FastRW: A Dataflow-Efficient and Memory-Aware Accelerator for Graph Random Walk on FPGAs

Yingxue Gao, Teng Wang, Lei Gong*, Chao Wang*, Xi Li, Xuehai Zhou
University Of Science and Technology Of China
{gyingxue, wangt635}@mail.ustc.edu.cn, {leigong0203, cswang, llxx, xhzhou}@ustc.edu.cn

***Abstract*—Graph random walk (GRW) sampling is becoming increasingly important with the widespread popularity of graph applications. It involves some walkers that wander through the graph to capture the desirable properties and reduce the size of the original graph. However, previous research suffers long sampling latency and severe memory access bottlenecks due to intrinsic data dependency and irregular vertex distribution.**

This paper proposes FastRW, a dedicated accelerator to release GRW acceleration on FPGAs. FastRW first schedules walkers' execution to address data dependency and mask long sampling latency. Then, FastRW leverages pipeline specialization and bit-level optimization to customize a processing engine with five modules and achieve a pipelining dataflow. Finally, to alleviate the differential accesses caused by irregular vertex distribution, FastRW implements a hybrid memory architecture to provide parallel access ports according to the vertex's degree. We evaluate FastRW with two classic GRW algorithms on a wide range of real-world graph datasets. The experimental results show that FastRW achieves a speedup of 14.13× on average over the system running on two 8-core Intel CPUs. FastRW also achieves 3.28×∼198.24× energy efficiency over the architecture implemented on V100 GPU.

***Index Terms*—graph random walk, accelerator, dataflow, memory architecture.**

I. Introduction

Graph random walk (GRW) sampling aims to capture the desirable graph properties that are used to estimate the original graph. It involves some sampling walkers that wander through the whole graph to collect paths, which has been applied to many application fields, such as graph learning [1], embedding representation [2], [3], and graph ranking.

Unlike traditional graph algorithms, which treat all neighbor vertices similarly, GRW sampling performs a more sparse operation. It only selects one neighbor of interest among many potential candidates to follow. Intuitively, the bulk of computation resides in the neighbor vertex selection. This process is expensive and has become a performance bottleneck.

Some existing general graph frameworks are configured to accelerate GRW sampling, such as DrunkardMob [4] and Deep Graph Library (DGL) [5]. However, they lack customized optimizations oriented to the sampling process. Then, many specialized software systems have been developed to boost the sampling operation, such as KnightKing [6], NextDoor [7], GraphWalker [8], and C-SAW [9]. The above CPU or GPU-based systems fail to provide appreciable performance due to random memory accesses and intrinsic data dependency during the sampling process. As a promising platform, FPGAs can offer ideal parallelism in many fields, such as [10]–[12]. In addition, [13] implements an FPGA-based architecture to accelerate GRW sampling. However, it still faces two problems, limiting its efficiency and versatility. First, it suffers from inefficient dataflow due to a lack of efficient pipeline parallelism. Second, it lacks support for more challenging dynamic GRW sampling.

This paper concludes that GRW sampling has the following characteristics that make it difficult to accelerate efficiently. 1) **Intrinsic data dependency**: The sampling operation leverages a walker-centric computation model to perform. However, it involves strict data dependency, which invalids pipeline parallelism and leads to long sampling latency. 2) **Frequent random accesses**: The vertex selection process involves sparse and random accesses. Predicting which vertices will be visited is difficult. So it fails to preload data on-chip and introduces a large number of off-chip accesses. 3) **Unbalanced vertex distribution**: Dynamic GRW sampling needs to check connectivity between vertices, which involves the loading of a large number of neighbor vertices. Even worse, the vertex distribution is unbalanced. The vertices with fewer neighbors need to wait for vertices with more neighbors to transfer.

