2017 IEEE International Symposium on High Performance Computer Architecture

GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks

Lifeng Nai[†], Ramyad Hadidi[†], Jaewoong Sim[*], Hyojong Kim[†], Pranith Kumar[†], Hyesoon Kim[†]
[†]Georgia Institute of Technology, Atlanta GA
[*]Intel Labs, Portland OR
[†]{lnai3, rhadidi, hyojong.kim, hyesoon.kim}@gatech.edu
[*]jaewoong.sim@intel.com

***Abstract*—With the emergence of data science, graph computing has become increasingly important these days. Unfortunately, graph computing typically suffers from poor performance when mapped to modern computing systems because of the overhead of executing atomic operations and inefficient utilization of the memory subsystem. Meanwhile, emerging technologies, such as Hybrid Memory Cube (HMC), enable the processing-in-memory (PIM) functionality with offloading operations at an instruction level. Instruction offloading to the PIM side has considerable potentials to overcome the performance bottleneck of graph computing. Nevertheless, this functionality for graph workloads has not been fully explored, and its applications and shortcomings have not been well identified thus far.**

In this paper, we present GraphPIM, a full-stack solution for graph computing that achieves higher performance using PIM functionality. We perform an analysis on modern graph workloads to assess the applicability of PIM offloading and present hardware and software mechanisms to efficiently make use of the PIM functionality. Following the real-world HMC 2.0 specification, GraphPIM provides performance benefits for graph applications without any user code modification or ISA changes. In addition, we propose an extension to PIM operations that can further bring performance benefits for more graph applications. The evaluation results show that GraphPIM achieves up to a 2.4× speedup with a 37% reduction in energy consumption.

***Keywords*-processing-in-memory; PIM; graph computing; hybrid memory cube; HMC**

I. Introduction

The massive explosion in data volumes has made graph computing increasingly popular as a tool for processing large-scale network data over the past decades. Today, graph computing is being applied to a variety of domains, including social networks [1], e-commerce recommendations [2], and bio-informatics [3]. It is expected to become more prevalent in the future as many large-scale, real-world problems can be effectively modeled as graphs. Consequently, a significant amount of research efforts has been invested in graph computing to improve its execution efficiency, from high-level graph analytics [1] to low-level system implementations [4]–[7]. Despite all these efforts, however, graph computing still does not perform well on modern computing systems because of the irregular memory access to graph data. To improve the execution efficiency of graph processing, it is critical to provide a graph computing framework that incorporates *architectural innovations* to overcome the inefficient utilization of memory subsystems.

As a technique for addressing the bottleneck in the memory subsystem, processing-in-memory (PIM) was proposed a few decades ago with a variety of proposals [8]–[11]. Unfortunately, the initial attempt for PIM was not entirely successful because of its design complexity, fabrication difficulty, and less immediate needs. Recently, PIM architectures have regained the attention of researchers as a result of the advances in 3D-stacking technologies and an increasing concern on the memory bottleneck. Several PIM architectures and programming models have been recently proposed by academia [12]–[14], and memory vendors have also started to incorporate compute units into the memory architecture, such as Hybrid Memory Cube (HMC) proposed by Micron [15].

In this paper, we explore incorporating *real-world* PIM technology into graph computing to improve its execution efficiency by addressing hardware and software challenges. In particular, our study follows the HMC 2.0 specification that will be available in the near future. In HMC, a logic layer and several DRAM layers are stacked together using vertical interconnects called through-silicon vias (TSVs). The logic layer provides hardware for compute functionality as well as accommodates the memory controller for the DRAM layers. Starting from HMC 2.0, it supports the execution of 18 *atomic* operations in its logic layer.[1] Atomic operation support is limited to several basic operations, but it introduces the possibility of offloading computation at an instruction granularity. To this end, we propose GraphPIM, a full-stack solution that enables PIM for modern graph frameworks. GraphPIM involves addressing two key challenges.

What to Offload to PIM: Exploiting PIM in an effective way requires the identification of the right candidate for offloading, which has not been well discussed in prior studies. GraphPIM is based on the key observation that the atomic access to the graph property is the main culprit for the inefficient execution of graph workloads on modern systems. Thus, by offloading the atomic operations on the graph property to the PIM side, GraphPIM avoids the overhead of performing the atomic operations in the host processor and the inefficient utilization of the memory subsystem caused by irregular data accesses.

[1]HMC 2.0 differs from HMC Gen2, which follows the HMC 1.0 specification. HMC 2.0 hardware is not publicly available yet, but HMC atomic support is a practical, real-world design.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.54

How to Offload to PIM: Another key challenge is designing an interface between the host processor and PIM architectures, which is less intrusive to the computing ecosystem. Unlike a recent PIM study that requires programmers to explicitly invoke PIM operations using new host (native) instructions [14], GraphPIM does not add an extra burden on application programmers by leveraging existing host instructions. The key idea is to map host atomic instructions directly into PIM atomics using *uncacheable memory support* in modern architectures. With this approach, we demonstrate that we can provide performance benefits for a wide range of graph workloads without any changes in user applications or ISA; we only need a minor extension to the host processor and the graph framework. As a result, GraphPIM is more non-intrusive to the current software and hardware environment than other solutions, which we discuss in Section III-B.

In summary, this paper makes the following contributions.

- We study a wide range of graph workloads and identify the potential target for PIM offloading. We demonstrate that the key performance benefit of PIM for graph computing comes from reducing atomic overhead.
- We propose GraphPIM, which efficiently utilizes the real-world PIM functionality for graph computing. With minor architectural extensions to support PIM instruction offloading, our GraphPIM solution significantly improves graph processing performance (up to 2.4x) without any changes in user applications and ISA.
- We study the applicability of the atomic operations in HMC 2.0 for modern graph workloads. Based on the analysis, we propose a potential extension to the current set of HMC atomics and enable the PIM functionality for more graph applications.

II. Background and Motivation

Processing-in-memory (PIM) is a decades-old concept of incorporating computation functionality directly in the memory. The integrated compute units can be fully programmable cores, such as CPU and GPU, or simple ones that execute fixed-function PIM operations. As one of the first few industrial practices of PIM, Hybrid Memory Cube (HMC) provides compute capability, starting from HMC 2.0 [16]. In this section, we first provide background on the PIM implemented in HMC and then discuss how to exploit it for graph workloads.

A. HMC-Atomic Operation

In addition to a dramatic improvement in memory bandwidth, HMC introduces the possibility of supporting a variety of processing-in-memory (PIM) functionalities within the memory cube. PIM operations in HMC basically perform three steps: reading data from DRAM, performing computation on the data in the logic die, and then writing back the result to the same DRAM location. According to HMC 2.0, the PIM units perform read-modify-write (RMW) operations *atomically* within an HMC package. The corresponding DRAM bank is locked during the RMW operation, so any other memory requests to the same bank cannot be serviced. In addition, all PIM operations include only one memory operand; the operations are performed on an immediate value and a memory operand.

TABLE I: Atomic operations in HMC 2.0

Type	Data Size	Operation	Return
Arithmetic	8/16 byte	single/dual signed add	w/ or w/o
Bitwise	8/16 byte	swap, bit write	w/ or w/o
Boolean	16 byte	AND/NAND/OR/NOR/XOR	w/o
Comparison	8/16 byte	CAS-if equal/zero/greater/less, compare if equal	w/ w/

Table I lists several types of PIM operations supported by HMC 2.0: arithmetic, bitwise, boolean, and comparison. Although some operations also support 8-byte size, the default data size of PIM operations is 16 bytes. Depending on the definition of specific commands, a response may or may not be returned. If the response is returned, it will include an atomic flag that indicates whether the atomic operation was successful. Depending on the commands, the original memory data may also be returned along with the response.

