Exploring PIM Architecture for High-performance Graph Pattern Mining

Jiya Su, Linfeng He, Peng Jiang, Rujia Wang

Abstract—Graph mining applications, such as subgraph pattern matching and mining, are widely used in real-world domains such as bioinformatics, social network analysis, and computer vision. Such applications are considered as a new class of data-intensive applications that generate massive irregular computation workloads and memory accesses, which are different from many well-studied graph applications such as BFS and page rank. In this paper, we use the emerging process-in-memory architecture to accelerate data-intensive operations in graph mining tasks. We first identify the code blocks that are best suitable for PIM execution. Then, we observe a significant load imbalance on PIM architecture and analyze the root cause for such imbalance in graph mining applications. Lastly, we evaluate several scheduling schemes that help reduce the load imbalance and discuss potential optimizations to enhance performance further.

Index Terms—Process-in-memory, Graph pattern mining

1 Introduction

Process-in-memory architecture (PIM) [1] is considered as a promising solution to enhance the performance of memory-bounded data-intensive applications. With PIM architecture, it is possible to integrate general-purpose or specialized computation units in or near the memory module. When the application and data are appropriately placed and scheduled on the PIM and host CPU, we can reduce massive data movement between the CPU and memory module to achieve high-performance and energy-efficient computation. For example, classical graph processing applications, such as BFS and page rank, have been implemented on emerging PIM architectures with software and hardware co-designs [1], [14].

Recently, *graph pattern mining* (GPMI) algorithms emerge as a new class of data-intensive applications that has attracted extensive attention from system [13], [19], [20] and architecture [2], [16], [21] domain. GPMI has many real-world use cases, such as motif extraction from gene networks [15] and pattern search over semantic data [5]. GPMI is fundamentally different from the general graph processing applications in several ways: 1) the computation involves more complex iterations which may cause load imbalance; 2) the computation involves enormous data accesses (more details in §2.2). Therefore, it is challenging to use conventional hardware, such as CPU or GPU, to accelerate the computation.

Therefore, we are motivated to examine the new class of GPMI applications and study the challenges of applying PIM architecture to such applications. We first explore the memory access characteristics of the graph matching algorithm and find the intersection and subtraction (I/S) operations are memory access intensive, which are suitable for PIM architecture. Then we compare the performance of I/S operations on CPU host and PIM to evaluate the potential performance gain. We identify that the workload distribution to PIM cores could cause significant imbalance and hurt the performance improvement from PIM architecture. We evaluate several scheduling schemes which reduce the load imbalance in selected workloads. In addition, we identify the root cause of such load imbalance regarding the input graph and patterns and propose potential schemes that can overcome such challenges.

- *J. Su and R. Wang are with the Computer Science Department, Illinois Institute of Technology, Chicago, IL. E-mail: jsu18@hawk.iit.edu, rwang67@iit.edu.*
- *L. He and P. Jiang are with the Computer Science Department, University of Iowa. E-mail: linfeng-he@uiowa.edu, peng-jiang@uiowa.edu.*

2 Background and Motivation

2.1 Process-in-Memory Architecture

Processing-in-Memory (PIM) integrates processing units inside the memory to reduce the overhead of frequent data movement. PIM can be implemented using a variety of technologies. 3D-stacking with TSVs technology is a commonly used technology for PIM due to its large bandwidth and energy advantages. Two of the most prominent 3D-stacked memory technologies today are Hybrid Memory Cube (HMC) [3] and High Bandwidth Memory (HBM) specification [9], [11], both of which consist of one logic die stacked with several DRAM dies. The PIM cores could be either implemented on the logic die [1], [4], [6], [8], [14] or in the DRAM banks [2], [11].

In this work, we assume that the PIM cores are integrated into the HMC architecture, and they can process the same ISA as the host. The host access the HMC with an external link, while the PIM cores access the HMC via internal TSVs. Note that the latest PIM module from Samsung [11] uses the HBM architecture with 128 programmable computing units. Due to the limited documentation on the new PIM interface, we follow prior research work and use the HMC interface for evaluation purposes.

2.2 Graph Mining Applications and Algorithms

Applications: In this work, we focus on the motif counting (MC) GPMI application. A motif is any connected, unlabeled graph pattern. The goal is to identify all motifs (patterns) with k vertices and count the embeddings of each of the patterns. This kernel is widely used in bioinformatics. The evaluated patterns are shown in Figure 1.