To address the above problems, this paper proposes FastRW, a dedicated accelerator to boost GRW sampling on FPGAs. FastRW first schedules walkers' execution to address data dependency and mask long sampling latency. Then, FastRW leverages pipeline specialization and bit-level optimization to customize a processing engine with five modules and achieve a pipelining dataflow. Finally, to alleviate the differential accesses caused by unbalanced vertex distribution, FastRW implements a hybrid memory architecture to provide the parallel access ports according to the vertex's degree. FastRW also designs a bit-level structure unit with low hardware overhead for fast connectivity checking. We evaluate FastRW on the Xilinx Alveo U50 card. The results show that FastRW achieves a speedup of 14.13× on average over the CPU system and 3.28×∼198.24× energy efficiency over the GPU implementation.

To summarize, the main contributions are as follows:

- We identify the characteristics of the GRW sampling and propose FastRW, an accelerator to boost both static and dynamic GRW samplings on FPGAs.
- FastRW first schedules walkers' execution to address data dependency. Then, FastRW leverages pipeline specialization and bit-level optimization to customize a processing

*Corresponding authors: Lei Gong and Chao Wang.

978-3-9819263-7-8/DATE23/© 2023 EDAA

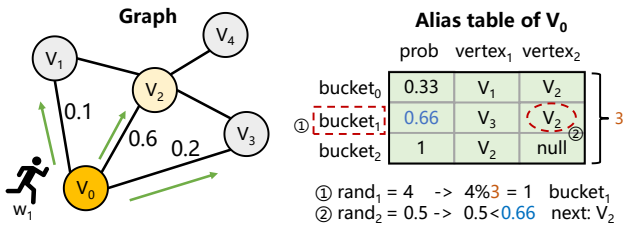

Fig. 1. The left is a GRW sampling schematic, including a graph entry and a walker w_1 starting from vertex V_0. The right side is the alias-based sampling process.

engine.

- FastRW also presents a hybrid memory architecture design adapted to differential accesses caused by unbalanced vertex distribution.
- FastRW achieves a speedup of 14.13× on average over the system running on two 8-core Intel processors. FastRW also achieves 3.28×∼198.24× energy efficiency over the architecture implemented on V100 GPU.

II. Background

In this section, we first describe the preliminaries of GRW sampling and its algorithm variants. Then, we introduce a walker-centric computation model.

A. Graph Random Walk

GRW sampling takes a graph as input, along with some walkers. Fig. 1 shows an example of the sampling operation. The walker w_1 starts from vertex V_0. Then, it needs to select one neighbor vertex (V_1, V_2, or V_3) to follow. To boost sampling efficiency, we adopt the alias-based sampling method [6]. It needs to construct the alias table in advance, which records the vertex connections and edge properties information. The right side of Fig. 1 shows an alias table built for V_0. The primary cell is the bucket. Each bucket contains three elements, $prob$, $vertex_1$, $vertex_2$. The sampling operation is based on the alias table to perform and contains two steps. First, the random number $rand_1$ (4) performs the remainder operation on the out-degree of V_0 (3), and the result is used as an index to access the $bucket_1$. Second, the random number $rand_2$ (0.5) is less than probability $prob$ (0.66), so the second vertex $vertex_2$ in $bucket_1$ is selected as a candidate.

This work focuses on two key GRW sampling algorithms: static sampling [2] and dynamic sampling [3]. For static sampling, the vertex selection is only related to the walker's current residing vertex. The dynamic algorithm needs to consider the vertices it has previously visited. It is more potent in reality and has more deployment challenges.

B. Walker-centric Computation Model

The sampling operation can be further abstracted as a walker-centric computation model, as shown in Algorithm 1. First, the walker's state parameters are initialized (line 1). Then, with the pre-built alias table $aliasTable$ and the random number $rand_1$ as inputs, one $bucket$ in the alias table is selected (line 4). Next, with the random number $rand_2$

Algorithm 1: Walker-centric computation model

```
Input: alias table, random number
Output: sampled path
1  init: walker's state parameters;
2  while checkLength(walker) do
3      rand1, rand2 =getRands();
4      bucket=Sample(walker, aliasTable, rand1);
5      candidate=Select(bucket, rand2);
6      if isAccepted(candidate) then
7          walker.Update(candidate);
8          path.Add(candidate);
9      end
10 end
```

as input, one vertex in $bucket$ is selected as a $candidate$ vertex (line 5). If the $candidate$ is accepted, the walker's state parameters are updated with $candidate$ and add this $candidate$ to the sampled path (lines 6-9). There is a strict data dependency during the sampling process. Specifically, the output of the previous round (line 7) is the input of the next round (line 4). The sampling operation will finish until the walker's length reaches a threshold.

