An Order Sampling Processing-in-Memory Architecture for Approximate Graph Pattern Mining

Ziqian Wan[1,2], Guohao Dai[1], Yun Joon Soh[3], Jishen Zhao[3], Yu Wang[1]
[1]Department of Electronic Engineering, BNRist, Tsinghua University, Beijing, China [2]Department of Electrical Engineering and Automation, Tsinghua University, Beijing, China [3]University of California, San Diego, CA, USA
E-mail: wanzq16@tsinghua.org.cn, {daiguohao, yu-wang}@mail.tsinghua.edu.cn, {yjsoh, jzhao}@eng.ucsd.edu

ABSTRACT

There have been increasing interests in graph pattern mining due to the booming of data volume in various domains. Conventional graph mining implementations which calculate the exact count of patterns usually suffer from huge amounts of intermediate data and low performance on large-scale graphs. With the observation that the exact pattern counts are not required in many real-world graph pattern mining problems, previous works (e.g., ASAP [5]) proposed an approximate graph pattern mining algorithm and improved the performance of graph pattern mining by up to two orders of magnitudes. The crucial sampling operation in the ASAP algorithm exposes high parallelism and complex edge searching. Moreover, the performance of ASAP is closely related the sampling order. However, previous works failed to tackle these problems in the design. Thus, we propose a novel Processing-in-Memory (PIM) architecture for parallel approximate graph pattern mining problems. We introduce dictionaries on the logic layer of PIM devices for edge indexing. We also explore the design space of sampling orders and give the optimal sampling strategy. The comprehensive experimental results show that, our design achieves up to 97× performance improvement against ASAP system.

KEYWORDS

large-scale graph mining, approximate computing, processing-in-memory

ACM Reference Format:
Ziqian Wan[1,2], Guohao Dai[1], Yun Joon Soh[3], Jishen Zhao[3], Yu Wang[1]. 2020. An Order Sampling Processing-in-Memory Architecture for Approximate Graph Pattern Mining. In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20), September 7–9, 2020, Virtual Event, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3386263.3406912

1 INTRODUCTION

As we are living in the "big data" era, the data scale has been dramatically increasing. Meanwhile, data analysis has been much more complex in recent years. Graph pattern mining is an important category of graph problems, which finds a certain pattern (e.g., triangles, four cliques, etc.) in a large-scale graph. Such a problem plays a crucial role in many different domains, such as financial risk management [10], drug finding [12], social network similarity analysis [9], accounting fraud detection [2], etc. Many systems and architectures have been proposed to provide solutions to implement graph mining algorithms, such as Graphminer [17], Pegasus [6], Arabesque [13], RStream [14], etc [15][16].

The typical method to implement the graph mining algorithms can be described as joining different candidate sets. The underlying complexity of pattern mining algorithms largely undermines the scalability of these systems. Taking advantage of distributed systems looks a reasonable solution for processing such massive intermediate data, but it arouses the problem of doing expensive joins to create candidates, which severely degrades the performance. Arabesque [13] proposes a new abstraction that optimizes how intermediate candidates are stored, but it still lacks scalability.

In order to improve the performance of graph pattern mining problems, ASAP [5] proposed an approximate method based on edge sampling. In ASAP, edges are gradually sampled and added to an embedding until a pattern is completed or fails to complete it, and the sampling probability is used to calculate the count of the pattern. Millions of estimators are issued and results can be averaged. With such an approximation strategy, ASAP outperforms Arabesque by up to 77× on the LiveJournal [19] graph while incurring less than 5% error on a 16 machine cluster.

Table 1: Utilization of Memory Bandwidth

task	triangle	4-clique(N)	4-clique(D)
utilization(%)	7.41	7.66	7.56
task	5-clique(N)	5-clique(D+N)	5-clique(N+D)
utilization(%)	7.89	13.25	11.16

However, there still exist several problems in the ASAP system. (1) **Expensive edge searching.** ASAP requires frequent searching operations during sampling. However, edge searching on an unordered edge stream based on traversing poses heavy overheads on the memory system. (2) **Inefficient sampling order.** ASAP system adopts only neighborhood sampling. However, in cases when the maximum degree of the graph is too large, the number of estimators needed becomes too large, causing the mining task to become too heavy. (3) **Lack of parallelism and memory bounded.** The approximate graph pattern mining task is composed of massive independent parallel estimators. Moreover, the algorithm involves intensive random memory accesses, causing the actual bandwidth much lower than available memory bandwidth. Table I gives out the memory bandwidth usage of mining different patterns on com-Youtube [19] graph. We observe that the utilization is only around 10%.