Representative algorithms: There are two available algorithms that can support GPMI applications. 1) Exhaustive-check method, used in Arabesque [19], RStream [20], Gramer [21], which explores the subgraphs to a certain size, and performs isomorphism checks to aggregate the explored subgraphs; and 2) Pattern-enumeration method, used in Automine [13], GraphZero [12], GraphPi [18], directly finds the

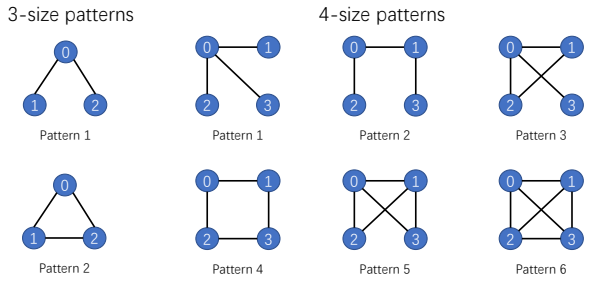

Fig. 1. Patterns with 3 and 4 vertices.

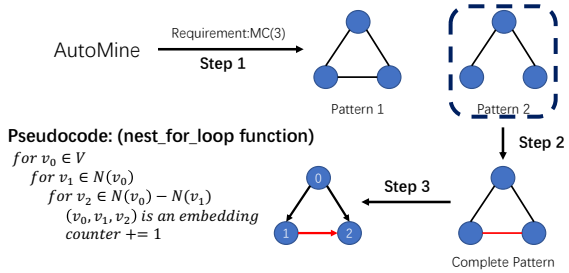

Fig. 2. Pattern enumeration with AutoMine [13] method.

subgraphs that are isomorphic to the pattern. Compared with the exhaustive-check method, the pattern-enumeration method can achieve higher performance since it eliminates the computation-intensive isomorphism tests with lots of edge-dimension random accesses and avoids checking the subgraphs not matching the pattern. AutoMine [13] outperforms RStream [20] and Arabesque [19] by several orders of magnitude on real-world graphs of different scales. Therefore, we focus on the *pattern-enumeration* algorithm for GPMI applications.

Pattern enumeration steps: Figure 2 shows the steps of pattern enumeration with AutoMine algorithm. First, it generates all patterns according to the requirements of the application (Step 1). Then, for each pattern, it first constructs a colored complete pattern graph to encode all the neighborhood relations of the vertices in the pattern (Step 2). Specifically, it paints all present edges black and adds red edges for the absent ones. Next, it assigns an order to the vertices of the pattern, and specifies the direction for the edges from small id vertices to large id vertices (Step 3). Finally, according to the vertex ids and the directed edges, we can construct a multi-layer *nest_for_loop* (pseudocode in Figure 2) to find all embeddings (also called subgraphs) that match the pattern. Each vertex in the pattern is associated with a *for* loop. The loops start from the smallest vertex id v_0. If the incoming edge (i, j) is black, which means there is an edge between vertices i and j, then vertex j belongs to the *intersection* of the neighbor sets of vertex i; if the edge is red, which means there is no edge between vertices i and j, then vertex j belongs to the *subtraction* of the neighbor set of the vertex i. Take v_2 in the 3-size pattern 1 as an example, since the incoming edge $(0, 2)$ is black and edge $(1, 2)$ is red, $v_2 \in N(v_0) - N(v_1)$.

2.3 Motivation

Identify suitable PIM workload from GPMI applications. As shown in Figure 2, accessing the neighbor vertices and operating on the neighbor vertex list (e.g., $N(v_0)$, $N(v_1)$) with intersect($\cap$) and subtraction($-$) set operations (I/S operations in short) are critical and memory-intensive. This makes I/S operations include most of the memory accesses of the *nest_for_loop* function. Table 1 summarizes the execution time and memory access ratios of the I/S operations in the *nest_for_loop* function on 5 different graphs (The details of the graphs are shown in Table 2).

TABLE 1
The time and memory access ratios of I/S operations in the *nest_for_loop* function.