III. FastRW Design Overview

In this section, we first introduce the architecture design of FastRW and its execution flow. Then, we present the pipeline specialization and bit-level optimization.

A. FastRW Architecture

Fig. 2(a) illustrates the overall architecture of FastRW, which contains multiple parallel processing engines (PEs) specialized for GRW sampling. Each PE is responsible for processing a group of sampling walkers. In addition, each PE occupies some private storage resources, including on-chip buffers and HBM channels (PC). Multiple PEs leverage the data-level parallelism of the overall architecture. The path buffers are responsible for storing the sampled paths.

The block diagram of the processing engine (PE) is shown in Fig. 2(b), which consists of five customized modules and some local buffers. Next, we present the function of each module.

- *Scheduler*: A lightweight control core that coordinates the execution of sampling walkers inside a PE.
- *Sample module*: A memory-efficient module is responsible for loading the alias table from off-chip memory, which involves a multi-issue address generation unit, a data reception unit, and a comparator.
- *Degree module*: A module that is used to interface with customized memory architecture for neighbor vertices' loading, which contains an address analysis unit and a local buffer.
- *Connector module*: A customized module comprises a bit-level structure unit with low hardware complexity and a comparator. It is responsible for connectivity checking between vertices.

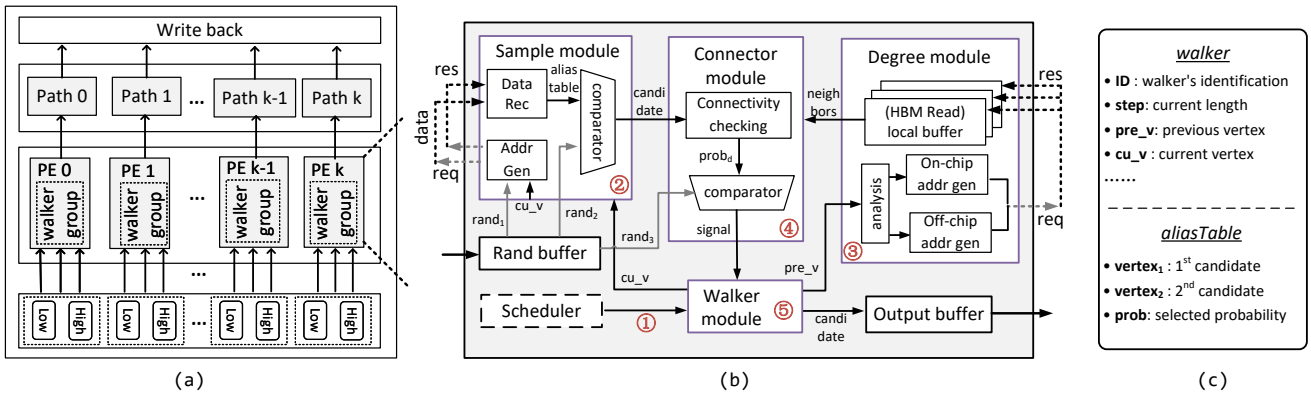

Fig. 2. (a) Overall architecture of FastRW (b) Single processing engine (c) The structure variables contained in the walker and alias table

- *Walker module*: A module that is used to buffer and update walkers' state parameters. Fig. 2(c) shows the structure variables contained in the walker and alias table.
- *Output buffer* and *Rand buffer* are used to store sampled paths and random numbers generated by the host side.

