2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED) | 978-1-6654-3922-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISLPED52811.2021.9502483

CoPIM: A Concurrency-aware PIM Workload Offloading Architecture for Graph Applications

Liang Yan[1, 2], Mingzhe Zhang[1, 2], Rujia Wang[3], Xiaoming Chen[1, 2], Xingqi Zou[1, 2], Xiaoyang Lu[3]
Yinhe Han[1, 2], Xian-He Sun[3]
[1]Center for Intelligent Computing Systems, Institute of Computing Technology,Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences, Beijing
[3]Department of Compute Science, Illinois Institute of Technology, Chicago, IL
Email: {chenxiaoming, yinhes}@ict.ac.cn, sun@iit.edu

***Abstract*—Processing-in-Memory (PIM) is considered a promising solution to improve the performance of graph-computing applications by minimizing the data movement between the host and memory. Which workload to offload and how to offload it to PIM logic determine whether the PIM architecture is well utilized. Offloading too much or too little workload from the host processor to the PIM side could hurt overall performance. On the other hand, the offloading granularity needs to be representative without losing generality. In this paper, we present CoPIM, a novel PIM workload offloading architecture that can dynamically determine which portion of the graph workload can benefit more from PIM-side computation. CoPIM focuses on the loop code blocks of graph applications and evaluates the necessity of offloading based on a concurrent memory access model. We also provide detailed architectural designs to support the offloading. In this way, CoPIM reduces the size of offloading instructions and also improves the overall performance with less energy consumption. The experimental results show that compared with other state-of-the-art PIM workload offloading frameworks, CoPIM achieves a speedup by the geometric mean of 19.5% and 11.4% than PEI and GraphPIM, respectively. On the other hand, CoPIM also reduces the un-core energy consumption by 6.8% and 6.5% on average over PEI and GraphPIM, respectively.**

***Index Terms*—Processing-in-memory, hybrid memory cube, workload partitioning**

I. Introduction

With the rapid development of information technology such as big data, cloud computing, and artificial intelligence, information processing has shifted from computing-intensive to data-intensive. The conventional von Neumann architecture faces the memory wall bottleneck [1]. The challenges mainly come from two aspects: first, the frequent data transmission between computing units and memory units has become the bottleneck of power dissipation and system performance [2]; second, with the increase of data size [3], the effectiveness of the conventional memory hierarchy based architectures utilizing data locality decreases or even becomes invalid.

In terms of technology, with the development of the 3D-stacking and Through Silicon Vias techniques, a new solution for the memory wall problem was proposed: the Processing-in-Memory (PIM) architecture. PIM integrates computing resources into the memory to make full use of the near-data advantage [4], [5]. PIM reduces unnecessary data movement between the on-chip caches and the memory, thus significantly improving memory access's energy efficiency. In terms of application, modern big data applications run on massive data sets, and the data movement is significant. Among them, graph-computing applications are particularly popular, because graph naturally captures the relationship between data items, and allows data analysts to obtain valuable insights from patterns in data for wide application [6]. However, graph-computing brings great challenges to memory system due to the random memory access. As a result, because of the advances in both technology and application, the research community and industry become increasingly interested in applying PIM to graph-computing applications.

However, how to divide the graph-computing program reasonably and select the proper instructions for PIM has become a significant challenge. An unreasonable workload offloading will lead to frequent data movements between CPU and memory, resulting in performance and energy overhead. When the in-memory computing units are configured as general-purpose cores, the computation partitioning will have a greater impact. Therefore, it is necessary to establish an accurate and effective workload offloading model to improve the performance and energy efficiency of the PIM system.

Most of the partitioning strategies aim to move highly data-intensive portions of the application to PIM logic units. However, some studies illustrate that certain codes can still benefit from the host CPU due to the performance gap between the CPU and the PIM logic [7]. Therefore, an efficient workload offloading architecture should take into account the performance differences between the CPU and PIM core to maximize the efficiency of the overall system.

Intuitively, an instruction that is suitable for PIM execution should involve a cache miss. This idea has been adopted by [4].