B. Modern Graph Computing

Graph computing has been applied to a variety of domains as an important tool for processing large-scale network data. In real-world practices, because of the unique characteristics of graph data, graph computing shows distinct and diverse behaviors that are different from other computing types.

Framework-Based Computing: Unlike general applications that are often written from scratch, graph computing applications are typically implemented on top of underlying *graph frameworks* [5]–[7]. Graph frameworks provide user primitives for elementary graph operations, such as finding vertices and updating graph properties, while hiding the complexity of graph data management from application programmers. In other words, graph frameworks decouple user application code from low-level data management and OS/hardware-related code. This allows us to integrate optimization techniques into the graph frameworks without adding an extra burden on programmers.

Diversity of Graph Applications: The complexity and diversity of graph data result in a wide range of graph workloads with various computation behaviors [1], [17]. Modern graph applications are generally classified into three categories depending on the computation type.

(1) Graph Traversal (GT): In this category of applications, most computation occurs while traversing the graph through edges and jumping from a vertex to other connected vertices that are scattered through the graph. Traversing incurs a large number of irregular memory accesses for applications such as breadth-first search, shortest path, and page rank [18].

(2) Rich Property (RP): In this category, vertices are associated with *rich* graph properties, which can be as complex as large stochastic tables, such as those in Gibbs inference [19] and Bayesian network computation [20]. Computation in this category, unlike graph traversal (GT), occurs within the graph

properties. As this category of graph applications performs heavy numeric operations, it shows behaviors similar to conventional applications.

(3) Dynamic Graph (DG): Applications in this category perform computations on the dynamic graphs where the structure changes over time. Examples include graph triangulation (TMorph) and streaming graph (GCons). The applications in this category show memory-intensive behavior and irregular memory accesses similar to the static graphs. However, the dynamic graph structure leads to more diverse computation behavior than GT because of dynamic memory footprint and heavy write accesses.

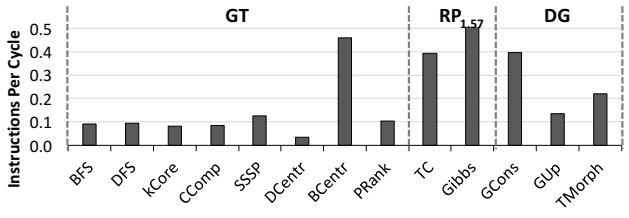

Fig. 1: Instructions per cycle (IPC) of graph workloads on an Intel Xeon E5 machine

Inefficient Execution on Modern Systems: To understand the performance behaviors of diverse graph applications on modern architectures, we measure the instructions per cycle (IPC) of typical graph workloads in each category on an 8-core Intel Xeon E5 machine. As shown in Figure 1, most workloads experience extremely poor performance. For example, many applications in the GT category show an IPC of less than 0.1, and while the workloads in DG show a bit higher performance than GT, they are still well below an IPC of 1. In general, graph computing applications (especially for the applications in the GT and DG categories) suffer from significantly poor performance on conventional architectures.

C. Bottlenecks in Graph Computing

The bottlenecks in graph computing arise from two major sources. First, graph computing typically entails a large number of *irregular memory accesses* because of the scattered graph connectivity. This makes on-chip caches mostly ineffective and thus leads to poor utilization of compute resources due to the access to the long-latency main memory. Second, graph data is typically processed in parallel due to its massive scale. Such parallel graph data processing heavily performs *atomic operations* to avoid contention of shared data. In general, atomic execution involves multiple operations and incurs non-negligible performance overhead [21]. Below, we provide an analysis to understand the bottlenecks of graph computing.

Irregular Memory Access: To understand the impact of irregular memory accesses on graph computing, we break down the execution time in the processor pipeline and measure the misses per kilo instructions (MPKI) of on-chip caches across a variety of graph workloads. The experiment is performed on an Intel Xeon E5 machine using hardware performance counters [22].

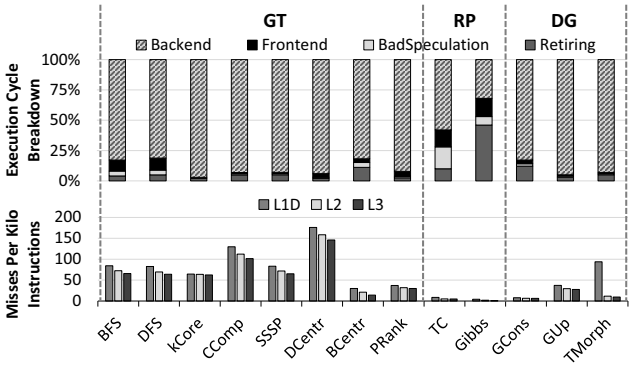

Fig. 2: Architectural behaviors of graph workloads on an Intel Xeon E5 machine

First, the top graph in Figure 2 shows the execution time breakdown following a top-down methodology described in the Intel manual [22], [23]. `Frontend` and `Backend` represent the execution time spent by frontend and backend-caused stalls.[2] Also, `BadSpeculation` shows the cycles resulting from miss speculation, while `Retiring` represents the cycles of successfully retired instructions. Note that the pipeline stalls caused by the memory subsystem are included in `Backend`.

As shown in the figure, graph computing spends most of its execution time on `Backend`, which is higher than 90% in some workloads, indicating that the memory subsystem is the key bottleneck for graph workloads. Such an observation is further supported by the MPKI results in the bottom graph, where L3 MPKI can be as high as 145 for the Degree Centrality (DCentr) workload. Also, the L2 and L3 caches do not provide sufficient benefit for most graph workloads as presented in the L2/L3 MPKI values.

Data Components: Figure 3 illustrates a code snippet of *breadth-first search* (BFS) using a *vertex-frontier* based algorithm [18]. The code goes through a loop that iterates over the traversal steps in a synchronized way. The algorithm in each step processes the vertices in the frontier that contains the vertices with the same depth. For each vertex, the algorithm checks the depth of its neighbors to see if it has been visited, and if not the depth information is updated using a *compare and swap (CAS)* atomic operation. Then, the newly visited vertices form the frontier for the next iteration. Here, the data access can be classified into three different components: meta data, graph structure, and graph property.

(1) Meta Data: Meta data include any local variables (e.g., `d`) and task queues (e.g., `F` and `F'`). These are frequently accessed and are also small, so they are cache-friendly. Thus, the access to meta data mostly hits in the L1/L2 caches.

(2)Graph Structure: To check the status of neighbors, the graph structure is accessed for retrieving the neighbor vertices. Since each vertex's neighbor list is usually organized in an array-like data structure, access to the graph structure has good spatial locality. Thus, the memory requests to this component

[2] `Frontend` includes instruction fetch, decode and allocate. `Backend` includes instruction scheduling, executing, and retiring.

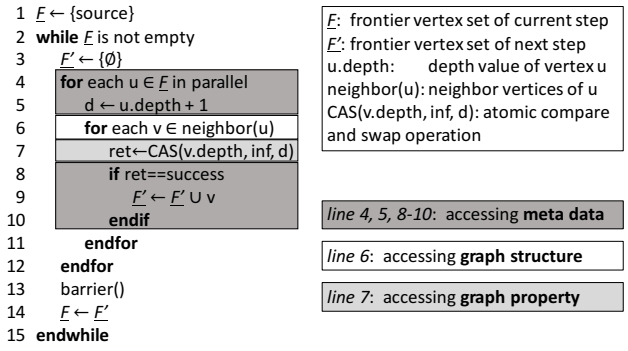

Fig. 3: Code snippet for breadth-first search (BFS)

also do not incur a large number of main memory accesses.

(3) Graph Property: During the traversal, BFS updates the property of the neighbor vertices. Due to the irregular nature of graph connectivity, working on the property list incurs accesses that are spread throughout the entire graph. However, only a small portion of the graph property is captured in the cache because of the large size of graph data. As a result, the access to the graph property usually leads to a high number of last-level cache (LLC) misses.