B. FastRW Execution Flow

We take dynamic sampling as an example to introduce the execution flow in FastRW, as shown in Fig. 2(b). ①At the beginning, the **scheduler** selects a walker to launch the sampling operation. ②The walker's current residing vertex cu_v and the random number $rand_1$ are sent to the **sample module** for accumulation. The first operand is the remainder of $rand_1$ and the cu_v's out-degree, and the second operand is the offset of cu_v. The sum serves as an address to load the alias table stored off-chip. Then, according to the comparison result between random number $rand_2$ and $prob$ in the alias table, one $vertex$ in the alias table is selected as a candidate. ③At the same time, the walker's previous visited vertex (pre_v) is sent to the **degree module** for neighbor vertices' loading. The hybrid memory architecture is designed for differentiated vertex accesses. Section IV will present more design details. The loaded neighbors will be buffered on-chip. ④The **connector module** receives the candidate and neighbors for connectivity checking. A bit-level structure unit is designed to check connectivity and output probability $prob_d$. Then, the comparison result of random number $rand_3$ and probability $prob_d$ will be sent to the walker module. ⑤It is decided whether to accept the candidate according to the received signal. If accepted, the **walker module** will update the state parameters and send the candidate to the output buffer. For the static sampling, steps 3-4 are omitted.

C. Pipeline Parallelization and Bit-level Optimization

This subsection presents the details of pipeline specialization and bit-level optimization to enable a high-performance architecture.

*1) **Dataflow and Pipeline Parallelization**:* For walker sampling, the output of the previous round is the input of the next round. Fig. 3(a) shows its execution schematic, where the intrinsic data dependency severely limits sampling efficiency.

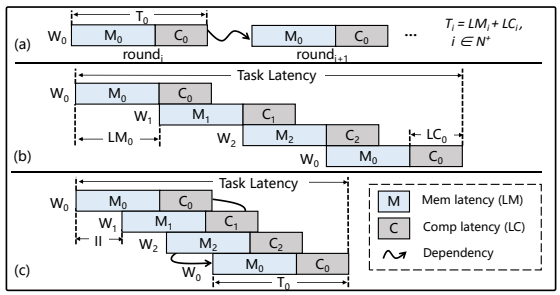

Fig. 3. (a) The original execution diagram with data dependency. (b) The execution diagram that computation latency is masked. (c) The execution diagram that both computation and memory access latency are masked.

Even worse, the sampling operation involves long computation and memory access latency. The computation latency comes from the address generation process. The remainder calculation of the random number and vertex's out-degree consumes about 35 cycles. One off-chip access takes about 81 cycles. So a total of 116 cycles is required in one round of sampling.

Fortunately, the sampling operation between multiple walkers is independent, and the sampling execution is lightweight. So in our architecture, each PE is assigned a group of walkers. The scheduler is responsible for alternating walkers' execution to achieve a pipelining dataflow. As shown in Fig. 3(b), between the two rounds of sampling of walker$_0$, unrelated walker$_1$ and walker$_2$ can be inserted to address data dependency and mask long computation latency. Then, each walker with a separate ID is treated as a private execution, and multiple walkers work together to achieve pipeline parallelism.

After data dependency is addressed, sampling efficiency is still limited by random memory access. Specifically, the access of the alias table involves small individual transmission, which disables burst transmission and requires a more flexible design. So the sample module in FastRW is implemented to allow multiple outstanding memory requests, which can keep the pipelining of data transmission by consuming more BRAM for the interface buffer. Fig. 3(c) shows the final execution schematic. Both computation and memory access latency can be masked. Note that II represents the iteration interval, and the latency of one sampling consumes T cycles. The number

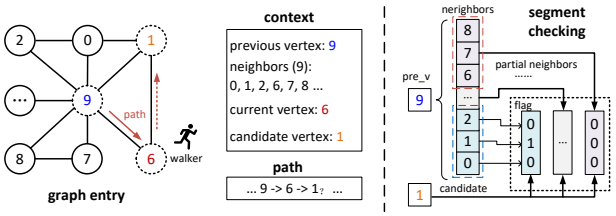

Fig. 4. An example for illustrating connectivity checking. The sampling walker currently resides at vertex 6. It needs to check the connectivity between candidate vertex 1 and previous vertex 9. On the right is a bit-level structure unit for checking connectivity.