This work was supported by National Key Research and Develop Program of China (2018YFA0701500), by Key Research Program of Frontier Sciences, Chinese Academy of Sciences (ZDBS-LY-JSC012), by Strategic Priority Research Program of Chinese Academy of Sciences (XDB44000000), by National Natural Science Foundation of China (61804155, 61834006), by the Youth Innovation Promotion Association CAS, and by Beijing Academy of Artificial Intelligence (BAAI), and by ZheJiang Lab (2019KC0AB01).

978-1-6654-3922-0/21/$31.00 ©2021 IEEE

However, a modern memory system is supported not only by memory hierarchy but also by various data access concurrency. Techniques that exploit memory concurrency or data parallelism, such as out-of-order execution and non-blocking cache, have been applied in modern systems to effectively overlap computation and memory access. Concurrent memory access enables multiple memory requests to overlap to reduce the memory stall time. Cache misses can be divided into two different types considering memory concurrency: mixed misses (hit/miss overlapping) and pure misses [8] (no hit/miss overlapping). A mixed miss can reduce the performance loss because the processor can still perform execution from an overlapped cache hit. On the other hand, a pure miss with no hit-miss overlapping could hurt the overall performance more. As a result, a pure miss will indeed induce performance loss in the modern memory system with the consideration of concurrency [8], [9].

As discussed, most of the existing PIM workload partitioning strategies did not fully consider the memory access concurrency. There remains space exploration to find an accurate criterion to define which part of the code will cause the CPU to stall and so that we should move them into PIM. This paper aims to find an approach to wisely partition the code between CPUs and PIM processors, which can transfer codes that indeed need to be executed in memory with the consideration of memory concurrency using last level cache (LLC) pure miss cycle rate. We make the following contributions in this paper.

- We utilize the concept of pure miss in the cache hierarchy, which considers both access concurrency and locality. We identify that using pure miss and its related metrics can help find critical parts of the workload that should be executed on the PIM for better performance.
- We then propose CoPIM, a concurrency-aware PIM workload partition and offload architecture that integrates memory access concurrency measurement and the workload source code sampling. We present the detailed workflow and architectural extensions of CoPIM.
- We compare CoPIM with other state-of-the-art PIM code partition frameworks and show the improvements with detailed experimental results. We show that we can achieve higher performance with less workload offloading and energy consumption.

II. Background and Motivation

A. Processing-in-Memory

To mitigate the memory wall bottleneck, the new computing paradigm, PIM, proposes performing the computation directly inside or near the memory modules. Researchers have proposed various PIM designs and showed significant performance improvement for applications like data analytic [10], graph processing [6], [11], DNN training [12], etc.

In general, the PIM cores are much weaker in performance than CPU, as they lack large caches and sophisticated instruction-level parallelism (ILP) techniques. As a result, the workload to be executed by PIM needs to be appropriately chosen. Offloading too much or an unsuitable workload to the PIM side could hurt the overall system performance.

As for the partitioning approach, there are already some typical partitioning methods. For example, PEI [4] considers the locality of the program and uses LLC miss as the standard of code partitioning. GraphPIM [6] demonstrates the performance benefits for graph applications by offloading the atomic operations to PIM. Nonetheless, concurrent memory access is not fully taken into account in both PEI and GraphPIM as far as we know, so it may result in unnecessary code being transferred into PIM. We will then present why and how concurrent memory access could change and impact overall system performance.

B. Concurrent Memory Access Model

Average memory access time (AMAT) is a standard metric to analyze memory system performance. An implicit assumption of AMAT is, the memory only supports sequential accesses and do not have multiple accesses coinciding. A recent performance model, Concurrent-AMAT (C-AMAT) [8], [9], was developed to extend with the consideration of concurrent memory access. C-AMAT provides a more accurate analysis of modern memory systems where concurrent memory accesses are common.

One of the important concepts introduced by C-AMAT is *pure miss*. *pure miss* contains at least one miss cycle which does not have any hit access overlapped with. Due to concurrent memory access, during miss cycles overlapped with one or more hit cycles, the processor can still work. However, if a miss cycle has no hit to overlap with, it can severely hurt the performance. Therefore, reducing pure misses, as well as pure miss cycles, can significantly improve the overall performance.