We propose our design to tackle the problems existing in ASAP by offering three optimizations to it. (1) **Dictionaries**. We introduce

GLSVLSI '20, September 7–9, 2020, Virtual Event, China

ACM ISBN 978-1-4503-7944-1/20/09…$15.00
https://doi.org/10.1145/3386263.3406912

dictionaries in our design to make edge searching easier and faster. The dictionaries store the neighboring relationships between vertices and edges, as well as the location of edges in the edge stream. The dictionaries relieve the design from traversing the edge stream to search for an edge. (2) **Order-Aware Sampling**. For complex patterns, there are multiple sampling orders of edges in a pattern, and there exists the most efficient sampling order. Our design can evaluate the overhead of every sampling order, and adopt the one with the least overhead as our sampling order. (3) **Processing-in-Memory**. The ASAP algorithm requires massive parallelism as well as fast memory access, thus we implement the algorithm on Processing-in-memory (PIM) architecture to utilize the memory-capacity-proportional bandwidth and high parallelism it provides. To fully exploit the local bandwidth of a vault, we integrate SRAM-based On-chip Dictionaries Buffers in our design. All these three designs, lead to 97× performance improvement compared with the ASAP system. Contributions of this paper are concluded as follows:

- We introduce dictionaries that store neighboring and location information to the approximate pattern mining algorithms, offering easier and faster inquiry that greatly reduces overheads of searching edges.
- We design a method to evaluate overheads of different sampling orders, and enable our design to automatically adopt the fastest sampling order to maximize mining efficiency.
- We implement the algorithm on Processing-in-Memory (PIM) architecture for higher memory bandwidth and computing parallelism.

The rest of this paper is organized as follows. Section 2 introduces the background of approximate graph pattern mining and PIM architecture. Section 3 illustrates algorithm optimization. Section 4 proposes the PIM architecture of our design and introduces the processing flow. We show the results of comprehensive experiments in Section 5. We conclude this paper in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Approximate Graph Pattern Mining

Graph pattern mining is an important branch of graph processing problems. There are several different tasks of mining a graph, such as triangle counting, clique finding, and frequent subgraph mining. The conventional approach of graph pattern mining is inefficient due to complex computation tasks, massive intermediate candidate storing and expensive joins.

ASAP proposes an approximate pattern mining method based on edge sampling. ASAP organizes edges as a stream, and adopts neighborhood sampling to do the mining. An estimator first randomly samples an edge from the stream. Then, a second edge which is the neighbor of the first edge is sampled from the stream after the sampled edge, and so on. The sampling process ends when the pattern is completed, or no edges in the stream after the sampled edge can be found to complete the pattern. ASAP estimates the count of the pattern by calculating the probabilities of sampling edges. Millions of estimators are issued and results can be averaged for these estimators. We illustrate the process of neighborhood sampling with an example of triangle counting in Figure 1.

In this example, we sample triangles from the given graph. Four estimators are executed. The sampled edges are written in the

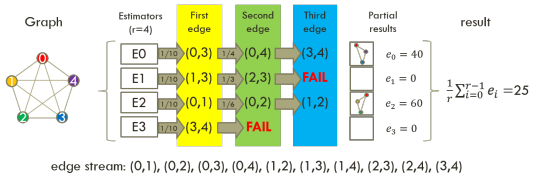

Figure 1: An example of mining triangles with neighborhood sampling. Four estimators are issued and the final answer is calculated with the probabilities of sampling.