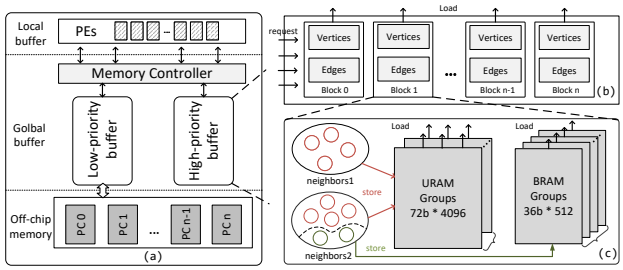

Fig. 5. Hybrid memory architecture design

of walkers num needs to be greater than T/II to enable the correctness of pipelining execution.

*2) **Bit-level connectivity checking**:* The bit-level structure unit is designed to check connectivity. It converts element-level checking to bit-level checking. Fig. 4 shows how it works. The sampling walker currently resides on vertex 6, and vertex 1 is selected as a candidate. Then, it needs to check the connectivity between vertex 1 and previously visited vertex 9 and output a probability. We leverage the $flag$ to keep the connectivity information between vertices. By querying vertex 9's neighbors with the key '1', we get a bitset '010 ... 000'. The non-zero $flag$ indicates vertex 1 is connected with vertex 9. Otherwise, there is no connection. When the number of neighbors is too large, this method will consume lots of LUT resources. So we divide neighbors into some segments and use the n-th bit of $flag$ to check the n-th, 2n-th, ... neighbors simultaneously, called segment checking.

*3) **Bit-width customization**:* The walker-centric computation involves the structure variables to record running contexts, as shown in Fig. 2(c). The alias table with larger bit width and more quantity becomes a storage bottleneck. When the size of the alias table is out of range, it requires the allocation of more channels for access. For the previous CPU/GPU-based design, the parameter type is rough and limited, such as 16-bit or 32-bit. In contrast, FPGA-based design can customize bit width according to actual requirements and reduce storage. For example, the index of the vertex can be represented by 24-bit, and the offset can be represented by 27-bit. Bit width customization is crucial for memory-intensive sampling operations. We present more analysis in the experimental section.

IV. Memory Design

In this section, we first introduce the memory hierarchy in FastRW. Then, we present our hybrid memory design adapted to differential vertex accesses.

A. Memory Hierarchy

Fig. 5(a) shows the memory hierarchy of our architecture, which contains three levels. The **local buffer** in each PE is responsible for buffering intermediate and output data. When a PE generates a memory request, it will send a request through the memory controller. Then, the corresponding response is returned to PE. The **global buffer** is divided into two parts: low-priority and high-priority buffers. Low-priority buffer is used as an intermediate between the local buffer and off-chip memory. High-priority buffer is used to buffer the frequently accessed vertices. Implementing a shared buffer to support concurrent accesses from multiple PEs is inefficient, leading to unexpected bank conflicts. So we partition the high-priority buffer into several isolated blocks, as shown in Fig. 5(b). The number of separate blocks is equal to the number of PEs, which can ensure the accesses from multiple PEs do not affect each other. This design can facilitate the scalability of the architecture while ensuring high performance. The **off-chip memory** serves as the last level of memory architecture. It is responsible for storing the large-scale alias table and most vertices, which interact with the on-chip buffer via the HBM channels. Each PE occupies several private channels to avoid off-chip access conflicts.

B. Hybrid Memory Design

The real-world graph usually follows a power-law distribution [14], [15]. The vertices with more neighbors occupy a small proportion. Based on the above observation, we implement a hybrid memory design, which can provide parallel access ports according to the degree of the vertex. We first divide vertices into two subsets: low-degree and high-degree vertices, and design them separately.

High-degree vertices: The off-chip HBM can provide high external bandwidth (U50: 316GB/s), but leveraging higher internal bandwidth (24TB/s) to provide more parallel ports is attractive. Considering that the vertices with more neighbors have a higher probability of being visited, and the capacity of the on-chip buffer is limited in size. So we only buffer a small proportion of high-degree vertices on-chip to improve data locality. Fig 5(c) shows the on-chip hybrid design. The targeted U50 card involves two key buffer resources: Ultra RAMs (URAM) and Block RAMs (BRAM). URAM is characterized by a large capacity, which can be configured as 72-bit*4096 depth and achieve on-chip data packing. Suppose the vertex is represented by 24-bit, and each cell in URAM can pack 3 vertices. BRAM is characterized by more quantities, which can be configured as 36-bit*512 depth. For each vertex, we allocate appropriate buffer space based on the number of its neighbors. Then, we leverage URAM with large depth and bit width as a primary buffer (buffer all high-degree vertices) and leverage BRAM for the secondary buffer (only buffer some higher-degree vertices).