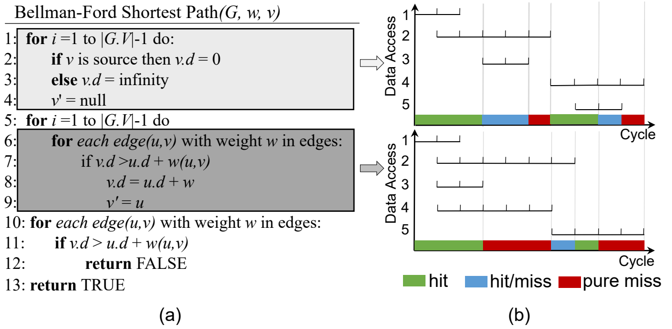
Based on the insights from the C-AMAT model, we use the *pure miss cycle rate of LLC (θ)* to describe the cache efficiency of a loop code block in graph-computing applications when considering concurrent misses:

$$\theta = \frac{\#\ of\ LLC\ Pure\ Miss\ Cycles}{\#\ of\ Total\ CPU\ Cycles}. \tag{1}$$

The higher the θ value, the greater the probability that this code block would introduce a longer CPU stall time. To maximize performance, we can offload the code block with a high θ value into PIM logic to relieve the pressure on memory accesses and reduce the overall execution time.

C. Motivation

Due to the performance difference between CPU and PIM processors, migrating too many tasks to the PIM side for execution may lead to a decline in overall performance. So the room for improvement of PIM workload partitioning is to find the most critical factor that causes the performance degradation to judge the code block whether needs to be transferred to PIM or not. It has been suggested in the C-AMAT model that pure miss becomes the critical factor that could most hurt the performance. As illustrated in [8], [9], memory



```
Bellman-Ford Shortest Path(G, w, v)
1:  for i =1 to |G.V|-1 do:
2:      if v is source then v.d = 0
3:      else v.d = infinity
4:      v' = null
5:  for i =1 to |G.V|-1 do
6:      for each edge(u,v) with weight w in edges:
7:          if v.d >u.d + w(u,v)
8:              v.d = u.d + w
9:              v′ = u
10: for each edge(u,v) with weight w in edges:
11:     if v.d > u.d + w(u,v)
12:         return FALSE
13: return TRUE
```

Fig. 1. Example of workload partitioning in CoPIM. (a) Code snippet of Bellman-Ford Shortest Path algorithm. G: graph structure; $|G.V|$: vertices; w: distance weight; v: vertex; v': predecessor of v; $v.d$: distance to v; u: intermediate node; $u.d$: distance to u; $w(u,v)$: distance from u to v. (b) Concurrent memory accesses of 2 loops.

concurrency reduces the memory stall time by overlapping multiple memory accesses. A single cache miss latency is no longer a determinant factor of the memory system.

By taking the C-AMAT model's inspirations, we find that code blocks with the higher pure miss cycle rate are the most potent part of an graph-computing application to be transferred to PIM. These could bring two benefits: first, the pure miss cycle rate is considered a more accurate metric to determine the critical memory misses so that we can achieve more optimal performance under this guidance; second, we can reduce unnecessary code migration, leaving more work executed on the host CPUs with higher performance.

III. CoPIM Framework

A. Overview

Previous works have used different granularities for PIM workload offloading. For example, PEI [4] offloads workload at the granularity of instruction. It is much more accurate than other methods, but one consideration is that the overhead of the frequent judgment of PIM instructions will be high. Some other works choose to offload the application under the code block granularity. For example, Prometheus [13] uses Low Level Virtual Machine (LLVM) to partition the application into many different code blocks. It examines the relationship between these blocks and the memory access characterization of the code blocks to establish the partitioning strategy. Compared with instruction-level offloading, this method reduces the frequent decision of offloading targets and reduces the overhead accordingly. Nevertheless, it takes too much time for large programs to use LLVM to obtain dynamic traces and may transfer too many code blocks to PIM.

In this work, we select the *loop body* in graph-computing applications as a code block to do the workload offloading. There are three reasons for offloading in the granularity of loop body: (1) Loop bodies account for a large proportion in graph-computing application and can easily be located. (2) The ubiquitous loop bodies often become the focus of optimization due to frequent memory access operations. (3) To avoid the significant overhead of the PIM target judgment compared with finer granularity such as instruction.