As explained above, the inefficient utilization of the memory subsystem is caused by the access to the *graph property* rather than to other data components. Also, due to the uncertain nature of graph connectivity, it is challenging to improve cache performance via conventional prefetching or data remapping techniques. In summary, we have two key observations: 1) the irregular access pattern occurs mostly in the graph property access (not spreading over all data components) and 2) the computation on the property data is a simple read-modify-write (RMW) *atomic* operation.

D. PIM Potential for Atomic Instructions

Any vertex in the graph can be shared among multiple threads. Thus, it is inevitable for most graph workloads to perform a large number of atomic operations to avoid contentions on the updates of the shared vertex property. For example, Figure 3 shows that all neighbor vertices' properties are accessed via CAS atomic operations. The heavy reliance on atomic operations incurs non-negligible performance overhead in modern general-purpose architectures [21].

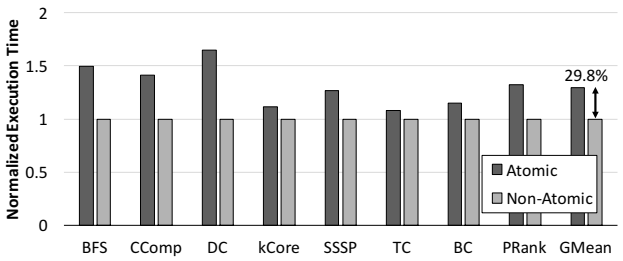

Fig. 4: Atomic instruction overhead of graph workloads on an Intel Xeon E5 machine

To measure the overhead of atomic operations for graph workloads, we conducted an experiment on an Intel Xeon E5 machine. We created micro-benchmarks performing one iteration of each graph workload and then ran the benchmarks while including/excluding the atomic operations on the graph property. As shown in Figure 4, compared with using regular read and write instructions, the atomic instruction incurs a 29.8% performance degradation on average (up to 64% for DCentr). The overhead of CPU atomic instructions comes not only from the extra cache invalidation and coherence traffic caused by the read-modify-write operation, but also from the pipeline freezing and write-buffer draining because of the consistency requirement. Such atomic overhead can potentially be avoided by utilizing PIM-Atomic in graph workloads. Because the read and atomic operations on the graph property occur at different execution phases and barriers guarantee that all previous atomic instructions are complete, the graph workloads naturally avoid consistency issues [24] and thus can benefit from using PIM-Atomic instructions.

III. GraphPIM Framework

A. Overview

Figure 5 shows an overview of our GraphPIM framework. GraphPIM enables instruction-level PIM offloading for generic graph computing frameworks with negligible changes in both software and hardware. Because of the separation of the user application layer from others, the changes are transparent to user applications.

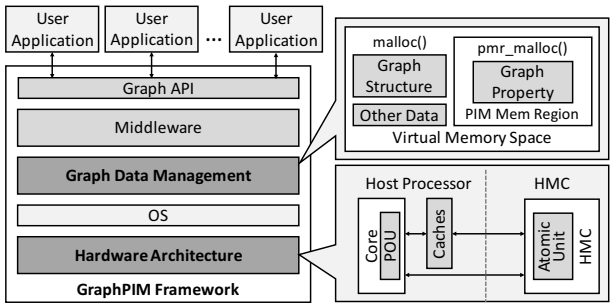

Fig. 5: Overview of GraphPIM framework

Graph-Data Management: As explained in Section II-C, both irregular memory accesses and atomic operation overhead in graph computing are caused by the access to the graph property. Therefore, in GraphPIM, we choose the atomic operations on the graph property as PIM offloading targets. To achieve this, GraphPIM requires the framework to allocate the graph property in the *PIM memory region (PMR)*, which is a continuous block of an uncacheable region in the virtual memory space. This is achieved by calling a customized *pmr_malloc* function (similar to jemalloc [25] and tcmalloc [26]). All host atomic instructions accessing the PMR are offloaded as PIM-Atomic requests.

Hardware Architecture: In GraphPIM, the host processor architecture implements a *PIM Offloading Unit (POU)* to

determine the data path of the current memory instruction. Atomic instructions accessing the PMR will bypass the cache hierarchy and be offloaded to HMC directly. The PIM region is uncacheable, so other non-atomic memory requests to the PIM region will also bypass the cache hierarchy.

Programming Model: An application programmer can use the same graph APIs and follow the same programming model provided by the underlying graph frameworks, so no application-level code change is required to benefit from GraphPIM from the programmer's perspective. The only change occurs *within* the framework. GraphPIM requires the graph framework to use a specialized *pmr_malloc* function to allocate memory space for the graph property. This modification to the framework is negligible and does not incur extra overhead for application programmers.

B. Architectural Extensions

Figure 6 shows the architectural extensions to the host processor in GraphPIM. We keep the architectural changes non-intrusive to current hardware architectures.

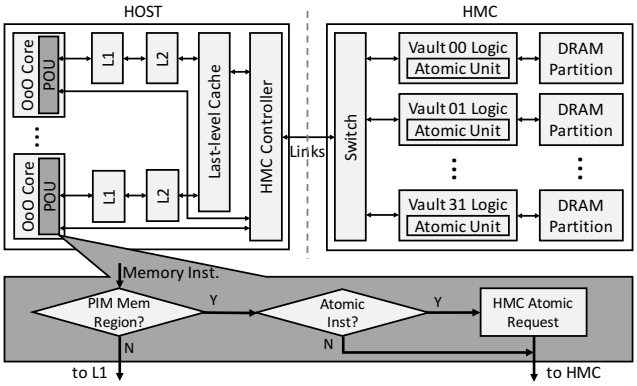

Fig. 6: Architectural extensions for GraphPIM (added parts are shown in dark gray)

PIM Memory Region: In GraphPIM, we define a PIM memory region (PMR) for the data of offloading targets. The PMR is specified in the virtual memory space by utilizing existing uncacheable (UC) memory support in x86 architectures [27]. The corresponding physical pages are marked as uncacheable by setting system registers (such as MTRRs in x86) from the operating systems. The underlying graph framework places the data of offloading targets into the PMR via a customized *pmr_malloc* function at the initial memory allocation phase.

PIM Offloading Unit: In each host core, GraphPIM integrates a PIM Offloading Unit (POU), which determines the data path of memory instructions. GraphPIM does not rely on special PIM instructions in the host processor, so the host processor ISA does not need to be changed. As shown in Figure 6, all atomic instructions, such as instructions with a "lock" prefix in x86, are regarded as HMC operations if they are accessing the PMR. Instead of being executed in the host processor, the atomic instructions are offloaded to the HMC by sending memory requests with atomic operation commands. Note that all other non-atomic instructions bypass the cache hierarchy just as with the original UC memory support.

Cache Policy: PIM-Atomic directly modifies the data within HMC. To maintain data coherency between HMC and cache, we follow a *cache bypassing* policy for offloading targets. By marking a page as uncacheable, all memory requests, including both offloading and non-offloading cases, will bypass the cache hierarchy if they are accessing the PMR. In this way, GraphPIM ensures that there is no data copy in the cache so that the coherence issue is avoided. Dealing with the cache bypassing policy is better than maintaining full coherence in terms of both performance and design complexity. Offloading targets in GraphPIM are graph property accesses, which are irregular. Thus, bypassing the caches for PIM-Atomic provides multiple benefits, such as avoiding unnecessary cache checking time, preventing cache pollution, and reducing memory bandwidth.