TABLE I
GRAPH BENCHMARK DATASETS

Datasets	#Vertex	#Edge	Field	Size
amazon0601 (AM)	0.40M	3.39M	Product	75MB
RoadNet-CA (RC)	1.98M	5.53M	Road	131MB
web-Google (WG)	0.92M	5.11M	Web	115MB
cit-Patents (CP)	3.77M	16.52M	Citation	361MB
Reddit (RE)	0.23M	23.21M	Community	509MB
As-skitter (AS)	1.70M	22.19M	Internet	496MB
wiki-topcats (WT)	1.79M	28.51M	Community	674MB
LiveJournal (LJ)	4.85M	68.99M	Social	1.58GB

Low-degree vertices: It occupies the majority of all vertices. We leverage large-capacity HBM (8GB) to buffer. To improve off-chip access efficiency, we also pack these vertices for merge transmission. The available off-chip bus width is N-bit, each vertex is represented by p-bit, and each pack can contain $n = N/p$ vertices. For Alveo U50, when the clock frequency is set to 200Mhz, the bus width can be set to 512-bit. Suppose the vertex is represented by 24-bit. We can put 21 vertices in each pack, which can improve bandwidth utilization effectively.

V. EXPERIMENTAL EVALUATION

In this section, we first describe the experimental setup. Then, we compare FastRW against previous implementations to demonstrate its effectiveness.

A. Experimental Setup

Platform. We deploy FastRW on the Xilinx Alveo U50 card, which contains 8GB HBM memory with 316GB/s off-chip bandwidth. FastRW is implemented with the Xilinx Vitis tool chain (v2020.2). The frequencies of FastRW architecture for static and dynamic samplings are 300MHz and 200Mhz, respectively. We report the running time includes initializing walkers and the sampling process but excludes the path collection. The runtime is collected from on-board execution. Power consumption is obtained from the place-and-route report in Vivado. The average power consumptions of static and dynamic samplings are 32.15W and 32.68W, respectively.

Models and Graph datasets. We evaluate two representative GRW sampling, static and dynamic sampling algorithms described in Section II. Table I lists the graph datasets used for evaluation. They come from different fields, such as product purchase, community, citation networks, etc. All datasets are stored in CSR format. For unweighted graphs, we randomly generate edge weights from the range [0.1-1.1). Size indicates the graph size after generating weights for them. For the undirected graph, the number of edges is doubled to report.

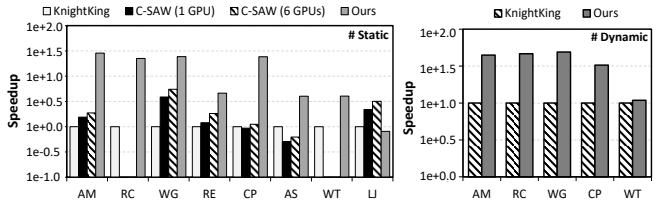

Fig. 6. Performance comparison of FastRW with KnightKing and C-SAW with 1 GPU and 6 GPUs. (Higher is better.)

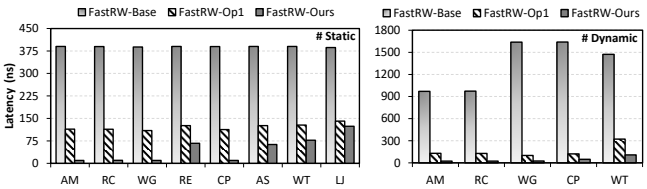

Fig. 7. Latency comparison. (Lower is better.)