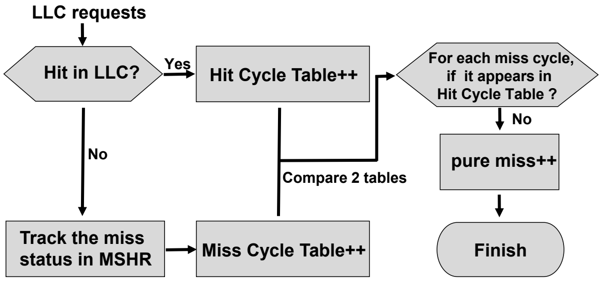


Fig. 2. Workflow of pure miss detection.

Fig. 1(a) illustrates a code snippet of the Bellman-Ford Shortest Path algorithm that computes the shortest paths from a single source vertex to all of the other vertices in a weighted graph. There are two main steps: (1) initialization (line 1-4): choose a starting vertex and assign infinity path values to all other vertices; (2) relaxation (line 5-9): better ones replace approximations to the correct distance until they eventually reach the solution. In the code snippet, we mark two loops to indicate the different memory behavior under the definition of pure miss. Fig. 1(b) shows the concurrent memory access schematic diagram of 2 marked loops. The first loop is for initialization, which causes two pure miss cycles; the second loop is for relaxation, which causes five pure miss cycles. The different memory access behaviors mainly come from the loops' different operation complexity. As illustrated in the example, pure miss is an effective measurement to examine memory behavior with the consideration of concurrency.

B. CoPIM Workflow

In the CoPIM workflow, the most important step is to detect the pure miss. We distinguish pure miss with the flow shown in Fig. 2. Once memory requests come into the LLC is hit, the request will be served by sending data to the register file immediately, and we will track the hit cycle in a Hit Cycle Table (HCT). Otherwise, if miss, a new Miss Status Holding Register (MSHR) entry and cache line will be reserved for this data request, and we will track this miss cycle in a Miss Cycle Table (MCT). After we get the hit and miss cycle tables of the first 5% loop iterations, we will search for each miss cycle of MCT in HCT. If the miss cycle does not appear in HCT, we define this miss cycle as a pure miss cycle. Additionally, we can measure the pure miss cycle rate defined in Section II-B.

In our work, we adopt the sampling method for the first 5% of the loop iterations on the host CPU side. Furthermore, the pure miss cycle rate (θ) is used to determine whether the rest codes of the loop are placed into memory for execution or not. The workflow of CoPIM is shown in Fig. 3. When an application comes in, CoPIM locates all the loops and puts the non-loop part directly on the host CPU for execution, thus completing a preliminary program division. As described in the prior sections, CoPIM focuses on loops, which are conditional offloading candidate code blocks. Therefore, when the program executes a loop body, we take the first 5% of the loop iterations to do sampling. We make these portions of

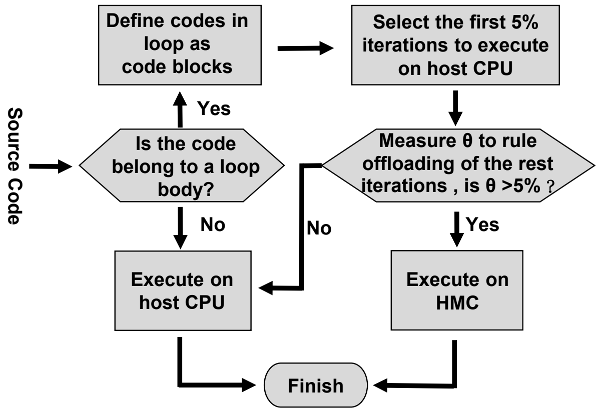

Source Code
Is the code belong to a loop body?
Yes
Define codes in loop as code blocks
Select the first 5% iterations to execute on host CPU
Measure θ to rule offloading of the rest iterations , is θ >5% ?
No
Yes
No
Execute on host CPU
Execute on HMC
Finish

Fig. 3. CoPIM workflow for selecting offloading targets.