Matching Size	Graph	Execution time ratio	Memory access ratio
3-size	CiteSeer	19.12%	38.97%
	MiCo	29.67%	94.16%
	Patents	23.65%	85.46%
	Youtube	16.67%	92.82%
	LiveJournal-1	24.70%	96.04%
4-size	CiteSeer	26.16%	36.29%
	MiCo	32.87%	96.06%
	Patents	25.00%	89.42%

The data is collected by running the applications on a 16-core CPU simulator (details in §4.1). We find that I/S operations account for 20∼33% of the entire *nest_for_loop* execution time. Moreover, the number of memory accesses coming from I/S operations accounts for 85∼96%, except for *CiteSeer*. *CiteSeer* graph is relatively small (84KB) and can fit into the cache (Table 3), so the I/S operations generate a relatively small proportion of memory accesses. In other larger graphs, I/S memory access accounts for more than 85%, which can be considered as a memory-intensive application. Therefore, it is reasonable to offload I/S operations to PIM. Note that the I/S operations do not dominant the execution time, other code blocks which involve complex computation but few memory accesses (e.g., select the execution order of the I/S operations, and remove the duplication of the result in each *for* loop) also need to be optimized with other software or hardware approaches.

Load imbalance of I/S operations. Additionally, from Figure 2, we know that finding n-size patterns requires n layers of *for* loops. When executing the code on multiple cores, the most straightforward way is to assign the I/S operations (the second *for* loop to the last *for* loop) to the same core base on the root vertex (v_0 in the first-level loop). Such a method (root vertex-based assignment) can guarantee the data dependency of following I/S operations.

However, the number of loops at each layer is determined by the *results* of I/S operations (e.g., $N(v_0) - N(v_1)$ in Figure 2), which varies a lot based on patterns or graphs, and cannot be determined by offline profiling. In comparison, for general graph processing applications such as BFS and PR, the workload of each vertex is small and easy to obtain from the vertex degree. Therefore, compared with general graph processing applications, the workloads of GPMI applications on different cores could differ significantly. As the matching size increases, the number of layers of the *for* loop increases, resulting in a more significant load imbalance. Also, PIM usually has much more cores (128 in this paper) than the host, making the load imbalance problem even worse. To fully utilize the parallelism brought by the PIM architecture, we should address the load imbalance issue properly.

3 I/S Operation Scheduling

We now describe the following three root vertex-based scheduling methods. Note that, round-robin and balanced queue are two common scheduling schemes to address load imbalance issue. We also show the results of No Dependency on PIM as a reference, representing the ideal case where we do not consider the execution dependencies between I/S operations. Our system settings can be found in §4.1.

- **I/S on CPU (CPU).** This scheme only uses CPU cores. The *nest_for_loop* function is assigned to different CPU cores according to their root vertices (v_0) IDs.
- **Round-robin on PIM (RR).** I/S operations are all executed on PIM cores. Based on the root vertex ids, the *nest_for_loop*

functions (tasks) on CPU core 0 are assigned from PIM core 0 to PIM core 7 in turn, and then from PIM core 7 to PIM core 0 in reverse.

- **Balanced queue on PIM (BQ).** In the PIM core, for each I/S operation, we sum the lengths of the two arrays ($N(v_i)$ and ($N(v_j)$) to estimate the workload of this I/S operation and store the estimated workload in a queue. Then for an incoming task, the host assigns the task to a corresponding PIM core with the least workload in the queue.
- **No Dependency on PIM (Ideal).** This scheduling method ignores the dependencies among all I/S operations with the same root vertex and treats each single I/S operation as an independent task. I/S operations are assigned to different PIM cores through the round robin strategy.

4 Evaluation

4.1 Experimental Setup

Graph Datasets. Table 2 shows the five real-world graphs used in our experiments. These graphs are used in most graph mining papers, such as Arabesque [19], RStream [20], and AutoMine [13]. Before execution, we sort the vertices based on their degree from largest to smallest (the id of the vertex with the highest degree is 0, the id with the second highest degree is 1, and so on).

Applications. We run 3-size and 4-size motif counting (MC) in the experiments. For 3-size MC, there are 2 different patterns; and for 4-size MC, there are 6 different patterns (Figure 1). We evaluate the performance of these 8 patterns separately.