columns and the probability of sampling the edge is written in the arrow. Take estimator E1 for example. **First edge** l_0. We first uniformly samples an edge from the graph as the first edge l_0, which is (1,3), and the probability is $Pr(l_0) = 1/m = 1/10$, where m is the number of edges in the graph. **Second edge** l_1. We uniformly sample one of l_0's adjacent edges from the graph as the second edge l_1, which is (2,3), note that l_1 appears after l_0. As there are 3 candidates, the probability is $Pr(l_1|l_0) = 1/c = 1/3$, where c is the number of l_0âĂŹs neighbors appearing after l_0. **Third edge** l_2. We try to find the third edge l_2 to finish the triangle, which should be (1,2). However, (1,2) appears before l_0 and l_1 in the edge stream, so we fail to complete the triangle and the partial result of estimator E1 is 0. If we successfully complete the triangle, like in estimator E0, the partial results is the reciprocal of the multiplication of the probability of sampling each edge, i.e. $e_0 = 1/[Pr(l_0)Pr(l_1|l_0)Pr(l_2|l_0, l_1)] = 1/(1/mc) = 40$. Finally, we calculate the average value of the partial results and get the final result, which is 25. Note that the number of estimators here is not enough, so the result is not accurate.

The ASAP algorithm consists 3 functions, *SampleEdge*, *ConditionalSampleEdge*, and *ConditionalClose*. The *SampleEdge* function uniformly samples an edge from the graph. The *ConditionalSampleEdge* function uniformly samples an edge that is adjacent to the given subgraph and comes after the subgraph in the order. Given a sampled subgraph, the *ConditionalClose* function checks if another subgraph that appears later in the order can be formed.

2.2 Processing-in-Memory

Processing-in-Memory is proposed to provide high bandwidth for large-scale problems. By moving processing units inside memory, PIM achieves memory-capacity-proportional bandwidth.

Hybrid Memory Cube (HMC) is a 3D die-stacking memory device of PIM. An HMC device is consisted of a single package containing multiple memory layers and one logic layer. These layers are stacked using through-silicon via (TSV) technology. Inside an HMC, memory is organized into vaults. Each vault is functionally and operationally independent.

The HMC has been successfully adopted by several data-intensive applications because of its advantages in high memory bandwidth and low data movement cost. For example, Tesseract [1], GraphP [20] and GraphH [3] all take advantage of memory-capacity-proportional bandwidth HMC provides to efficiently do graph analysis, and achieve one to two orders of performance improvement against DDR-based systems. The problem faced with graph analysis such as random access pattern, poor data locality and unbalanced workloads also exist in approximate graph mining. Furthermore, the need for a great number of independent trials provides great chances for

parallelism. HMC with multiple cubes and vaults is perfect candidate for such high parallelism. Thus in our design, we adopt HMC as the hardware foundation.

3 ALGORITHM OPTIMIZATIONS

The original algorithm in ASAP system suffers from expensive search operations, and the processing efficiency is limited by inefficient sampling order. To tackle theses problems, we introduce dictionaries to help searching, and adopt Order-Aware Sampling to select the most efficient sampling order.

3.1 Sorting and Dictionaries

In the ASAP algorithm, the graph is represented as an edge stream. The original ASAP system uses a random ordering edge stream. But we observe that if we sort the edge list in the pre-processing stage, we obtain three dictionaries, **Dict_edge**, **Dict_loc**, and **Dict_node**, which store the neighboring information between vertices and edges as well as the location information of edges in the edge stream, thus can greatly speedup the search operation in the ASAP algorithm.

In *ConditionalSampleEdge*, to sample an edge that is adjacent to and comes after the sampled subgraph, we need to know which edges are eligible. Instead of searching from the whole edge stream, we store the neighboring information in **Dict_edge**, which gives out the edges that is adjacent to a given node.

In *ConditionalClose*, all of the nodes in a pattern are sampled, and we need to check if the not-sampled edges in the pattern can be found after the sampled subgraph in the stream. In conventional approach, we need to search for these edges from the whole remaining stream. With the help of dictionaries, the search range is greatly shortened. **Dict_loc** stores the location information of edges in the edge stream, and **Dict_node** stores the adjacency information between nodes. We first check if the edge exists by searching for it in **Dict_node**. If the edge is not found, the searching operation is precluded. If the edge is found, then we check if it is within the remaining stream. **Dict_loc** tells us the the location range of the edge in the stream, and we search for the edge in the intersection of the location range and the remaining stream. Thus the work is lightened.