TABLE II: Summary of PIM offloading targets

Workload	Offloading Target	PIM-Atomic Type
Breadth-first search	lock cmpxchg	CAS if equal
Degree centrality	lock addw	Signed add
Shortest path	lock cmpxchg	CAS if equal
K-core decomposition	lock subw	Signed add
Connected component	lock cmpxchg	CAS if equal
Triangle count	lock add	Signed add

Offloading Target: GraphPIM regards the host atomic instructions that access the PMR as offloading targets. Table II summarizes the offloading targets for each workload as well as the corresponding PIM-Atomic operations (the workload applicability will be further discussed in Section III-C). As shown in the table, the corresponding x86 instructions with a "lock" prefix access the graph property and thus are offloaded to HMC. In the table, the graph workloads utilize two types of PIM-Atomic operations, `CAS if equal` and `Signed add`, which can be directly mapped from the host atomic instructions. Note that there are a few PIM-Atomic operations that do not match the host atomic instructions, such as `CAS if greater` and `CAS if less`. In the host processor, the functionality of these operations is achieved via a small instruction block that consists of other existing host atomic instructions. Such an instruction block is usually generated by the compiler. To fully utilize all PIM-Atomic operations, the host architecture may incorporate a mechanism to identify such small instruction blocks that can translate into the PIM-Atomic operations. Similar to other host atomic instructions, the identified instruction block will be regarded as a PIM offloading target if it is accessing the PMR.

Discussion: In GraphPIM, instead of adding special PIM instructions to the host processor, we choose to mark the special memory region for three major reasons: 1) Cache coherence. When using PIM instructions, because non-offloading instructions may also access the same data, we have to maintain costly coherence between data copies in the caches and memory. Such an issue is naturally avoided in our method because of the memory address-based PIM offloading. 2)

Programmer overhead. If new PIM instructions are introduced, application programmers typically must modify higher level software. With our proposed method, however, there is no extra burden for application programmers using PIM. Only a simple malloc function replacement is needed in the graph framework. 3) Cache checking overhead. In our method, all data accesses to PMR bypass the cache hierarchy, which brings extra performance benefits.

GraphPIM utilizes the atomic operations in the HMC 2.0 specification. Nevertheless, the proposed technique can be applied to other instruction-level PIM offloading environments. Likewise, GraphPIM can also be beneficial for non-graph workloads that perform atomic operations on irregular data. Furthermore, GraphPIM can be applied on systems equipped with both HMCs and DRAMs. In this case, the graph property data allocated in DRAMs will be processed in the conventional way, while the graph data in HMCs can still receive the same benefit from PIM-Atomic. In addition, it should be noted that without PIM-Atomic, bypassing the cache for atomic instructions would incur a huge performance degradation because the cache-line lock will be downgraded to bus locking in this case.

C. Applicability of PIM-Atomics

As discussed in Section II-B, modern graph computing covers a wide range of applications that exhibit different computation characteristics. Although we showed the feasibility of offloading the accesses to the graph property using BFS as an example, a further question may arise as to whether the same technique can be applied on other graph computing applications. In this section, we discuss the applicability of PIM-Atomic operations on various graph workloads.

The current PIM-Atomic support has two major limitations. First, only simple arithmetic operations are currently implemented; complex operations (e.g., floating point operations) are not supported in HMC 2.0. Second, only one memory operand is allowed in the operations. The operations that need to specify multiple memory locations have to be split into separate requests. Thus, to benefit from GraphPIM, the target applications should fulfill two key requirements: 1) the target workloads should contain a large number of irregular memory accesses triggered by atomic operations on the graph property and 2) the atomic operations on the target memory regions should be simple enough to be mapped to the existing PIM-Atomic operations. Most graph traversal applications, such as our BFS example in Figure 3, fulfill the requirements. To further study the applicability of PIM-Atomic on graph workloads, we analyze all workloads from the GraphBIG benchmark suite [17].

Inapplicable Graph Workloads: As shown in Table III, most of the traversal-oriented workloads, noted as Graph Traversal, can make use of PIM-Atomic operations. The two exceptions are Betweenness Centrality and Page Rank, which require floating point operations. On the other hand, the graph workloads in the Dynamic Graph (DG) category frequently perform graph structure/property updates and involve complex code structure and access patterns. Therefore, they require more complex memory operations, such as indirect accesses and multiple memory operands. For the workloads in the Rich Property (RP) category, Triangle Count can use PIM functionality. However, the workloads in this category perform computation within the vertices' properties, so they are more computation intensive than Graph Traversal. Thus, PIM-Atomic may not provide huge performance benefits for them.

TABLE III: Summary of PIM-Atomic applicability with GraphBIG workloads

Category	Workload	Applicable? (Missing operation)
Graph Traversal	Breadth-first search	√
	Depth-first search	√
	Degree centrality	√
	Betweenness centrality	× (Floating point add)
	Shortest path	√
	K-core decomposition	√
	Connected component	√
	Page rank	× (Floating point add)
Dynamic Graph	Graph construction	× (Complex operation)
	Graph update	× (Complex operation)
	Topology morphing	× (Complex operation)
Rich Property	Triangle count	√
	Gibbs inference	× (Computation intensive)

Potential Extension to PIM Atomics: The logic die in HMC enables the possibility of implementing a wide range of computation logic within the memory package. Although HMC 2.0 currently defines 18 simple operations, HMC technology has the full potential of implementing new operations if needed. As previously discussed, some applications deal with floating point (FP) operations; for example, both Betweenness Centrality and Page Rank perform FP add operations when updating the graph property. As FP add/sub operations are relatively simple compared to other complex ones, FP add/sub support might be a reasonable extension to the future PIM-Atomic to provide PIM benefits for more graph workloads. In Section IV, we further evaluate the performance benefits of supporting FP add/sub operations.

IV. Evaluation

A. Methodology

We evaluate our method using the Structural Simulation Toolkit (SST) [28] with MacSim [29], a cycle-level architecture simulator. HMC is simulated by VaultSim, a 3D-stacked memory simulator. We extend VaultSim with extra timing models based on DRAMSim2 [30]. Table IV shows the configuration of our evaluated system. We model a processor with 16 out-of-order cores and a single 8GB HMC that follows the HMC 2.0 specification [16]. We use the benchmarks from GraphBIG [17], a graph benchmark suite covering a wide scope of graph computing workloads, as the workloads, and the LDBC graph as the input dataset. In addition, two other large-scale graphs, *bitcoin* and *twitter*, are further used to evaluate our method with real-world applications.

TABLE IV: Simulation configuration

Component	Configuration
Core	16 out-of-order cores, 2GHz, 4-issue
Cache	32KB private L1 data/instruction caches 256KB private L2 inclusive cache 16MB shared L3 inclusive cache 64-byte cache line, MESI coherence protocol
HMC	8GB cube, 32 vaults, 512 DRAM banks [16] $t_{CL} = t_{RCD} = t_{RP} = 13.75\ ns$, $t_{RAS} = 27.5\ ns$ [31] 4 links per package, 120GB/s per link [16]
Benchmark Dataset	GraphBIG benchmark suite LDBC graph (1M vertex) [32], ∼900 MB footprint Bitcoin graph, ∼10 GB footprint Twitter graph, ∼5 GB footprint

B. Evaluation Results

In this section, we evaluate our proposed GraphPIM with three system configurations, as explained below. All results are normalized to the baseline unless otherwise stated.

- *Baseline*: This is a conventional architecture using HMC as main memory and does not utilize instruction offloading functionality.
- *U-PEI*: This configuration enables instruction offloading by following a mechanism similar to PEI [14] except that we assume perfect locality-aware offloading and ideal coherence management. In particular, all offloading requests that can hit in the cache are processed within the host processor, and the coherence between caches and HMC is assumed to incur no extra overhead; hence, this configuration shows the performance upper-bound of PEI.
- *GraphPIM*: This is our proposed instruction offloading technique, in which the atomic instructions accessing the PIM memory region bypass the cache hierarchy and are offloaded to HMC.