System Configurations. We use ZSim [17] with Ramulator [10] to simulate the host CPU and PIM system. We modify Zsim to generate traces for the CPU and PIM when executing the *nest_for_loop* function. We also modify Ramulator to support 16 CPU cores with 3-level caches and 128 PIM cores (following Samsung PIM core number [11]) with L1i and L1d caches. All caches use LRU policy. The host CPU frequency is 4 times of the PIM core frequency, which is also adopted from the Function-In-Memory DRAM [11]. We allow one CPU core to assign tasks to the corresponding 8 PIM cores (CPU core 0 offloads tasks to PIM cores 0-7, etc). Table 3 shows the detailed configurations of our simulated system.

4.2 Experimental Results and Analysis

Figure 3 shows the I/S operation execution time with different scheduling schemes. We present two values for each experiment: the *longest time* (in light color) all cores finish the workload; the *average time* (in shaded color) spent on each core to complete the workload. The closer the longest time is to the average time, the more balanced the workload at each core.

Average time. For all graphs and patterns, the average I/S time on PIM is around half the average time on the CPU. This indicates that the selected I/S operations are indeed suitable for PIM execution. While the number of PIM cores is 8 times the CPU cores, the frequency of the PIM is 4 times slower than that of the CPU. Taken together, PIM is twice as fast as CPU. Second, for the three scheduling methods on PIM, although the task scheduling methods are different, the total workload is the same, so the average time of the three methods is very close.

TABLE 2
Graph Datasets [13]
5% Deg. = the top 5% node degrees / all node degrees.

Graphs	Vertices	Edges	Size	Avg.Deg.	Max.Deg.	5% Deg.
CiteSeer	3264	4536	84KB	2.78	99	23.2%
MiCo	100K	1.08M	18MB	21.60	1359	29.9%
cit-Patents	3.77M	16.52M	332MB	8.75	793	22.9%
com-Youtube	1.13M	2.99M	57MB	5.27	28,754	56.7%
soc-LiveJournal1	4.85M	43.11M	1.2G	17.79	20,334	42.4%

TABLE 3
System configurations

Host Processer [1], [6]	
Cores	16 OoO cores, 4GHz, 4-issue
L1I Cache	private, 32KB, 4-way, 4-cycle , 64B, 16 MSHRs
L1D Cache	private, 32KB, 8-way, 4-cycle , 64B, 16 MSHRs
L2 Cache	private, 256KB, 8-way, 12-cycle, 64B, 16 MSHRs
L3 Cache	shared, 16MB, 8 banks, 16-way, 28-cycle, 64B, 16 MSHRs per bank
PIM Cores	
Cores	128 in-order cores, 1GHz, 4-issue [1], [7], [11]
L1I Cache	private, 32KB, 4-way, 4-cycle, 64B, 16 MSHRs [6]
L1D Cache	private, 32KB, 8-way, 4-cycle, 64B, 16 MSHRs [6]
3D Memory Stack	
Organization	4GB, 4 layers × 32 vaults × 1 stack [3]
Timing Parameters	$t_{CK} = 1$ ns, $t_{RAS} = 27$ ns, $t_{RCD} = 14$ ns, $t_{CL} = 14$ ns, $t_{WR} = 15$ ns, $t_{RP} = 14$ ns [14]
Serial links	4 links, 16 bits link width, 30Gb/s lane speed, total 240GB/s bandwidth [3]
Internal links	32 links, 12 Bytes/cycle, 15GB/s per link, total 480GB/s bandwidth [14]

Load imbalance on CPU and PIM. We observe moderate load imbalance on CPU for most patterns and graphs, except for Youtube 3-size pattern 1 and CiteSeer 4-size pattern 1. Meanwhile, since PIM has more cores, we observe a much more severe load imbalance. Compared to the CPU cores, while the average time with PIM cores is reduced, the longest time may not(e.g., with RR scheduling). In CiteSeer, 3-size Youtube, and 4-size Mico pattern 1&3, imbalanced workload makes I/S operations completion time on PIM longer than on CPU.