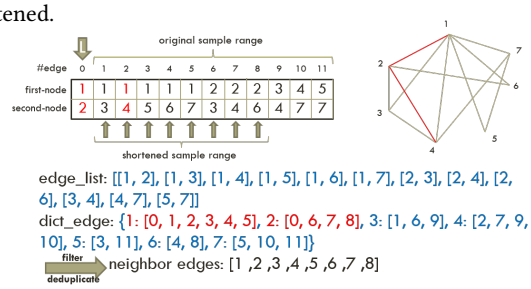

Figure 2: An example of the use of Dict_edge in *ConditionalSampleEdge*. The sampled edges and used dictionary information are highlighted.

We illustrate the use of **Dict_edge** in Figure 2. Let's consider mining triangles on the given graph. Assume that the first edge we sample is (1,2), whose ID is 0. Then we sample the second edge from the neighbors of edge (1,2). Without **Dict_edge**, we need to search from 1 to 11 in the edge stream. But with the **Dict_edge**, we look up the neighbors of node 1 and node 2, filter by the requirement that the ID must be larger than 0, and deduplicate to delete repeated neighbors, finally we get the IDs of neighboring edges. The left work is to sample one edge from the neighbors.

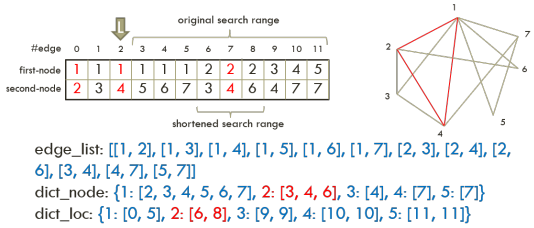

Figure 3: An example of the use of Dict_node in *ConditionalClose*. The sampled edges and used dictionary information are highlighted.

We illustrate the use of **Dict_node** and **Dict_loc** in Figure 3. Assume that the second edge is (1,4), whose ID is 2. To complete this pattern, we need to check if the last edge, (2,4), exist in the remaining stream. Without dictionaries, we need to search it from the whole remaining stream. But with dictionaries, we first look up **Dict_node** to check if the edge exists in the stream, and we find it. Then we look up **Dict_loc** to find out the location range of edges coming out of node 2, which is from 6 to 8. The remaining stream range is from 3 to 11, and the location range is inside the remaining stream range. So we can assert that edge (2,4) is inside the remaining edge stream and this pattern is completed.

3.2 Order-Aware Sampling

In the ASAP system, only the neighborhood sampling strategy is used. However, disjoint sampling has great potential to outperform neighborhood sampling. For example, when the maximum degree of a graph is too large, the number of estimators needed for neighborhood sampling also grows very large, while disjoint sampling is not affected by it. Disjoint sampling strategy uniformly samples disjoint edges from a graph. For example, in four-clique mining, we can simply sample two disjoint edges, and check if the other edges exist to complete the pattern.

With the introduction of disjoint sampling, there are multiple orders of sampling when mining more complex patterns. In this paper, we study strategies of mining k-cliques, where k is an integer larger than 3. Let's denote neighborhood sampling as N, and denote disjoint sampling as D. For three-clique, there's only one strategy, neighborhood sampling (N). For four-clique, there are two strategies, neighborhood sampling (N+N) and disjoint sampling (D). For k-clique, where k is larger than 4, we can extend from (k-2)-clique by sampling a disjoint edge, or extend from (k-1)-clique by sampling a neighboring edge. The different sampling orders of four-clique and five-clique is shown in Figure 4.

The execution time of an order is the multiplication of number of estimators and time cost of an estimator. First, we consider the execution time of each estimator. With the introduction of dictionaries, we greatly reduce the execution time of *ConditionalClose* function. We find that the benefit of the introduction of dictionaries for disjoint sampling is greater than that for neighborhood sampling. For example, mining Four-clique on graph ego-Facebook [8], the improvement for neighborhood sampling is 35.77×, while the improvement for disjoint sampling is 222.7×. In this occasion, the

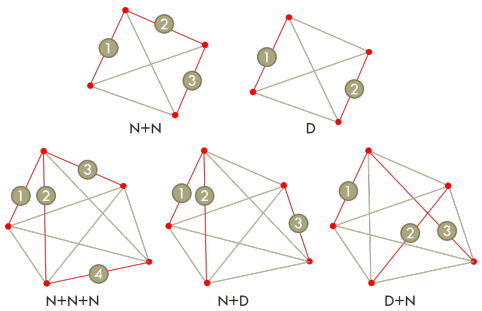

Figure 4: Multiple sampling orders of four-clique and five-clique. The number indicates the sampling order. Sampled edges are highlighted.

disjoint sampling strategy becomes more efficient than neighborhood sampling for one estimator. Similar relationships can be found in other graphs.