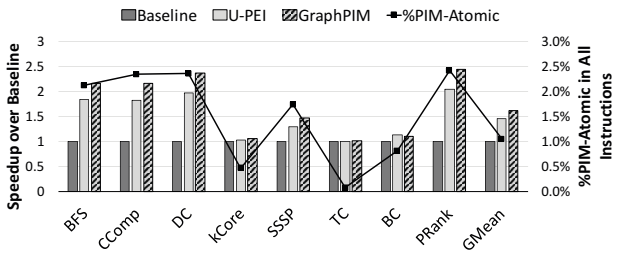

Fig. 7: Speedups over the baseline system

1) Performance Evaluation: Figure 7 shows the performance results. Compared to the baseline, GraphPIM achieves as high as a 2.4× speedup (PRank) and more than a 2× speedup for Breadth-first Search (BFS), Connected Component (CComp), and Degree Centrality (DC). On average, GraphPIM improves performance by 60% over the baseline. However, we also observe a negligible speedup for kCore Decomposition (kCore) and Triangle Count (TC). This is because they have a low percentage of offloaded PIM-Atomic operations. Thus, the performance potential is quite low to begin with. For instance, kCore spends a significant amount of time checking inactive vertices, not on accessing properties of neighbor vertices. Also, TC performs most operations within graph properties and thus is more compute intensive than other workloads.

As discussed in Section III-C, with additional support for the floating point add operation, Betweenness Centrality (BC) and Page Rank (PRank) can also benefit from GraphPIM. The result shows that PRank experiences a significant performance improvement with instruction offloading while BC does not. This is because BC has a large number of centrality computations on thread-local data structures, which makes the workload more compute intensive and the impact of improving atomic performance relatively small.

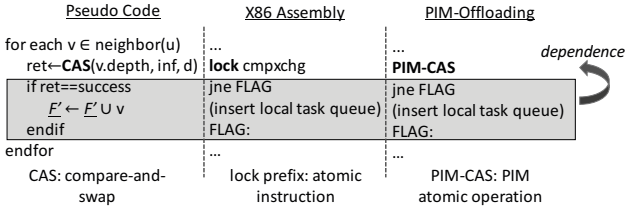

Fig. 8: Illustration of atomic instruction overhead

Atomic Overhead: The major performance gain of GraphPIM comes from avoiding the host-side atomic instructions, which incurs non-negligible overhead. In graph workloads, the long latency of atomic instructions delays not only the atomic instructions, but also subsequent dependent instructions. As shown in Figure 8, the CAS operation in the pseudo code will be compiled as a lock cmpxchg instruction and then offloaded to the HMC side as a CAS-if equal operation. The following branch instruction and task queue scheduling code depend on its return value. The long latency of atomic instructions will delay the retirement of the depending instructions. The dependent instruction block would greatly reduce the efficiency of out-of-order execution and therefore cause low processor ILP.

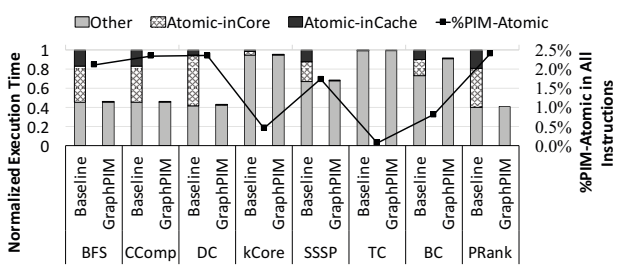

Fig. 9: Breakdown of normalized execution time (Atomic-inCore: atomic instruction cycles for pipeline freezing and write-buffer draining; Atomic-inCache: atomic instruction cycles for cache checking and coherence traffic; Other: cycles of other instructions' execution and stall)

To estimate the impact of atomic instruction overhead, we measure the breakdown of atomic and non-atomic instructions in total execution time, as shown in Figure 9. Note that atomic

instructions incur not only in-core atomic overhead but also cache checking and coherence traffic overhead. As shown, in the baseline system, most workloads spend a large portion of their execution time in atomic instructions. For example, BFS, CComp, DC, and PRank all show above 50% atomic instruction overhead. However, in kCore and TC, their small atomic instruction count limits the atomic overhead. Also, in-core overhead, which includes the time for pipeline freezing and write-buffer draining, is the major source of overhead. In most workloads, in-core overhead above 30% is observed. The result also shows close to a 20% cache overhead for some workloads, which comes from the cache checking latency for irregular property data and extra coherence traffic. In GraphPIM, all workloads show an execution time for the non-atomic part similar to the baseline system except for BC, in which non-atomic part requires more execution time. This is because in BC, non-atomic instructions reuse shared data from the atomic part. Such data locality cannot be utilized with PIM offloading.

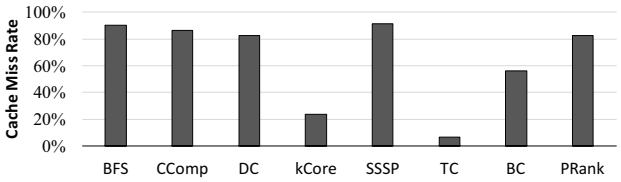

Fig. 10: Cache miss rate of offloading candidates

Cache Bypassing: We previously discussed that our offloading targets (i.e., graph property access) do not have data locality and that it is therefore better to bypass the cache hierarchy. The experiment results also support our discussion. From the results in Figure 7, we can see that GraphPIM outperforms U-PEI because it avoids the unnecessary cache checking time. Note that U-PEI is already an idealized configuration, which ignores the extra overhead of two key factors: (1) maintaining offloaded data coherence and (2) computing offloaded instructions that hit in the cache. In a more realistic system, both factors can bring significant performance overhead. Nevertheless, GraphPIM still shows a 20% higher speedup than U-PEI on average over the baseline. All workloads achieve better performance except for BC, in which thread-local data structures are heavily used with data locality. In addition, a cache analysis is also performed. In most workloads, more than 80% of offloading candidates are cache miss, as shown in Figure 10, which justifies the feasibility of GraphPIM's cache policy. kCore, TC, and BC show relatively lower cache miss rates than others. However, the performance of kCore and TC is not harmed because of their limited number of accesses, whereas data locality in BC affects performance and results in a slightly better speedup for U-PEI.

Functional Units: An 8GB HMC contains 32 vaults, each of which has 16 memory banks. Thus, 16 functional units (FUs) are enough for each vault for PIM-Atomic. However, if we have fewer FUs in each vault, the bottleneck may

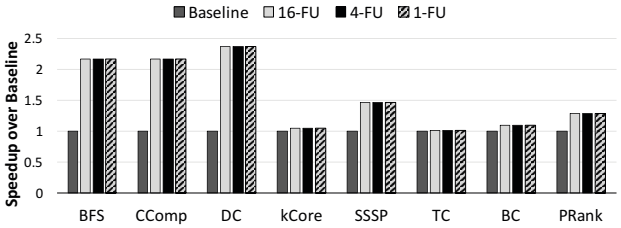

Fig. 11: Speedup over baseline system with different functional units (FU) per HMC vault

shift to the number of FUs. To estimate the impact of the number of FUs, we perform a sensitivity analysis. As shown in Figure 11, there is no noticeable performance impact with a different number of FUs. Even with only one FU in each vault, the performance is still roughly the same as with the 16-FU configuration. The result shows that the performance is not bounded by PIM-Atomic throughput in HMC. This results for two reasons: 1) HMC has 32 vaults, so the chances of consecutive HMC-Atomic requests mapped to the same vault are low. 2) The offloading target has depending instructions, which also introduce a large number of interleaving memory requests. This makes PIM-Atomic relatively sparse in total memory requests. For current atomic support in HMC 2.0, the FUs consume only negligible energy even with 16 FUs per vault. However, if floating point units are incorporated, the FU number can have a substantial impact on energy consumption. We further explain this in Section IV-B4. In our evaluation, we assume 16 regular FUs and one floating point FU per vault.