Effectiveness and limitations of PIM-side scheduling. With BQ scheduling, we use static information (length of two neighboring lists) to estimate each core's workload. BQ can effectively mitigate the load imbalance in various input graphs. However, compared to the Ideal case, we can further reduce a performance gap on 3-size pattern 1, 4-size patterns 1-3. The results show that, while the length of two neighboring lists can estimate the heaviness of workload per core, using the root vertex to partition the I/S operations could lead to a large task on a core, and no matter how the task is scheduled, it will always be the bottleneck.

Load imbalance regarding pattern size. As the pattern size increases, the number of *for* loop layers of the *nest_for_loop* function increases, and the workload of a single task scheduled by root vertex increases, resulting in more obvious workload differences of each task. For CiteSeer, Mico and Patents graphs, all the scheduling methods of 4-size pattern 1 are more unbalanced than the corresponding 3-size pattern 1.

Load imbalance regarding pattern shape. As discussed in §2.3, due to the root vertex-based task assignment, the task per core can vary a lot based on the results of the I/S operations. Additionally, the characteristics of patterns can also determine the load. The patterns in Figure 1 can be divided into three categories: a) 3-size pattern 1, 2; b) 4-size pattern 1, 3, 5, 6; c) 4-size pattern 2, 4. In each category, a following pattern has one more edge than the previous pattern. In the *nest_for_loop*, adding an edge to the pattern means that a subtraction($-$) operation in the *for* loop will be replaced by a intersection($\cap$) operation. Since $A - B = A \cap \overline{B}$, according to the data in Table 2, the edges of all graphs are sparse, which means that $|\overline{N(v)}| >> |N(v)|$. Therefore, $|N(v_i) - N(v_j)| = |N(v_i) \cap \overline{N(v_j)}| >> |N(v_i) \cap N(v_j)|$. As a result, the execution time and load imbalance decreases when we extend the edge in each category: a) 3-size: pattern $1 > 2$; b) 4-size: pattern $1 > 3 > 5 > 6$; c) 4-size pattern $2 > 4$. Our experimental results in Figure 3 also show the same behavior. This is also reason that for the 3-size pattern 2 and 4-size patterns 4, 5, and 6, the root vertex BQ scheduling method can almost mitigate the load imbalance, but the scheduling method needs to be optimized for other patterns with more subtraction operations.

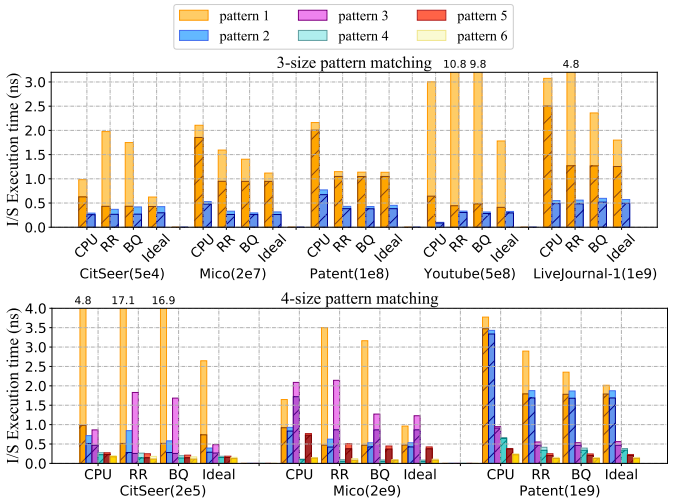

Fig. 3. The I/S operation performance of 3-size and 4-size matching.

Potential solutions to address load imbalance with PIM architecture. As such, to fully utilize the PIM architecture to accelerate the I/S operations, there is a need to address the load imbalance issue, which cannot be mitigated with conventional scheduling schemes (RR or BQ) for graph processing. Two potential solutions are: divide the workload based on non-root vertex id so that the scheduling can be done at a finer grain; implement work stealing mechanisms among PIM cores so that the workload distribution to PIM cores can be determined at runtime.

Impact of parallelism and bandwidth. We have also evaluated the execution time of the BQ scheduling method on 128 CPU cores. Due to the limited space, we discuss the average execution time of running 3-size pattern 1 on the CiteSeer graph as an example. On 16 CPU cores at 4GHz is 31.3 ms, on 128 CPU cores at 4GHz is 11.9 ms. By reducing the CPU core frequency to 1GHz, on 128 cores, the time is 21.3 ms. In comparison, on 128 OoO PIM cores, the time is 15.9 ms, and on 128 in-order PIM cores, the time is 21.6 ms. We observe that both parallelism and bandwidth of PIM architecture could impact the performance. The parallelism gained from 16 CPU cores to 128 CPU cores improves the I/S operation performance by 2.6x. When all are running at 1GHz, 128 OoO PIM cores can be 1.34x faster than 128 OoO CPU cores, which means that the high memory bandwidth can also help with the performance.