In order to estimate the execution time of one estimator, we run a small number of estimators beforehand and calculate the average time for one estimator. In our design, we run 10K estimators to reduce variance.

Then we consider the number of estimator needed for an (ϵ, δ)-approximation, i.e., in $(1-\delta)$ cases the relative error is within ϵ. In our experiments, we specify both δ and ϵ to be 0.05. The number of estimators needed for an (ϵ, δ)-approximation could be bounded using Chernoff bound. For triangle counting, the number of estimators needed is $r \geq \frac{6m\Delta}{\epsilon^2 p} \ln(\frac{2}{\delta})$, where m is the number of edges in the graph, Δ is the maximum degree, and p is the true triangle count. For four-clique mining, the number of estimators needed is $r_N \geq \frac{18m\Delta^2}{\epsilon^2 p} \ln(\frac{2}{\delta})$ for neighborhood sampling, and $r_D \geq \frac{3m^2}{\epsilon^2 p} \ln(\frac{2}{\delta})$ for disjoint sampling. For the three sampling orders of five-clique, the number of estimators required is $r_N \geq \frac{72m\Delta^3}{\epsilon^2 p} \ln(\frac{2}{\delta})$ for N+N+N sampling, $r_{ND} \geq \frac{6m^2\Delta}{\epsilon^2 p} \ln(\frac{2}{\delta})$ for N+D sampling, and $r_{DN} \geq \frac{12m^2\Delta}{\epsilon^2 p} \ln(\frac{2}{\delta})$ for D+N sampling. Although we don't know the true number of pattern count, we can estimate it by running a small number of estimators. With the ground truth number, we calculate the number of estimator needed for every sampling order.

The flow of selecting sampling order can be summarized as:

(1) Give out all possible sampling orders according to the pattern.
(2) Estimate the execution time of an estimator for each sampling order by running 10K estimators.
(3) Calculate the number of estimators needed for an (ϵ, δ)-approximation for each sampling order.
(4) For each sampling order, calculate the execution time by multiplying the time of each estimator with the number of estimators.
(5) Choose the sampling order that costs least time as our sampling order.

4 ARCHITECTURE DESIGNS

We adopt the Processing in memory (PIM) architecture as the hardware foundation in our design. PIM provides large amount of independent processors, thus provides enough parallelism to the the ASAP algorithms that is composed of massive independent estimators. We integrate On-chip Dictionary Buffer and optimize memory access pattern for faster memory access.

4.1 Overall architecture

We implement the algorithm on Hybrid Memory Cube (HMC) device. The architecture design is shown in Figure 5. The architecture includes a main processor and a set of HMC device. The main processor partitions the original graph into several subgraphs, issues estimators to HMCs, and gathers the partial output results of HMCs to calculate the final result. We implement micro-processors on logic layers in HMCs so that each vault is able to execute all operations needed in the approximate pattern mining algorithm.

The parallelism of our design comes from partitions and independent estimators. In our architecture design, there are 16 HMCs in the PIM, and we partition the input graph into 16 subgraphs and store each of them in an HMC. The computation on subgraphs are independent. Inside each HMC, there are 32 vaults, each with several memory layers and one logic layer. The memory storage of each vault is 256 MB, which is enough for subgraphs of even the largest graph, thus we are able to store a copy of the subgraph on the memory layers of each vault. Then we assign each vault with an independent computation work. With all the vaults working in parallel, the total execution time will become 1/32 of that without the parallelism.