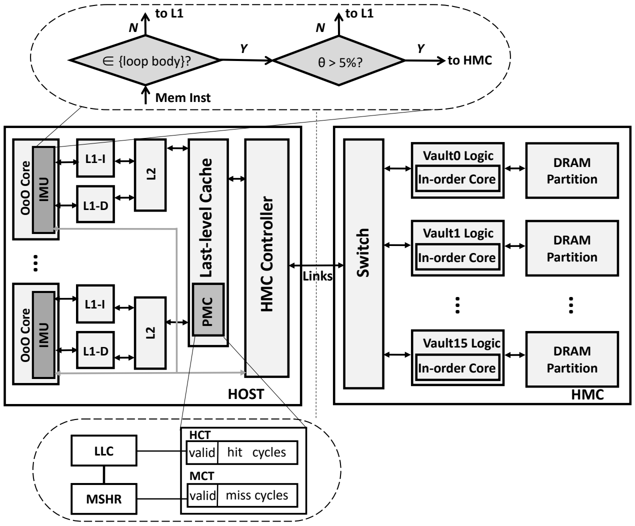


Fig. 4. CoPIM architectural designs to support the workload offloading.

loops executed on the host CPU and use the CoPIM offloading architecture to distinguish the LLC pure miss cycle rate(θ). In general, when the code block leads to a high value of θ, the memory access characteristics of the code block are indigent. If the value of θ is greater than 5%, we transfer all the rest of the loop block into memory for execution; Otherwise, we continue to execute the rest portion of the loop on the host CPU. The threshold of θ will be discussed in Section IV-C5.

CoPIM does not restrict the method to identify loops. For example, loops can be annotated manually in the source code or identified automatically by the compiler. By any loop identification method, the instruction set needs some modifications to let the CPU know which instructions are related to loops in the original source code. In CoPIM, we use a CUDA-like mechanism for loop identification. We use MACRO to mark the start and the end of the loop, then the compiler can identify the loops and translate the marks into the binary code.

C. Architecture Integration

Fig. 4 shows the architecture of CoPIM, which can be divided into the Host and Hybrid Memory Cube (HMC) parts. On the host side, we have configured 4 out-of-order (OoO) CPU cores and equipped them with three cache levels, in which L1 and L2 are private for each core, and LLC is shared. L1 is divided into the I-cache and D-cache. To keep the architectural changes non-intrusive to current hardware architectures, we only make simple modifications to L1 I-cache and L3 cache to implement the functions of CoPIM, which will be described below. The HMC side mainly contains the data switch component, the vault logic, and the stacked DRAM layers. There are 16 vaults in our work. At the same time, we equip the logic layer of each vault with a simple sequential CPU core. The host and HMC are connected by high-speed links, which typically run at several gigabits per second per bit-lane. Requests flow over the high-speed links to an interconnect that transmits them to their target vault controller. Each vault controller sends commands to write/read data to/from the memory banks in each partition. After the DRAM access completes, data will be transferred back through the interconnect to the high-speed links.

1) Pure Miss Counter: In CoPIM, We have implemented the measurement of pure miss in LLC. The measurement is realized by adding a Pure Miss Counter (PMC) unit in the LLC. We record the information of hit/miss cycles at the same time to find out the pure miss cycle. In the PMC unit, we design two tables: HCT and MCT, to record the behavior of the LLC and the cycle of hit and miss. As shown in Fig. 4, the PMC unit is implemented by connecting the LLC and the MSHR structures. The case of hit cycle can be obtained by comparing the tag of LLC. When the tag is matched, the cache is hit. All cache miss status is tracked by MSHR. When a missed request arrives at the MSHR, the MCT will also track this miss status. Furthermore, a pure miss cycle is determined by searching each miss cycle of MCT in HCT to see if this miss cycle has no hit cycles overlapped.

2) Instruction Management Unit: The program is divided into two parts: loops and the others. For the portion without loops, we directly execute these parts of code on the host CPU, while the real potential PIM execution object is the loop body. After sampling, if its pure miss cycle rate of LLC is greater than 5%, we will transfer this loop into memory to execute. As shown in Fig. 4, CoPIM integrates an Instruction Management Unit (IMU) in each host core, which determines the data path of instructions. We load the instructions that do not need to be executed in memory directly into the next level cache to complete the execution on the host CPU side. While for the instructions annotated as PIM execution, these instructions are delivered to the PIM by sending memory requests, which is shown as the grey line in Fig. 4, and the PIM side will execute these instructions in succession.

