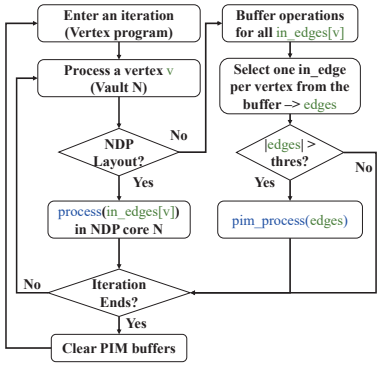


Fig. 4. **Processing flow of the hybrid run-time.**

the in-degree as the maximum number of operations required for each vertex. Based on this, we can estimate the maximum execution cost of processing a group of vertices using each layout.

B. Hybrid Graph Processing Run-Time

PIM operations can provide performance improvement only if we process a lot of computations at the same time. In the NDP-only graph acceleration, the vertex program parallelizes computations for different vertices across NDP cores, and each NDP core sequentially processes all in-coming edges for each vertex. However, this processing flow is not suitable for the PIM data layout, which can exploit the parallelism inside a vault with *row-parallel* operations. To schedule parallel PIM operations, we propose a graph processing run-time which uses different flows to process in-core and in-memory operations.

Figure 4 shows a flowchart of the hybrid run-time, which processes all active vertices in an iteration of the vertex program. To process an in-core vertex, the run-time system sequentially calls the $process$ function for all in-coming edges of the vertex on the corresponding NDP core and exploits the parallelism of NDP cores to process multiple vertices at the same time. To exploit the advantages of parallel PIM operations, the run-time uses a batching scheme to process in-memory vertices. Instead of processing up-coming operations for in-memory vertex properties immediately, the run-time buffers these vertex operations (related to in-coming edges) in a queue and then schedules parallel PIM computations to process a batch of buffered operations.

During the scheduling period, the run-time system selects one edge for each buffered vertex to create the processing batch. To avoid scheduling few operations each time, the run-time only schedules PIM operations when the batch size is larger than a threshold and then sets up the PIM operations, including data movement commands to align operands in the memory. Such data movements only involve the data that is specific to the in-coming edges (e.g., vertex properties and edge weights). In this case, we can avoid setting up all constants every time we issue PIM operations. In Section III-C, we further improve the performance of setting up PIM data by hardware modification. After aligning operands in the memory for computing a PIM batch, the run-time issues PIM commands in corresponding memory components to process the vertex program function.

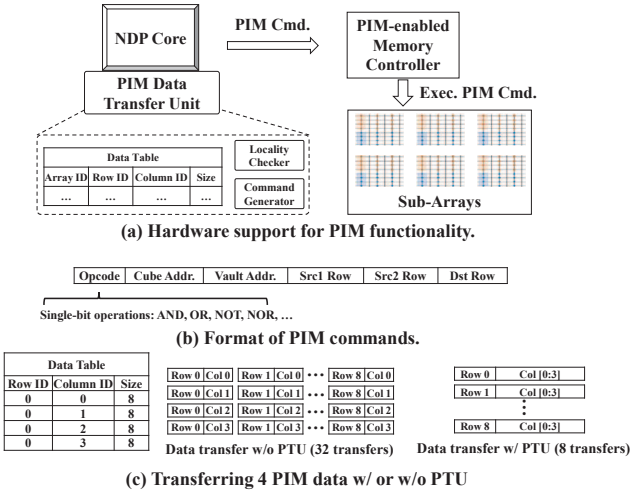


Fig. 5. **Hardware support for PIM functionality.**

Section III-C introduces the detailed format of PIM commands handled by the hybrid system, which ensures the flexibility and the parallelism of PIM operations.

C. Hardware Support

HyGraph contains two customized hardware components, including a PIM-enabled memory controller and a PIM data transfer unit, to effectively and efficiently process graph applications on the NDP-PIM hybrid system, as shown in Figure 5.

1) ***PIM-enabled Memory Controller****:* Previous research has shown that commodity DRAM chips can support arbitrary PIM computations by custom timing constraints of the memory controller [4]. This work adopts such controller design in the NDP device to enable different PIM operations through several new memory commands handled by the memory controllers.