As the dictionaries are frequently referred to when executing the algorithm, we integrate an SRAM on logic layer to store the dictionaries. This will further reduce the data movement. We assume that the size of logic layer is the same as memory layers, and use CACTI 6.5 to evaluate the size of DRAM. The area of an 8 GB DRAM with 64-bit input/output bus under 32 nm process is 257.02 mm^2, and the area of a 1MB SRAM is 1.65 mm^2 under 32 nm process [3].Typically the size of the dictionaries are more than 1 GB, making it impossible to store the entire dictionaries on logic layer. Fortunately, most natural graphs follow the Power-Law [4], which means a little proportion of vertices takes up most of the degree in a graph. As the edges with large degree are more often sampled and searched in our algorithm, we store only the neighboring and location information of these high-degree nodes in the dictionaries, thus greatly reduce needed storage room. It proves that about 90% of the search operations are related to these high-degree vertices.

4.2 Processing Flow

In this section, we briefly introduce the processing flow of our architecture. **Inputting**. The graph to be mined is input into the main memory. **Partitioning**. The graph is partitioned into 16 subgraphs using random vertex partitioning [4] by the main processor. Each subgraph is stored in an HMC. Note that the copies of subgraphs are stored on each of the 32 vaults in an HMC. **Preprocessing**. In each HMC, the vault processor sorts the edge stream, generate dictionaries and store dictionaries onto the On-chip Dictionary Buffer. **Selecting**. In each vault, different sampling orders are compared and the most efficient one is chosen. The number of estimators needed for this sampling order is calculated. **Sampling**. In each vault, the logic layer execute estimators in parallel, and output the partial answer. **Gathering**. When all vaults finish the sampling work, the vault processor gathers the partial answers and compute

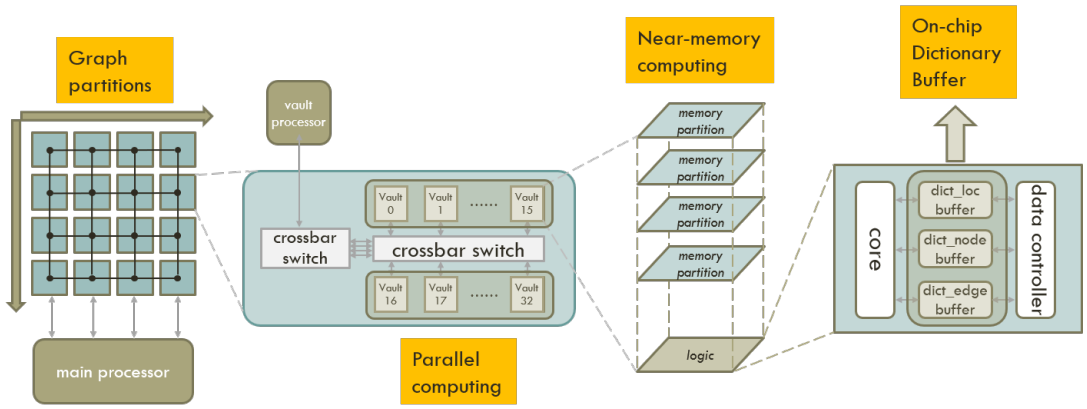

Figure 5: Architecture Design. The PIM architecture provides massive parallelism for independent estimators. The On-chip Dictionary Buffer lighten search operations and further utilize the local bandwidth.

the answer for each subgraph and output it to the main processor. The main processor gathers the result of each subgraph and get the final answer. Outputting. The main processor output the final answer as the estimated number of the mined pattern.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We simulate the performance of our architecture with cycle accurate in-house simulator. We count the number of each instructions with the help of dis dissemble in Python. We refer to the ARM manual to get the cycle counts for each instruction. It is important to note that in our design, the dictionaries that are frequently referred to are stored on SRAM on logic layer, thus the data access efficiency is larger than DRAM memory layers. We analyze the benefit of algorithm optimizations (including dictionaries and order-aware sampling) and the PIM architecture respectively. With the optimizations both on algorithms and architecture, we prove our architecture to be up to 97× faster than the ASAP system.

5.2 Workloads

We adopt 5 nature graphs from the Stanford Large Network Dataset Collection [7] to evaluate the performance of our architecture in Table 2. The mining tasks we assign to our architecture are triangle, four-clique and five-clique finding.