2) Bandwidth Analysis: HMC uses a packet-based protocol for the links between the host processor and HMC, where the packets consist of 128-bit data units called FLITs [16]. The packet sizes of regular memory requests and atomic operations is summarized in Table V. A 64-byte READ/WRITE request consumes 6 FLITs in total, while the atomic operations need only 3 or 4 FLITs. Therefore, PIM-Atomic provides the benefit of bandwidth savings compared to regular memory requests due to its smaller packet size.

Figure 12 shows the breakdown of bandwidth consumption normalized to the baseline. We can see that GraphPIM reduces the bandwidth consumption by around 30% in BFS, CComp, DC, SSSP, and PRank. Because graph workloads are more intensive in read requests, most bandwidth savings are from the response part. Besides, the bandwidth impact of GraphPIM in kCore and TC is negligible because of their limited number of offloaded operations. Similarly, the bandwidth benefit of

TABLE V: HMC memory transaction bandwidth requirement in FLITs (FLIT size: 128-bit)

Type	Request	Response
64-byte READ	1 FLITs	5 FLITs
64-byte WRITE	5 FLITs	1 FLITs
add without return	2 FLITs	1 FLITs
add with return	2 FLITs	2 FLITs
boolean/bitwise/CAS	2 FLITs	2 FLITs
compare if equal	2 FLITs	1 FLITs

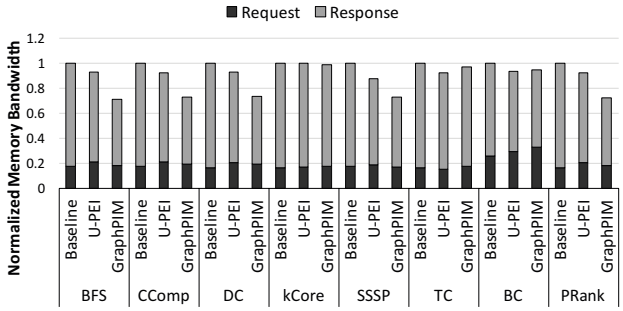

Fig. 12: Normalized bandwidth consumption with request/response breakdown

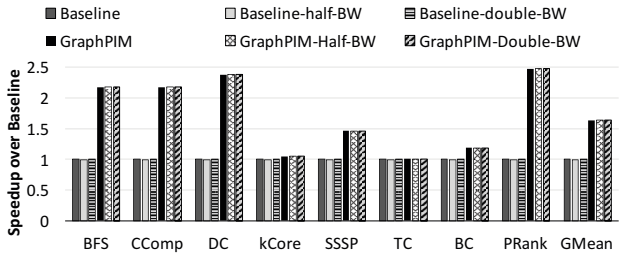

Fig. 13: Speedup over baseline system with different HMC link bandwidth

BC is offset by the existence of data locality. In addition, compared to U-PEI, the cache bypassing policy of GraphPIM can help reduce bandwidth consumption for most workloads. For example in BFS, the bandwidth reduction is further improved from 7% to 29%. However, outliers also exist. In BC and TC, because of their data locality, GraphPIM shows slightly higher bandwidth consumption than U-PEI.

Although the reduction in memory bandwidth consumption can result in energy benefits, it is not the case in the context of performance. Figure 13 shows the speedup with different HMC link bandwidths over the baseline system with original link bandwidth. As shown, since Baseline-half-BW and Baseline-double-BW are almost the same as Baseline, we can conclude that the baseline system is not sensitive to bandwidth variations. Likewise, the speedup of GraphPIM remains the same with different HMC link bandwidth configurations. From the results, we can observe that with the existing rich bandwidth resources of HMC, graph workloads are insensitive to bandwidth variations and therefore bandwidth savings cannot be effectively translated into performance gains.

3) Sensitivity on Graph Size: In graph computing, input data have a significant impact on the applications' behaviors, especially for the data access pattern. To study the impact of input graph data on performance, we perform experiments using the LDBC synthetic graph [32] with four different graph sizes, from 1K vertices to 1M vertices. The dataset details are summarized in Table VI. The four graphs share the same connectivity feature but with different memory footprints.

Figure 14(a) shows the performance improvement of GraphPIM over U-PEI. As explained previously, the offloading target in our method is graph property access, which does not have

TABLE VI: Experiment datasets

Name	Vertex #	Edge #	Footprint
LDBC-1M	1M	28.8M	∼900 MB
LDBC-100k	100K	2.8M	∼100 MB
LDBC-10k	10K	296K	∼10 MB
LDBC-1k	1K	29K	∼1 MB

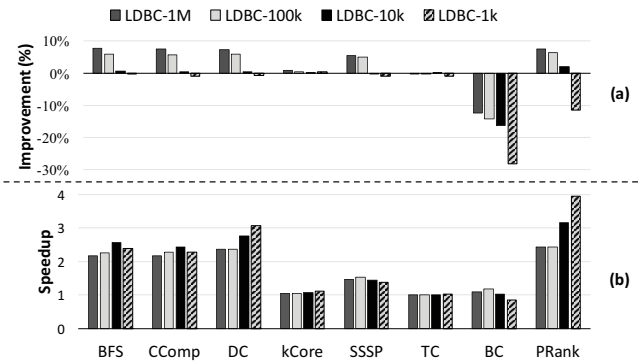

Fig. 14: (a) GraphPIM performance improvement over U-PEI (b) GraphPIM speedup over baseline

data locality. Therefore, it is more desirable to bypass the cache hierarchy for the offloaded operations. However, such a conclusion may be changed depending on the input data size. From the results in Figure 14(a), we can see that the benefit of cache bypassing decreases with smaller graph size. In some workloads, U-PEI starts to show better performance with the LDBC-10k graph. This is because the data size starts to fit into the L3 cache capacity. The degradation introduced by cache bypassing becomes much more obvious for the LDBC-1k graph. In BC, cache bypassing is always worse because of its data locality, and similarly a smaller graph brings more performance degradation.

Although the benefit of cache bypassing varies with the data size, GraphPIM still keeps a desirable overall performance gain. As shown in Figure 14(b), the speedup of GraphPIM does not vary as much as in previous improvement results. This is because our method still can reduce significant atomic instruction overhead, which is less sensitive to the data size. Moreover, in most workloads, LDBC-10k shows a better speedup than much larger graphs. This is because the atomic instruction density of graph workloads stays at a similar level with different graph sizes, while other components, such as task scheduling, are reduced for smaller graphs. Thus, sometimes smaller graphs show even better speedup.

4) Energy Analysis: GraphPIM can save data transfer energy by reducing memory traffic. However, the PIM operation in HMC may also incur extra energy consumption. To estimate such a potential tradeoff, we perform an analysis of uncore energy consumption in this section. Figure 15 shows the normalized uncore energy breakdown. We model energy consumption of caches using CACTI 6.5 [33] and compute the energy of HMC SerDes links, DRAM layers, and functional units using the energy models from prior works [34]–[36]. HMC uses four high-speed Serializer/Deserializer (SerDes)

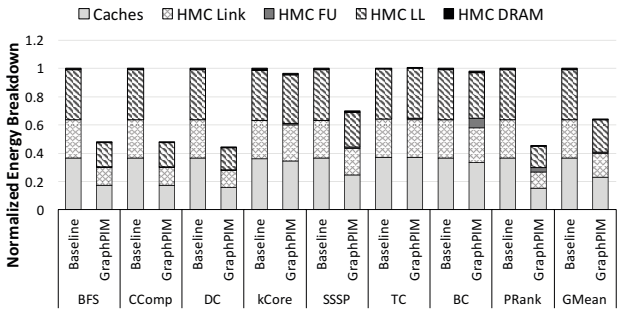

Fig. 15: Breakdown of uncore energy consumption normalized to baseline (Caches: Host cache hierarchy; HMC Link: SerDes and data transfer; HMC FU: Functional units; HMC LL: HMC logic layer; HMC DRAM: HMC DRAM dies)

links to provide high bandwidth. However, they consume nearly 43% of HMC's power [34], [36].