B. Overall Comparison

*1) **FastRW vs KnightKing**:* KnightKing [6] is a software system for GRW sampling implemented on CPUs. We deploy it on a server with two 8-core Intel Xeon E5-2620v4 processors and 32 threads operating at 2.10GHz. Fig. 6 depicts the comparison results, where all numbers are normalized to the KnightKing. The static and dynamic samplings results are reported separately. For a fair comparison, we measure KnightKing with only sampling latency. On average, FastRW achieves 14.13× and 36.71× speedups on static and dynamic sampling compared to KnightKing. The higher speedup is achieved on dynamic sampling, which can be attributed to two reasons: i) KnightKing suffers severe performance loss due to inefficient execution flow. On the contrary, FastRW leverages the advantage of FPGA to customize the data path and achieve a pipelining dataflow. ii) The hybrid memory architecture and bit-level structure unit design in FastRW can check connectivity between vertices quickly.

*2) **FastRW vs C-SAW**:* C-SAW [9] is evaluated on NVIDIA Teala V100 GPU with 900GB/s bandwidth operating at 1.455GHz. The thermal design power (TDP) is 250W, and we use it to estimate GPU power. FastRW is compared against C-SAW implemented on 1 GPU and 6 GPUs. The performance of C-SAW is from the published paper, and RC and WT datasets are not evaluated. As shown in Fig. 6, in terms of performance, on average, FastRW achieves 10.48× and 8.15× speedup compared to C-SAW with 1 GPU and 6 GPUs, respectively. For energy efficiency, FastRW achieves remarkable benefits of 3.28×∼198.24× compared with C-SAW implemented on a single GPU. For the LJ dataset, although FastRW obtains a lower performance, it still achieves a significant energy efficiency benefit of 3.28× compared to C-SAW. The reasons that FastRW can achieve higher gains are as follows: i) FastRW customizes an alias-based sampling architecture, while C-SAW supports ITS sampling, which requires expensive running overhead. ii) Although the targeted U50 card has a lower bandwidth than V100 GPU, 316GB/s vs. 900GB/s, the isolated memory design (Section IV.A) in FastRW enables accurate accesses and no conflicts between multiple PEs, showing the advantages of customization.

C. Effectiveness of FastRW Designs

*1) **Dataflow and Pipeline Specialization**:* We further evaluate the effectiveness of FastRW by comparing it with two baselines: i) ***FastRW-base***: The basic architecture that suffers

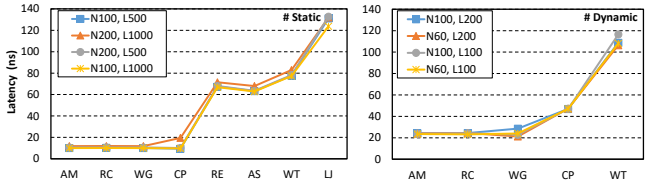

Fig. 8. The sampling latency under different parameter settings (N: the number of walkers. L: the sampling length of walkers.)

inefficient dataflow. Previous FPGA-based work [13] also adopts a similar design. ii) ***FastRW-op1***: The architecture that integrates the dataflow scheduling to mask computation latency. iii) ***FastRW-final***: The architecture that integrates the dataflow scheduling and multi-issue design. To make a fair comparison, we deploy them on the same U50 platform. Fig. 7 shows the comparison results, where less latency corresponds to higher sampling performance.

For static sampling, compared to *FastRW-base* and *FastRW-op1*, *FastRW-final* achieves 22.12× and 6.50× speedup on average, showing the effectiveness of the dataflow scheduling and pipeline specialization. To be specific, *FastRW-final* achieves 38.75×∼40.24× speedup on the small-scale dataset (e.g., AM, RC), and achieves 3.12×∼6.21× speedup on the larger dataset (e.g., WT, LJ) compared to *FastRW-base*. The main reason is that sampling latency is affected by the stride of random access. Larger datasets correspond to larger strides, resulting in longer sampling latency. For dynamic sampling, *FastRW-final* achieves 40.01× and 4.14× speedup on average compared with *FastRW-base* and *FastRW-op1* baselines.