Table 2: Properties of Graphs

Graph	# Edges	# Vertices	Max Degree
ego-Facebook (e-F) [8]	88,234	4,039	1,045
gemsec-Deezer-HR (g-D) [11]	498,202	54,573	420
gemsec-Facebook-Artist (g-F) [11]	819,306	50,515	1,469
com-Youtube (c-Y) [18]	2,987,624	1,134,890	28,754
com-LiveJournal (c-L) [18]	34,681,189	3,997,962	14,815

5.3 Benefits of Algorithm Optimizations

To testify the effectiveness of the algorithm optimizations, we implement both the original and the optimized algorithm on CPU+DRAM architecture to compare the performance difference. We run the experiments on a personal computer equipped with a quad-core Intel i7 CPU running at 2.6GHz.

5.3.1 Benefits of Sorting and Dictionaries. We first test the improvement gained from sorting and introducing dictionaries. The result is shown in Figure 6. We observe that the speedup is up to 2010×.

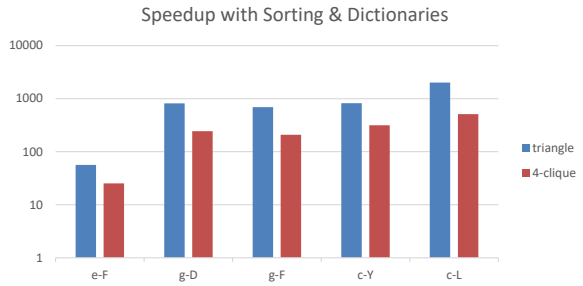

Figure 6: Speedup with Sorting and Dictionaries

5.3.2 Benefits of Order-Aware Sampling. Then we test the improvement gained from order-aware sampling. We take both the number of estimators and time cost of each estimator into consideration. The results are shown in Figure 7. We discover that the speed up to 14,649×. The great improvement comes from the drastic drop of number of estimators better sampling order needs to get the same accurate result. For example, mining 5-clique on c-Y with N+N+N sampling requires 3300× more estimators than with N+D sampling. The improvements of N+D sampling is even larger due to the smaller time overhead for each estimator. We are confident to predict that the speedup of mining more complex patterns will be even larger due to more available sampling orders.

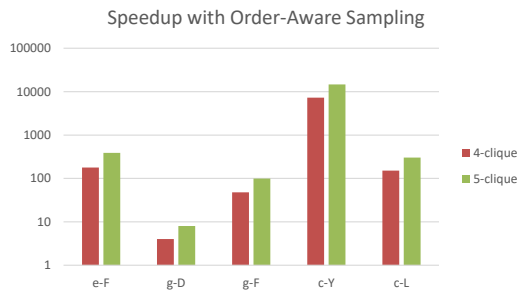

Figure 7: Speedup with Order-Aware Sampling

5.4 Benefits of PIM Architecture

Based on the algorithm optimizations, we execute the improved algorithm on both CPU+DRAM and PIM architecture to compare

their performance. The overhead of CPU+DRAM architecture is measured by running the algorithm on a personal computer equipped with a quad-core Intel i7 CPU running at 2.6GHz, while the overhead of PIM architecture is estimated with the setup in section 5.1. The result is shown in Figure 8. We discover that the speedup is greater when mining more complex patterns, we assume the reason is that mining more complex pattern requires more frequent edge sampling and lookup tasks, which are greatly accelerated with the PIM architecture.

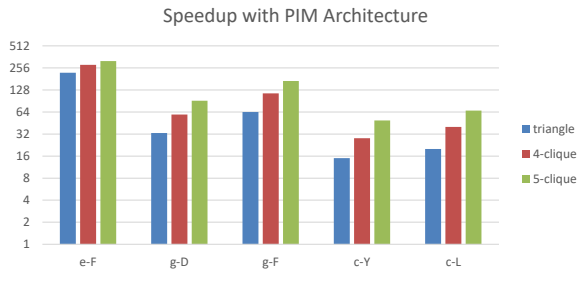

Figure 8: Speedup with PIM Architecture

5.5 Overall Performance

To test the overall performance improvement gained from our design, we assign different mining tasks on Youtube graph and LiveJournal graph to our architecture and compare the execution time with the ASAP system. The result is shown in Figure 9. Note that the execution time of mining 5-clique with ASAP system is not given. We can see that the performance improvement is up to 97× when mining four-clique, and the execution time for mining five-clique is acceptable, while ASAP system fails to mine patterns more complex than four-clique.