As shown, GraphPIM reduces the uncore energy consumption by 37% on average. The energy savings mainly come from caches, HMC links, and the HMC logic layer. This is because of the reduction of cache accesses in the host side and the reduction of memory bandwidth consumption, which saves the energy of data transfers via HMC SerDes links. Moreover, GraphPIM improves performance significantly. The shorter execution time also helps reduce the uncore energy.

From the results, we can also observe that the energy consumption of HMC comes mostly from the links and logic layer. HMC FUs consume negligible energy in most workloads except for BC and PRank, in which floating point computation introduces relatively higher FU energy even though GraphPIM follows a low-power design of floating point units and enables only one floating point unit per vault. As explained in a previous section, the performance of graph workloads is not sensitive to variations in FU number. Thus, it is more desirable to incorporate only one floating point FU per vault.

In general, GraphPIM achieves a substantial uncore energy reduction over the baseline. Even in the worst case, GraphPIM does not exceed the uncore energy consumption of the baseline. Note that besides uncore energy, we also evaluate the overall system energy and observe the same trend as the performance speedup result because of GraphPIM's significant reduction in execution time.

5) Real-World Applications: In real-world graph computing, large-scale graphs need to be processed with a complex combination of algorithms. To estimate the benefit of GraphPIM in real-world scenarios, we perform experiments with the following two real-world applications.

(1) Financial Fraud Detection (FD): This application is a graph-based financial fraud detection system, which detects first-party bank fraud and money laundering behaviors. It does graph traversal-based computations to uncover fraud rings in data relationships [37]. In our experiment, the input data is a *Bitcoin* transaction graph [38], in which each vertex represents a Bitcoin account and each edge represents a Bitcoin transaction. The Bitcoin graph contains 71.7M vertices and 181.8M edges with around a 10 GB memory footprint.

(2) Recommender System (RS): This application provides product/service recommendations for e-commerce customers. It follows an *item-to-item collaborative filtering* method [39], which is also applied in the Amazon recommender system [2]. The experiment uses a *Twitter* graph as input data [40]. It represents the friendship/followership between Twitter users. The graph contains 11M vertices and 85M edges, leading to around a 5GB memory footprint.

TABLE VII: Experiment configuration

Item	Description
Platform	Intel Xeon E5-2620, 2.3 GHz 2 sockets×6 cores×2 threads, 124GB memory 32KB/256KB private L1/L2, 15MB shared L3
Application	Financial fraud detection (FD) Recommender system (RS)
Dataset	Bitcoin graph, ∼10 GB memory footprint Twitter graph, ∼5 GB memory footprint

TABLE VIII: Real-world application experiment results

Type	Event	FD	RS
Performance Counter	IPC	0.1	0.12
	LLC MPKI	21.3	20.6
	LLC hit rate	2.8%	13.4%
	Uncore time	65.8%	52.7%
	Backend stall	83.8%	88.8%
	%PIM-Atomic	1.3%	2.9%
Analytical Model	Total host overhead	17%	32%
	Total cache checking	7%	17%

Because the application size exceeds the capability of architectural simulations, we perform real machine experiments by collecting hardware performance counters. The architectural results are then generated via an analytical model. We summarize the experiment configuration in Table VII and the experiment results in Table VIII.

In our analytical model, the execution cycles per instruction (CPI) is split into two components, atomic and other non-atomic instructions. Although both components may share overlapping cycles because of the out-of-order execution, the overlap is expected to be relatively small compared to the long latency of atomic instructions. Meanwhile, in the baseline system, atomic instructions always pay the penalty of cache checking time even though their miss rates are high. Such cache checking and in-core atomic overhead are avoided in GraphPIM. The analyical model is summarized as follows.

$$CPI_{Baseline} = CPI_{other}(1 - overlap\%) + R_{atomic} \times (T_{AO} + Lat_{cache} + Miss_{atomic} \times Lat_{mem}) \quad (1)$$

$$CPI_{GraphPIM} = CPI_{other}(1 - overlap\%) + R_{atomic} \times Lat_{PIM} \quad (2)$$

CPI_{other}: CPI of other (non-atomic) instructions
$overlap\%$: percentage of overlapped cycles
R_{atomic}: rate of atomic instructions
T_{AO}: atomic instruction overhead
$Lat_{cache}/Lat_{mem}/Lat_{PIM}$: average cache/memory/PIM latency
$Miss_{atomic}$: miss rate of atomic instructions

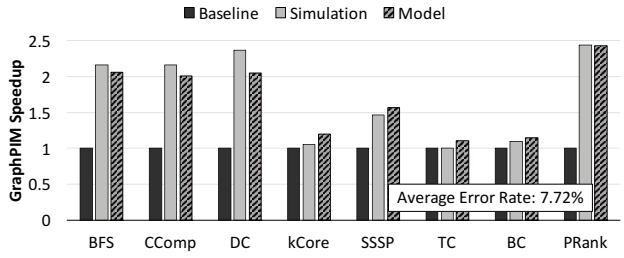

Fig. 16: Comparison between the architectural simulation and analytical model results in speedup over baseline

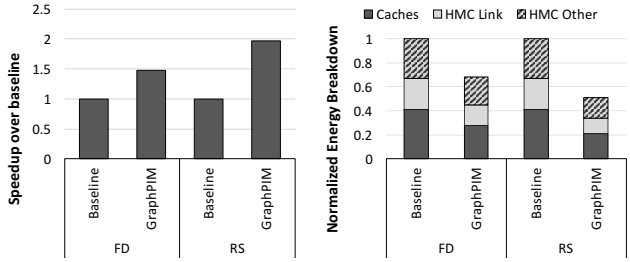

Fig. 17: Performance and energy results of two real-world applications based on an analytical model (FD: Financial fraud detection; RS: Recommender system)

Before applying the analytical model on the large-scale graph applications, we validate the correctness and accuracy of our model by comparing it with previous simulation results. As shown in Figure 16, our analytical model achieves a speedup estimation similar to architectural simulations. The error rate is within one digit in most workloads and 7.72% on average.

Figure 17 shows both performance and energy results. The energy consumption is modeled in the same way as described in the previous section. As shown in the results, GraphPIM significantly improves performance and energy consumption for both applications. The recommender system (RS) achieves as high as a 1.9× performance speedup over the baseline. Financial fraud detection (FD) also shows a 1.5× speedup. FD shows a bit lower performance benefit because it contains multiple non-graph computing components, which offset the overall benefit of GraphPIM. The energy comparison is also shown in Figure 17. GraphPIM achieves 32% and 48% energy reductions in FD and RS, respectively. The energy reduction comes from multiple factors, including cache hierarchy, data link, HMC logic, and DRAM. From the experiments of real-world applications, we can see that GraphPIM can still achieve satisfactory improvement in both performance and energy for complex real-world applications with large-scale graph data.

V. Related Work

Recent advances in die-stacking technology have made researchers investigate its use cases in computing systems. There have been several prior studies exploiting the technology for die-stacked memory [41]–[45], but beyond that, die-stacking technology can also facilitate implementing the processing-in-memory (PIM) concept, which was envisioned decades ago [8]–[11]. To our knowledge, this is the first work that explores utilizing processing-in-memory based on an industrial specification (i.e., HMC 2.0).

Most of the recent near-data processing (NDP) works focused on fully-programmable cores for in-memory processing. Gao et al. proposed an NDP architecture for data analytics applications [12]. Ahn et al. proposed an in-memory accelerator for graph processing [13]. Hsieh et al. [46] proposed a PIM architecture that enables programmer-transparent NDP GPU systems. Fully-programmable PIM offers great flexibility but introduces non-negligible hardware complexity. In addition to fully-programmable PIM, PIM taxonomy also includes fixed-function in-memory processing [47], among which HMC 2.0 is one of the examples of industrial proposals.