IV. Evaluation

A. System Configuration

We use `PIMSim` [14], an open-source PIM simulator based on `Gem5`, to evaluate the effectiveness of CoPIM. `PIMSim`

TABLE I
SYSTEM CONFIGURATIONS.

		Configuration
Host Side	Processor	4 Cores, OoO, 2GHz, 192-entry ROB
	Private L1 Cache	Separated 32KB I/D-Cache per core, 8-ways, 2-cycle hit latency, 8-entry MSHR, 64B line size
	Private L2 Cache	256KB/core, 8-ways, 12-cycle hit latency, 12-entry MSHR, 64B line size
	Shared L3 Cache	shared 2MB/core, 32-ways, 35-cycle hit latency, 32-entry MSHR, 64B line size
PIM Side	PIM Logic	16 PEs, in-order pipeline, 1 PE/vault, 500MHz
	Private Cache	32KB/PE, 4 ways, 64B line size
	Memory	HMC v2.1 [15], 16 vaults, 256 banks

TABLE II
BENCHMARKS AND DATA SETS.

Benchmark	Dataset
Breadth-First Search (BFS), Bellman Ford Shortest Path (SP), PageRank (PR)	p2p-Gnutella30 (36K Verticies, 88K Edges), com-DBLP (317K Verticies, 1M Edges), com-Youtube (1.1M Verticies, 2.9M Edges), wiki-Talk (2.3M Verticies, 5M Edges), soc-LiveJournal (4.8M Verticies, 6.9M Edges)

provides a variety of PIM instructions for the PIM architecture. To realize the multi-platform implementation of PIM architecture, PIMSim uses Gem5 pseudo instructions to implement PIM instructions. The detailed PIM architecture parameters are listed in Table I.

The following experimental results involve four kinds of simulation configurations (1) CPU-only: this is a conventional architecture that uses HMC as the main memory and does not offload any operations to the memory. (2) PEI [4]: during the real execution, we assume the system uses a locality-aware offloading approach to decide where the PIM operations should be executed (i.e., all requests that can incur the on-chip cache hit are served by the host processors. Otherwise, they will trigger further execution within the memory). (3) GraphPIM [6]: based on GraphPIM's observation of the source program of graph applications, it is found that accessing the graph property using atomic functionalities is easy to cause the inefficient utilization of the memory subsystem, which is suitable for PIM. (4) CoPIM: the partitioning method discussed in this paper.

B. Benchmarks and Workloads

We use several typical graph workloads as PIM applications for evaluation, which were also used in [4], [6]. We use real-world graphs to make the results more reliable. As shown in Table II, the numbers of the workloads' vertices vary from 36K to 4.8M.

C. Experimental Results

1) Percentage of code offloading: To explore why and how CoPIM improves the performance, we evaluate the percentage of instructions executed at the PIM side during the execution of the entire application when using different data sets. As shown in Fig. 5, CoPIM tends to offload 51.1% fewer instructions into PIM on average compared with PEI. Furthermore, for each application using different sizes of the data set, CoPIM all tends to offload fewer instructions than PEI. Compared with GraphPIM, CoPIM also offloads 33.0% fewer instructions on average.

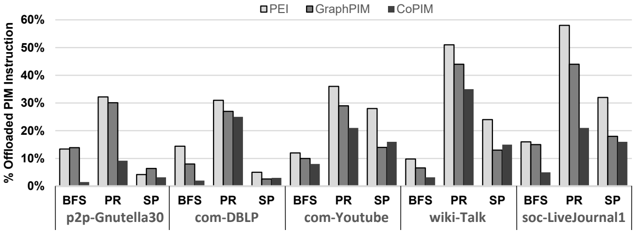


Fig. 5. Percentage of offloaded instructions into memory.