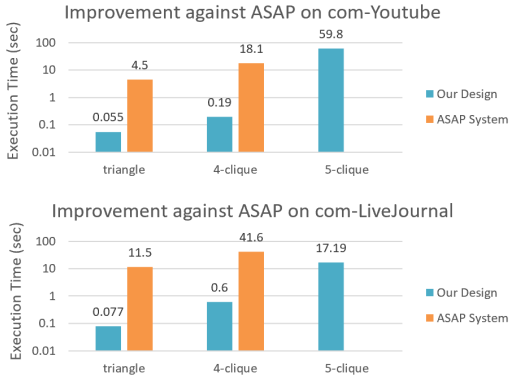

Figure 9: Overall Performance. Our design outperforms ASAP system up to 97× and is capable of mining more complex patterns.

6 CONCLUSION

In this paper, we propose an order-aware Processing-in-Memory architecture design for graph pattern mining. To tackle the problem such as expensive search operation, inefficient sampling order, limited memory bandwidth and parallelism faced by the previous systems, we propose three optimizations to it. First, we introduce dictionaries to make search operation easier and faster. Second, we explore the possibility of adopting different sampling order and design a method to select the most efficient order. Third, we implement the algorithm on PIM architecture for sufficient memory bandwidth and higher parallelism. We also integrate On-chip Dictionary Buffer to further utilize the local bandwidth. With all these designs, our architecture proves to be up to 97× faster than ASAP system, and is capable of mining more complex patterns.

ACKNOWLEDGEMENT

This work was supported by National Key Research and Development Program of China (No. 2017YFA0207600), National Natural Science Foundation of China (No. 61832007, 61622403, 61621091, U19B2019), China Postdoctoral Science Foundation (No. 2019M660641). Beijing National Research Center for Information Science and Technology (BNRist), and Beijing Innovation Center for Future Chips.

REFERENCES

[1] Junwhan Ahn et al. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 105–117.
[2] Leman Akoglu and Christos Faloutsos. 2013. Anomaly, event, and fraud detection in large network datasets. In *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 773–774.
[3] Guohao Dai et al. 2018. Graphh: A processing-in-memory architecture for large-scale graph processing. *IEEE TCAD* 38, 4 (2018), 640–653.
[4] Joseph E Gonzalez et al. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*. 17–30.
[5] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *OSDI*. 745–761.
[6] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. 2009. Pegasus: A peta-scale graph mining system implementation and observations. In *SIGMOD*. Washington, DC, USA, 229–238.
[7] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[8] Jure Leskovec and Julian J Mcauley. 2012. Learning to discover social circles in ego networks. In *NIPS*. 539–547.
[9] Natasa Pržulj et al. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.
[10] Bernardete Ribeiro, Ning Chen, and Alexander Kovacec. 2017. Shaping graph pattern mining for financial risk. *Neurocomputingshapng* (2017).
[11] Benedek Rozemberczki, Ryan Davies, Rik Sarkar, and Charles Sutton. 2019. GEMSEC: Graph Embedding with Self Clustering. In *ASONAM*. ACM, 65–72.
[12] Ichigaku Takigawa and Hiroshi Mamitsuka. 2013. Graph mining: procedure, application to drug discovery and recent advances. *Drug discovery today* 18, 1-2 (2013), 50–57.
[13] Carlos HC Teixeira and other. 2015. Arabesque: a system for distributed graph mining. In *SOSP*. ACM, 425–440.
[14] Kai Wang et al. 2018. RStream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *OSDI*. 763–782.
[15] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction RefinementâĂŤScalable and Programmable Analytics over Very Large Graphs on a Single PC. In *ATC*. 387–401.
[16] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *ATC*. 651–664.
[17] Wei Wang, Chen Wang, Yongtai Zhu, Baile Shi, Jian Pei, Xifeng Yan, and Jiawei Han. 2005. Graphminer: a structural pattern-mining system for large disk-based graph databases and its applications. In *SIGMOD*. ACM, 879–881.
[18] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012). arXiv:1205.6233 http://arxiv.org/abs/1205.6233
[19] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.
[20] Mingxing Zhang et al. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *HPCA*. IEEE, 544–557.