To our knowledge, the most relevant work that exploits fixed-function PIM is PEI [14]. PEI implements a set of PIM operations in both host processor and memory, and user applications take advantage of PIM by using special PIM instructions. The PIM operations will either be processed in the host processor or be offloaded to memory based on the locality monitoring result. As PEI does not bypass cache for PIM data, it needs to write back the data for the offloaded operations to ensure data coherence. Nai and Kim [48] also presented a case study of instruction-level PIM offloading for graph applications. As a preliminary study, it provides analysis without detailed performance evaluations.

Compared to prior research, we demonstrate that the shared data between cores with CPU *atomic operations* should be the first offloading target (rather than naively offloading the operations that would experience cache misses). Such an observation has not been well pointed out in the previous studies. In particular, compared to PEI, our work does not require any efforts from application programmers to take advantage of PIM, and provides higher performance with much less software and hardware complexity.

As discussed in Section II-C, atomic overhead is one of the major bottlenecks for graph computing. There have been multiple research efforts to reduce the overhead of locks and critical sections by accelerating critical sections [49] or by using Speculative Lock Elision (SLE) [50] and Transactional Memory [51]. However, our graph workloads use lock-free programming techniques, which are widely adopted in modern DB and OS designs. Because there are no locks, prior techniques on the critical section optimization are not directly applicable. Although we can replace existing codes with lock-based designs and enable SLE/TM optimizations, the performance would still be worse than the lock-free design because of the overhead of lock/unlock/SLE operations [52], [53]. Prior research also targets reducing application-specific performance overhead introduced by atomic operations. For example, Lee et al. proposed BSSync [54] to alleviate the long latency and serialization overhead caused by atomic operations in parallel machine learning (ML) applications. It demonstrates that offloading atomics to PIM can significantly improve the performance of parallel ML workloads using the stale synchronous parallel model, but the proposed technique is not directly applicable to our graph workloads.

VI. Conclusion

In this paper, we present GraphPIM, a full-stack solution that enables PIM instruction offloading for graph computing. GraphPIM utilizes the atomic operations specified in a real-world PIM proposal, HMC 2.0 specification, and enables PIM offloading in a non-intrusive way without requiring application programmers' effort or ISA changes. It needs only minor extensions to the host processor and the graph framework. GraphPIM is based on the key observation that the atomic access to the graph property is the main culprit for the inefficiency of graph workloads. Thus, by offloading the atomic operations on the graph property, GraphPIM avoids the overhead of executing atomic instructions in the host processor and the inefficiency caused by irregular data accesses. Our evaluation shows that GraphPIM achieves up to a 2.4× speedup and a reduction of 37% in energy consumption for a wide range of graph benchmarks and real-world applications. By incorporating architectural innovations in a practical way, GraphPIM presents a promising solution to overcome the bottlenecks of graph computing on modern systems.

Acknowledgements

The authors thank the anonymous reviewers for their valuable comments. We also thank Sandia National Labs for providing the SST/VaultSim framework. We gratefully acknowledge the support of National Science Foundation XPS 1337177 and XPS 1533767.

References

[1] C.-Y. Lin *et al.*, "Social network analysis in enterprise," *Proceedings of the IEEE*, vol. 100, no. 9, 2012.
[2] G. Linden *et al.*, "Amazon.com recommendations: item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, 2003.
[3] T. Aittokallio and B. Schwikowski, "Graph-based methods for analysing networks in cell biology," *Briefings in bioinformatics*, vol. 7, no. 3, 2006.
[4] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *SIGMOD*, 2010.
[5] I. Tanase *et al.*, "A highly efficient runtime and graph library for large scale graph analytics," in *GRADES*, 2014.
[6] A. Kyrola *et al.*, "GraphChi: Large-scale graph computation on just a PC," in *OSDI*, 2012.
[7] Y. Low *et al.*, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *VLDB*, 2012.
[8] M. Gokhale *et al.*, "Processing in memory: The Terasys massively parallel PIM array," in *IEEE Computer*, vol. 28, 1995.
[9] M. Hall *et al.*, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *SC*, 1999.
[10] Y. Kang *et al.*, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD*, 1999.
[11] D. Patterson *et al.*, "A case for intelligent RAM," *IEEE Micro*, vol. 17, no. 2, 1997.
[12] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT*, 2015.
[13] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[14] J. Ahn *et al.*, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA*, 2015.
[15] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips*, 2011.
[16] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 2.0," 2014.
[17] L. Nai *et al.*, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *SC*, 2015.
[18] S. Hong *et al.*, "Efficient parallel graph exploration on multi-core CPU and GPU," in *PACT*, 2011.
[19] S. Lynch, *Introduction to Applied Bayesian Statistics and Estimation for Social Scientists*. Springer New York, 2007.
[20] R. E. Neapolitan, *Probabilistic Methods for Bioinformatics: With an Introduction to Bayesian Networks*. Morgan Kaufmann, 2009.
[21] H. Schweizer *et al.*, "Evaluating the cost of atomic operations on modern architectures," in *PACT*, 2015.
[22] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2015.
[23] A. Yasin, "A top-down method for performance analysis and counters architecture," in *ISPASS*, 2014.
[24] P. Kumar *et al.*, "Analyzing consistency issues in hmc atomics," in *MEMSYS*, 2016.
[25] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.
[26] S. Lee *et al.*, "Feedback directed optimization of TCMalloc," in *MSPC*, 2014.
[27] *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A:System Programming Guide*, Intel Corporation, 2015.
[28] A. F. Rodrigues *et al.*, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, 2011.
[29] H. kim *et al.*, "MacSim: A CPU-GPU heterogeneous simulation framework user guide," Georgia Institute of Technology, 2012.
[30] P. Rosenfeld *et al.*, "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, 2011.
[31] G. Kim *et al.*, "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*, 2013.
[32] O. Erling *et al.*, "The LDBC social network benchmark: Interactive workload," in *SIGMOD*, 2015.
[33] N. Muralimanohar *et al.*, "CACTI 6.0: A tool to understand large caches," 2009.
[34] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *VLSIT*, 2012.
[35] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[36] S. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads," in *ISPASS*, 2014.
[37] G. Sadowksi and P. Rathle, "Fraud detection: Discovering connections with graph databases," Neo Technology, Inc., Tech. Rep., 2015.
[38] D. Ron and A. Shamir, "Quantitative analysis of the full bitcoin transaction graph," in *FC*. Springer, 2013.
[39] B. Sarwar *et al.*, "Item-based collaborative filtering recommendation algorithms," in *WWW*, 2001.
[40] R. Zafarani and H. Liu, "Social computing data repository at ASU," *Arizona State University*, 2009.
[41] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, 2008.
[42] J. Sim *et al.*, "A mostly-clean DRAM cache for effective hit speculation and self-balancing dispatch," in *MICRO*, 2012.
[43] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *MICRO*, 2012.
[44] J. Sim *et al.*, "Resilient die-stacked DRAM caches," in *ISCA*, 2013.
[45] J. Sim *et al.*, "Transparent hardware management of stacked DRAM as part of memory," in *MICRO*, 2014.
[46] K. Hsieh *et al.*, "Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems," in *ISCA*, 2016.
[47] G. H. Loh *et al.*, "A processing-in-memory taxonomy and a case for studying fixed-function PIM," in *WoNDP*, 2013.
[48] L. Nai and H. Kim, "Instruction offloading with HMC 2.0 standard: A case study for graph traversals," in *MEMSYS*, 2015.
[49] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.
[50] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *MICRO*, 2001.
[51] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.
[52] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA*, 2002.
[53] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, 1991.
[54] J. H. Lee *et al.*, "BSSync: Processing near memory for machine learning workloads with bounded staleness consistency models," in *PACT*, 2015.