5 Related Work

Several recent works proposed specialized hardware for GPMI applications. GRAMER [21] adds a specialized memory hierarchy, where the valuable data permanently resides in the static memory while others are dynamically maintained in a cache-like memory with a lightweight replacement strategy to improve the performance. IntersectX [16] accelerates graph mining with the help of the extension of stream instructions set and the architectural improvement based on conventional processors. SISA [2] uses specialized set-centric ISA and in-memory logic to alleviate the bandwidth requirements of the set operations. Compared with the concurrent work listed above, we focus on leveraging general-purpose PIM cores to accelerate GPMI applications with balanced data and task allocation. Moreover, we identify that, to efficiently exploit the computation power of host and PIM, we have to carefully schedule the operations while maintaining the correct dependencies between loops. The observations and challenges are not discussed in any other work.

6 Conclusions

To conclude, in this work, we identify the PIM-suitable I/S operations in GPMI applications and evaluate them on general purpuse PIM architecture. However, it is challenging to maintain good load balancing when assigning tasks from the host to the PIM. We find that load imbalance significance depends on many factors, including the pattern itself. We evaluate two classical scheduling schemes and find out that static scheduling schemes for improving load balance cannot fully solve the problem. The root cause is from the unique GPMI algorithm: using the root vertex to partition the I/S operations could lead to a large task on a core, and no matter how the task is scheduled, the large task will always be the bottleneck. Based on the observations, we plan to explore fine-grained scheduling schemes with runtime work-stealing to further release the power of PIM architecture for this new class of applications.

References

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *ISCA*, 2015.
[2] M. Besta, R. Kanakagiri, G. Kwasniewski, et al. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. *arXiv preprint arXiv:2104.07582*, 2021.
[3] H. Consortium et al. Hybrid memory cube specification 2.1. *Retrieved from hybridmemorycube. org*, 2013.
[4] P. Das, S. Lakhotia, et al. Towards near data processing of convolutional neural networks. In *VLSID*, 2018.
[5] S. Elbassuoni and R. Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 237–242, 2011.
[6] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *PACT*, 2015.
[7] P. Gu, X. Xie, Y. Ding, et al. ipim: Programmable in-memory image processing accelerator using near-bank architecture. In *ISCA*, 2020.
[8] W. Huangfu, X. Li, S. Li, et al. Medal: Scalable dimm based near data processing accelerator for dna seeding algorithm. In *MICRO*, 2019.
[9] H. Jun, J. Cho, K. Lee, et al. Hbm (high bandwidth memory) dram technology and architecture. In *IMW*, 2017.
[10] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *CAL*, 2015.
[11] Y.-C. Kwon, S. H. Lee, J. Lee, et al. A 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *ISSCC*, 2021.
[12] D. Mawhirter, S. Reinehr, and et.al. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877*, 2019.
[13] D. Mawhirter and B. Wu. Automine: harmonizing high-level abstraction and high performance for graph mining. In *SOSP*, 2019.
[14] L. Nai, R. Hadidi, J. Sim, et al. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *HPCA*, 2017.
[15] S. Parthasarathy, S. Tatikonda, and D. Ucar. A survey of graph mining techniques for biological datasets. In *Managing and mining graph data*, 2010.
[16] G. Rao, J. Chen, and X. Qian. Intersectx: An accelerator for graph mining. *arXiv preprint arXiv:2012.10848*, 2020.
[17] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ISCA*, 2013.
[18] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.
[19] C. H. Teixeira, A. J. Fonseca, et al. Arabesque: a system for distributed graph mining. In *SOSP*, 2015.
[20] K. Wang, Z. Zuo, J. Thorpe, et al. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*, 2018.
[21] P. Yao, L. Zheng, Z. Zeng, et al. A locality-aware energy-efficient accelerator for graph mining applications. In *MICRO*, 2020.