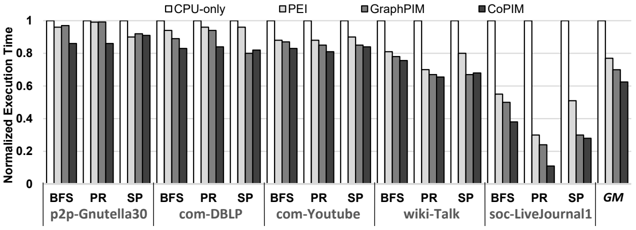


Fig. 6. Normalized performance evaluation using graphs of different sizes, *GM*: geometric mean.

2) Performance with different partitioning methods: In this paper, we use the execution time of the entire application to evaluate the different partition methods' efficiency. Fig. 6 shows the normalized execution time and geometric mean(GM) of 3 kinds of partition strategies. Generally, CoPIM achieves a 38% speedup by the geometric mean over CPU-Only. Compared with PEI, when the input size is small, such as p2p-Gnutella30, CoPIM performs 7.5% better than PEI on average. When the input size gets large, such as soc-LiveJournal1, CoPIM performs 46.4% better than PEI on average. CoPIM tends to show better performance as the data set enlarges and achieves a speedup by the geometric mean of 19.5% than PEI. Compared with GraphPIM, CoPIM achieves a speedup by the geometric mean of 11.4%.

3) Energy evaluation : Fig. 7 shows the normalized energy consumption breakdown of un-core aspects when running the BFS application with different data sets. We investigate the energy consumption considering three aspects which are caches, HMC Serializer/Deserializer (SerDes) Link and HMC-other(DRAM layer and logic layer). We model the cache using CACTI 6.0 [16]. The energy of HMC-link is considered to be 13.7pJ/bit [17]. Energy per bit is considered at 3.7 pj for the DRAM layers and 6.78 pj/bit for the logic layer [18]. As shown, CoPIM reduces the un-core energy consumption by 18% on average over CPU-only. Compared with PEI and GraphPIM, CoPIM reduces 6.8% and 6.5% energy consumption on average, respectively. The energy savings mainly come from HMC-link, and the HMC-other. This is because of the reduction of workload offloading, which saves the energy of

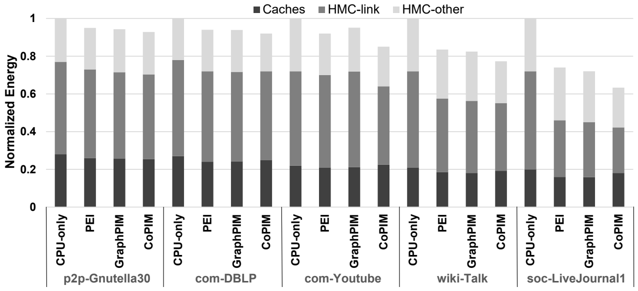


Fig. 7. Normalized energy consumption breakdown of un-core aspects.

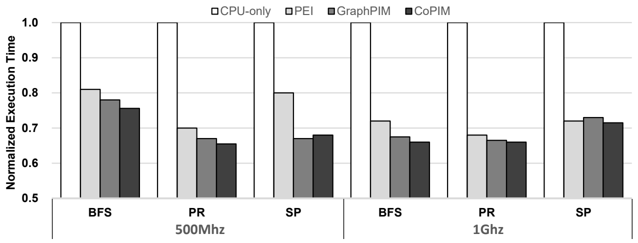


Fig. 8. Normalized Performance with different PIM processor frequencies using wiki-Talk data set.

data transfers via HMC SerDes links and the energy of DRAM and logic layer.

4) Performance with different PIM processor frequencies: As shown in Fig. 8, we evaluate the performance of 2 different PIM processors' frequencies: 500MHz and 1GHz with wiki-Talk data set. With the improvement of the PIM processors' frequency, the performance of three different offloading methods has been improved slightly. Compared with GraphPIM, CoPIM shows little difference in performance on average as the frequency increases. Compared with PEI, CoPIM achieves a 9.5% speedup on average with 500MHz PIM processors, but only 4.0 % when the PIM processors' frequency is promoted to 1GHz. Due to the lack of consideration of memory concurrency, PEI will introduce much more computation-transfer between CPU and PIM processors than CoPIM. When the PIM processors' frequency is 500MHz, more computation-transfer means the benefits of reducing the data movement can be offset by the weak performance of PIM processors.