Figure 5(b) shows the detailed format of proposed PIM-enabled memory commands. Each PIM command has several fields to indicate the memory area involving in the corresponding operation. Unlike the address in normal memory command, which only points to a specific byte, operands in PIM command may take effect on several rows in a memory array and process all bits in operand rows simultaneously. We use the row address as the operands in PIM commands for various logic operations including AND, OR, NOR, and XOR. Each PIM computation command stores the results of bit-wise logic computations in a result row. In this work, we use the vault as the basic unit of handling each PIM command to exploit the vault-level parallelism. Therefore, all operands in a PIM command share the same vault address, including the cube ID and vault ID.

2) ***PIM Data Transfer Unit****:* During the execution of graph processing, we need to update the application frequently. However, PIM operations require a cross-row layout for each value, which is incompatible with conventional row-buffer based DRAM design, hence under-utilizing the memory bandwidth. We propose a PIM data transfer unit (PTU) in each vault controller to handle PIM data transfer to solve this problem. Figure 5(c) shows the design of PTU, which consists of a data table, a

TABLE I
GRAPH WORKLOAD SUMMARY

graph	amazon	twitter	pokec	wiki	ljournal	uniform	g500
#vertex	403K	81K	1.6M	1.8M	5.4M	2.1M	2.1M
#edge	3.4M	1.8M	30.6M	28.5M	78M	33M	32M

locality checker, and a command generator. PTU stores the memory operations for PIM data in the data table with target memory locations. The data table is a content addressable memory (CAM) to support efficient searches. The locality checker periodically checks the stored operations in the data table to find several commands that access the same set of memory rows. If the number of such operations is larger than a threshold or the data table is full, the command generator selects a set of memory rows affecting the most memory operations stored in the table and generates corresponding row-based memory commands to the vault controller. As shown in the figure, PTU can reduce 4× data transfers for four 8-bit values.

IV. EXPERIMENTS

A. Experiment Setup

We implement an in-house cycle-accurate simulator in C++ to model different architectures including basic NDP architecture (Tesseract [1]), NDP architecture with the source-cut allocation (GraphP [16]), and HyGraph. We also model a PIM-only acceleration on the NDP architecture to compare HyGraph with PIM-based accelerator under the same hardware constraints. The parameters of NDP architectures based on HMC technologies are calculated based on previous work [1], [4], and published data [10]. We use the largest HMC cube, which has eight 8Gb layers per cube, and each cube has 32 vaults. Each vault has a dedicated memory controller and an NDP core, which is ARM Cortex-A5 processor [1]. The memory command bus runs at 500MHz frequency [4], [10]. The system has 16 cubes linked with eight 40 GB/s high-speed serial links per cube. We use Cacti [14] to simulate the latency and energy consumption of the customized hardware components at 32nm technology. We test the performance of our proposed designs on three popular graph applications, including breadth-first search (BFS), single-source shortest path (SSSP), and page rank (PR). Table I lists all tested graphs including 5 real-world graphs (amazon, twitter, pokec, wiki, ljournal) and 2 synthetic graphs (uniform, g500) [2].

B. Comparison to State-of-the-arts

We first compare HyGraph to several state-of-the-arts NDP and PIM graph processing systems [1], [16].

1) ***Performance****:* Figure 6 shows the performance comparison between Tesseract [1], GraphP [16], PIM-only acceleration, and HyGraph. We observe general performance improvements across different graph algorithms, where the speedups over Tessearct (GraphP) are 2.0×(1.4×), 1.9×(1.2×), and 2.0×(1.3×) for BFS, SSSP, and PR respectively. As compared with PIM-only solution, HyGraph exhibits 1.4× better performance on average because we handle non PIM-friendly parts in NDP cores. Such results show that we can achieve significant performance improvements by utilizing PIM functionality in NDP systems for different graph algorithms.