*2) **Bit-width Customization**:* Each PE occupies several private HBM channels for access. The alias table with larger bit width and more quantities requires the vast majority of channels for its intensive accesses. So the number of PEs is affected by the storage size of the alias table. Our architecture with customized bit width can reduce the storage of the alias table by 23.81% and 21.32% for static and dynamic samplings, respectively. Finally, FastRW can deploy up to 6 and 3 parallel PEs for static and dynamic samplings. The number of parallel PEs for dynamic sampling is less due to it requires additional channels for connectivity checking.

D. Sensitivity of FastRW

We investigate the sampling efficiency of FastRW by adjusting the numbers and length of sampling walkers in a single PE. Fig. 8 depicts the sampling latency of FastRW under different parameter settings. Within the margin of error, the length and number have little effect on the sampling efficiency of FastRW, reflecting its excellent scalability. Certainly, it is crucial that the number of walkers is greater than the threshold to enable the correction of the pipeline, as mentioned in Section III. The selection of the sampling length should also be large enough to mask the start-stop overhead of the pipeline.

VI. Conclusion

This paper presents FastRW, a dedicated accelerator to release GRW acceleration on FPGAs. FastRW first schedules walkers' execution to address data dependency and leverages pipeline and bit-level optimization to customize a processing engine and achieve efficient dataflow. Finally, FastRW implements a hybrid memory architecture adapted to differential vertex accesses. The results show FastRW achieves a 14.13× speedup on average over the CPU system and 3.28×∼198.24× energy efficiency over the GPU implementation.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grants 2017YFA0700900 and 2017YFA0700903, in part by the National Natural Science Foundation of China under Grants 62102383, 61976200, and 62172380, in part by Jiangsu Provincial Natural Science Foundation under Grant BK20210123, in part by Youth Innovation Promotion Association CAS under Grant Y2021121, and in part by the USTC Research Funds of the Double First-Class Initiative under Grant YD2150002005.

References

[1] H. Zeng, H. Zhou, and A. S. et al., "Graphsaint: Graph sampling based inductive learning method," in *8th International Conference on Learning Representations, ICLR*, 2020.

[2] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, 2014, p. 701–710.

[3] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016, p. 855–864.

[4] A. Kyrola, "Drunkardmob: Billions of random walks on just a pc," in *Proceedings of the 7th ACM Conference on Recommender Systems*, ser. RecSys '13, 2013, p. 257–264.

[5] M. Wang, L. Yu, and D. Z. et al., "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019.

[6] Y. Ke, Z. MingXing, and C. K. et al., "Knightking: A fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019, p. 524–537.

[7] J. Abhinav, P. Sandeep, and G. A. et al., "Accelerating graph sampling for graph machine learning using gpus," ser. EuroSys '21, 2021, p. 311–326.

[8] R. Wang, Y. Li, and H. X. et al., "GraphWalker: An I/O-Efficient and Resource-Friendly graph analytic system for fast and scalable random walks," in *2020 USENIX Annual Technical Conference*, 2020, pp. 559–571.

[9] S. Pandey, L. Li, and A. H. et al., "C-SAW: a framework for graph sampling and random walk on gpus," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020*, 2020.

[10] C. Wang, L. Gong, X. Li, and X. Zhou, "A ubiquitous machine learning accelerator with automatic parallelization on fpga," *IEEE Transactions on Parallel and Distributed Systems*, pp. 2346–2359, 2020.

[11] C. Wang, L. Gong, F. Jia, and X. Zhou, "An fpga based accelerator for clustering algorithms with custom instructions," *IEEE Transactions on Computers*, pp. 725–732, 2021.

[12] C. Wang, L. Gong, X. Ma, X. Li, and X. Zhou, "Wookong: A ubiquitous accelerator for recommendation algorithms with custom instruction sets on fpga," *IEEE Transactions on Computers*, 2020.

[13] C. Su, H. Liang, and W. Z. et al., "Graph sampling with fast random walker on hbm-enabled FPGA accelerators," in *31st International Conference on Field-Programmable Logic and Applications, FPL 2021*, 2021, pp. 211–218.

[14] J. E. Gonzalez, Y. Low, and H. G. et al., "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012*, 2012, pp. 17–30.

[15] C. Xie, L. Yan, W. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Advances in Neural Information Processing Systems 27*, 2014, pp. 1673–1681.