5) Sensitivity of θ : As shown in Fig. 9, we investigate the effect of θ value on execution efficiency. We select wiki-Talk as the input data set and examine the difference in the execution time of 3 different applications when the θ value varies from 1% to 10%. The results show that the lowest execution time of BFS and PR is around 5%, and the minimum execution time of SP is when θ is 8%, but the difference between the execution time when θ sets at 8% and 5% is only 1.2%. Thus, we select 5% as the threshold of θ.

V. Conclusions

In this work, we propose CoPIM, a novel PIM graph-computing workload offloading architecture. CoPIM offloads code in the granularity of the loop code block. Based on a concurrent memory access model, CoPIM identifies the candidates for PIM acceleration during the first few iterations of a loop code block. We also provide detailed architectural designs to support the offloading. The CoPIM has been evaluated on a state-of-the-art PIM architecture with a wide range of graph-computing applications. The results show that CoPIM reduces the size of offloading instructions and also improves the overall performance. Compared with other PIM workload offloading frameworks: 1) CoPIM achieves a speedup by the geometric mean of 19.5% than PEI with 51.1 % fewer offloaded instructions on average. 2) CoPIM achieves a speedup by the geometric mean of 11.4% than GraphPIM with 33.0% fewer offloaded instructions on average.3) CoPIM reduces the un-core energy consumption by 6.8% and 6.5% on average over PEI and GraphPIM, respectively.

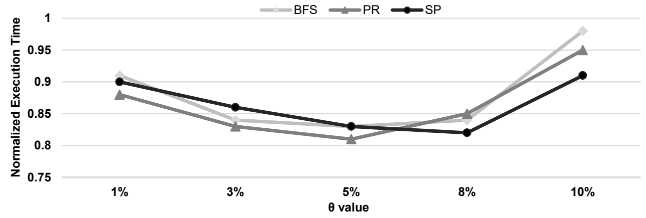


Fig. 9. Sensitivity of θ of 3 different applications with wiki-Talk data set.

References

[1] B. Rogers *et al.*, "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *ISCA*, 2009, pp. 371–382.

[2] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[3] S. Kanev *et al.*, "Profiling a Warehouse-Scale Computer," *IEEE Micro*, vol. 36, no. 3, pp. 54–59, 2016.

[4] J. Ahn *et al.*, "PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *ISCA*, 2015, pp. 336–348.

[5] M. Gao *et al.*, "Practical near-data processing for in-memory analytics frameworks," in *PACT*, 2015, pp. 113–124.

[6] L. Nai *et al.*, "Graphpim: Enabling instruction-level PIM offloading in graph computing frameworks," in *HPCA*. IEEE, 2017, pp. 457–468.

[7] A. Boroumand *et al.*, "CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators," in *ISCA*, 2019, pp. 629–642.

[8] Y. Liu and X. H. Sun, "LPM: A Systematic Methodology for Concurrent Data Access Pattern Optimization from a Matching Perspective," *IEEE TPDS*, vol. PP, no. 99, pp. 1–1, 2019.

[9] D. Wang and X. H. Sun, "APC: A Novel Memory Metric and Measurement Methodology for Modern Memory Systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1626–1639, 2014.

[10] M. Drumond *et al.*, "The mondrian data engine," in *ISCA*, 2017.

[11] Y. Zhuo *et al.*, "GraphQ: Scalable PIM-Based Graph Processing," in *MICRO*, 2019.

[12] J. Liu *et al.*, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *MICRO*, 2018.

[13] Y. Xiao *et al.*, "Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy," in *DATE*, 2018, pp. 1387–1392.

[14] S. Xu *et al.*, "PIMSim: A flexible and detailed processing-in-memory simulator," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 6–9, 2018.

[15] H. M. C. Consortium *et al.*, "HMC Specification 2.1," *Retrieved May*, 2019.

[16] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," *Bragantia*, 2009.

[17] E. Azarkhish *et al.*, "Design and evaluation of a processing-in-memory architecture for the smart memory cube," in *ICACS*, 2016.

[18] J.Jeddeloh *et al.*, "Hybrid memory cube new dram architecture increases density and performance," *Digest of Technical Papers - Symposium on VLSI Technology*, pp. 87–88, 2012.