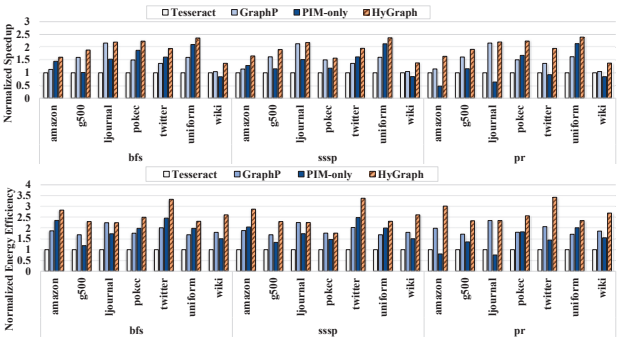


Fig. 6. **Cross-system performance and energy efficiency.**

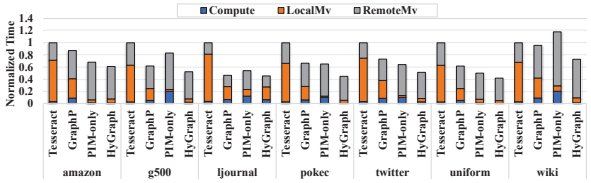


Fig. 7. **The time breakdown of page rank on four systems.**

The performance improvements of HyGraph across different graphs vary more significantly than improvements across different algorithms. For example, the average speedups of HyGraph for `ljournal` and `g500` over three algorithms are only 1.06× and 1.1×, as compared to GraphP. Such a gap between different graphs is caused by the graph structure, where most of the sub-graphs are not suitable for PIM operations. For other graphs, the hybrid solution can utilize PIM to process the highly parallel portion of graph processing, which cannot be fully accelerated by NDP cores.

2) ***Energy Efficiency****:* Figure 6 also shows the results of the energy consumption of three systems across tested workloads. HyGraph achieves 2.4×, 1.4× and 1.6× better energy efficiency over Tesseract, GraphP, and PIM-only respectively. As compared to performance improvements, the energy efficiency is slightly larger because the hybrid solution decreases energy-consuming operations from either NDP or PIM-only acceleration. For example, PIM operations can reduce costly data movements in NDP systems. However, PIM operations also introduce extra data movements because each computation requires a copy of the operand in the memory, which may require multiple transfers for a value. Furthermore, transferring PIM requires multiple row transfers because of the row-parallel layout (Section III-C).

3) ***Operation Breakdown****:* The above results show the proposed hybrid system provides significant improvements over previous memory-centric graph accelerators. To have more insights on such improvements, we investigate the detailed breakdown of performance and energy consumption on three key operations: computations, local data movement, and remote data movement. As compared to GraphP, the results show that HyGraph can decrease the latency of computation and local data movement by 2.5× and 4.7×, respectively, while introducing 1.1× more latency for remote data movement. The reduction of computation latency and local data movements mainly come from highly parallel PIM operations. As analyzed in the previous

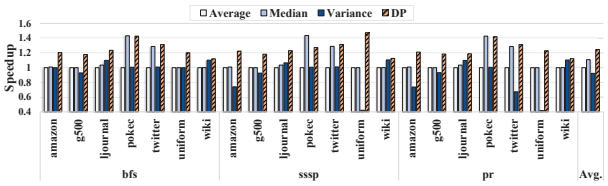


Fig. 8. **The performance of different layout methods.**

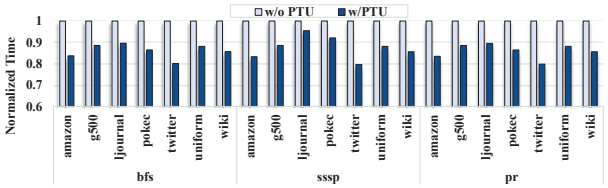


Fig. 9. **The performance of systems w/ or w/o PTU.**

section, PIM operations require more replicas to be transferred from remote vertices, increasing the data movements between different cubes. As compared to the PIM-only solution, HyGraph decreases the latency of computation and remote data movement by 35.8× 1.3×. The significant decrease in computation is a result of avoiding extremely sequential computations when processing high in-degree vertices.

C. Layout Strategy

We then compare the proposed layout algorithm to several heuristic-based layout strategies that evenly divide vertices into different vaults in the system and determine the data layout for each vault based on some statistics of allocated vertices. Specifically, the `average` (`median` or `variance`) method determines the data layout strategies of each group of vertices based on the average (median or variance) of in-degree across these vertices respectively. For each heuristic, we select the threshold with the best performance. Figure 8 shows the performance comparison of different layout strategies. All results are normalized to the `average` method. Based on our experiments, the DP-based algorithm provides 1.3×, 1.2×, and 1.4× better performance than the average, median, and variance-based methods, respectively. Such results show that the proposed DP-based algorithm can effectively find an efficient layout for different graph workloads.

The DP algorithm has a time complexity of $O(VN^2)$ where V and N are the numbers of vaults and vertices respectively. For scale-out graphs, this algorithm may take a long time to complete. However, we only need to run the algorithm once per graph workload offline. We can also reduce the time complexity of DP to $O(VCN)$ by constraining the number of vertices that can be allocated to a vault in a constant range of C.

D. Effects and Overhead of Custom Hardware

To evaluate the performance improvements provided by hardware customization, we compare HyGraph architecture to a PIM-enabled NDP architecture without PTU, which uses a conventional row-buffer to transfer all data. The result shows that PTU can effectively reduce 13.5% execution time because of more efficient data movements for PIM data. The most area consuming component is the data table in the PIM data transfer unit (PTU) which uses an SRAM-based CAM technology [11] (scaled to 32nm technology). Each data table contains 32 entries, and each entry has 5 bytes to store the information of address and data size. The area overhead of the data table is $0.0004mm^2$. Such area overhead is significantly smaller than an ARM Core, which takes $0.68mm^2$ area. Considering the large area available in the logic die of an HMC cube (e.g., $266mm^2$ for 8Gb DRAM chip), the area overhead of the proposed hardware is trivial.

V. Conclusion

In this work, we investigate the efficiency of integrating PIM-enabled memory layers in NDP graph accelerators. We propose a graph-aware source-cut algorithm the generates efficient data layout and operation scheduling in the hybrid system. We design a run-time system and custom hardware to efficiently support the NDP-PIM hybrid graph accelerator. Our experiment results show that the proposed method improves the performance and energy efficiency of state-of-the-art memory-centric systems by up to 1.9× and 2.4×. This work shows that the NDP-PIM hybrid system is promising for graph processing with appropriate software-hardware co-design.

Acknowledgment

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034. This work was also partially supported by SRC GRC Task No. 2988.001.

References

[1] J. Ahn et al. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA'15*. ACM/IEEE.
[2] S. Beamer et al. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IISWC'15*. IEEE.
[3] Charles Eckert et al. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA'18*. ACM/IEEE.
[4] Fei Gao et al. Computedram: In-memory compute using off-the-shelf drams. In *MICRO'19*. ACM/IEEE.
[5] Tae Jun Ham et al. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *MICRO'16*. ACM/IEEE.
[6] Mohsen Imani et al. Floatpim: In-memory acceleration of deep neural network training with high precision. In *ISCA'19*. ACM/IEEE.
[7] Mohsen Imani et al. Digitalpim: Digital-based processing in-memory for big data acceleration. In *GLSVLSI*, pages 429–434, 2019.
[8] Mohsen Imani et al. Deep learning acceleration with neuron-to-memory transformation. In *HPCA*, pages 1–14. IEEE, 2020.
[9] Mohsen Imani et al. Dual: Acceleration of clustering algorithms using digital-based processing in-memory. In *MICRO*, pages 356–371. IEEE, 2020.
[10] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSIT'12*. IEEE.
[11] Supreet Jeloka et al. A configurable tcam/bcam/sram using 28nm push-rule 6t bit cell. In *VLSI Circuits'15*. IEEE.
[12] Chao Li et al. A scalable design of multi-bit ferroelectric content addressable memory for data-centric computing. In *IEDM*. IEEE, 2020.
[13] Shuangchen Li et al. Drisa: A dram-based reconfigurable in-situ accelerator. In *MICRO'17*. ACM/IEEE.
[14] Naveen Muralimanohar et al. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *MICRO'07*. ACM/IEEE.
[15] L. Song et al. Graphr: Accelerating graph processing using reram. In *HPCA'18*. IEEE.
[16] Mingxing Zhang et al. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *IEEE HPCA'18*. IEEE.
[17] Minxuan Zhou et al. Gram: Graph processing in a reram-based computational memory. In *ASPDAC'19*. ACM.
[18] Minxuan Zhou et al. Gas: A heterogeneous memory architecture for graph processing. In *ISLPED*, pages 1–6, 2018.