LightRW: FPGA Accelerated Graph Dynamic Random Walks

HONGSHI TAN, National University of Singapore, Singapore
XINYU CHEN, National University of Singapore, Singapore
YAO CHEN, National University of Singapore, Singapore
BINGSHENG HE, National University of Singapore, Singapore
WENG-FAI WONG, National University of Singapore, Singapore

Graph dynamic random walks (GDRWs) have recently emerged as a powerful paradigm for graph analytics and learning applications, including graph embedding and graph neural networks. Despite the fact that many existing studies optimize the performance of GDRWs on multi-core CPUs, massive random memory accesses and costly synchronizations cause severe resource underutilization, and the processing of GDRWs is usually the key performance bottleneck in many graph applications. This paper studies an alternative architecture, FPGA, to address these issues in GDRWs, as FPGA has the ability of hardware customization so that we are able to explore fine-grained pipeline execution and specialized memory access optimizations. Specifically, we propose LightRW, a novel FPGA-based accelerator for GDRWs. LightRW embraces a series of optimizations to enable fine-grained pipeline execution on the chip and to exploit the massive parallelism of FPGA while significantly reducing memory accesses. As current commonly used sampling methods in GDRWs do not efficiently support fine-grained pipeline execution, we develop a parallelized reservoir sampling method to sample multiple vertices per cycle for efficient pipeline execution. To address the random memory access issues, we propose a degree-aware configurable caching method that buffers hot vertices on-chip to alleviate random memory accesses and a dynamic burst access engine that efficiently retrieves neighbors. Experimental results show that our optimization techniques are able to improve the performance of GDRWs on FPGA significantly. Moreover, LightRW delivers up to 9.55× and 9.10× speedup over the state-of-the-art CPU-based MetaPath and Node2vec random walks, respectively. This work is open-sourced on GitHub at https://github.com/Xtra-Computing/LightRW.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Information systems** → **Query operators**; • **Computer systems organization** → *Data flow architectures*; • **Mathematics of computing** → *Graph algorithms.*

Additional Key Words and Phrases: FPGA accelerator; parallel weighted reservoir sampling; random walk on graphs.

ACM Reference Format:
Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2023. LightRW: FPGA Accelerated Graph Dynamic Random Walks. *Proc. ACM Manag. Data* 1, 1, Article 90 (May 2023), 27 pages. https://doi.org/10.1145/3588944

1 INTRODUCTION

Random walk on graphs has been widely adopted by many applications, such as the recommendation system [23], bioinformatics [60], and network analysis [39]. Due to the ineffectiveness of

Authors' addresses: Hongshi Tan, hongshi@comp.nus.edu.sg, National University of Singapore, Singapore; Xinyu Chen, xinyuc@comp.nus.edu.sg, National University of Singapore, Singapore; Yao Chen, yaochen@nus.edu.sg, National University of Singapore, Singapore; Bingsheng He, hebs@comp.nus.edu.sg, National University of Singapore, Singapore; Weng-Fai Wong, wongwf@nus.edu.sg, National University of Singapore, Singapore.

2836-6573/2023/5-ART90
https://doi.org/10.1145/3588944

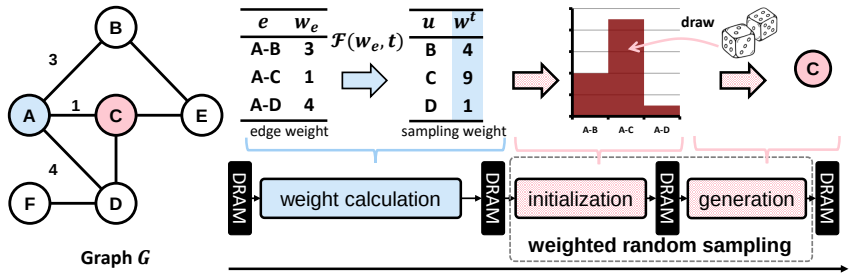


Fig. 1. Processing flow of GDRW on CPUs.

traditional random walk algorithms (i.e., uniform random walk and DeepWalk [44], etc.) in extracting temporal relations with structural information of the graph, graph dynamic random walks (GDRWs) have attracted increasing attention [19, 25, 26, 42, 58, 59]. For example, Node2Vec [25] has been widely used to generate graph embeddings for downstream machine learning models [42], and MetaPath [19] plays an important role in knowledge graph analytics [66]. However, due to inherent random memory accesses and high computational complexity, GDRWs generally dominate the overall execution time of these applications. For example, the state-of-the-art approach, ThunderRW by Sun et al. [54], showed that Node2Vec [25] can take up to two hours on a graph of 41 million vertices and 1.21 billion edges on a modern multi-core machine. Due to the explosive growth of graph data and the rising popularity of GDRWs in numerous applications, there is an urgent need to accelerate the performance of GDRWs.

Extensive efforts [54, 65] have been made to optimize GDRWs on multicore CPUs. For example, Yang et al. [65] explore the parallelism of GDRWs in a distributed environment by efficient graph partitioning techniques. ThunderRW [54] develops a stage design and interleaves memory accesses from massive queries for high memory efficiency and achieves state-of-the-art performance on modern CPU hardware. Random walk queries are processed concurrently with multiple threads, and each thread handles different queries with the execution flow illustrated in Figure 1. For a query starting from vertex A in graph G to move one step forward, *weight_calculation* traverses all neighbors of A at the start and then calculates the sampling weights using an application-specific weight update function $\mathcal{F}$, e.g., the sampling weights of vertex B, C and D ($\{4,9,1\}$) are calculated by $\mathcal{F}$ with the corresponding edge weights $\{3,1,4\}$ and the current state t. Then, weighted random sampling is conducted in two procedures. First, *initialization* builds an indexing table from the sampling weight, which describes the distribution of all neighbors and is stored in global memory. Second, *generation* generates uniformly distributed random samples to index the corresponding vertex. Then, the indexed vertex C is marked as the starting vertex for the next step. This process is repeated until it satisfies a specific termination condition, such as a target length being reached.

Despite those existing studies and optimizations for multi-core architectures, there are fundamental limitations in microarchitecture level to implementing GDRWs on multicore CPUs efficiently. Our comprehensive profiling on ThunderRW shows that even with advanced optimizations such as explicit memory prefetching, 59.9% of execution cycles are still stalled by memory accesses, with a cache miss ratio as high as 58.2% (see more details in Section 2.3). The irregular nature of graphs [49] results in GDRWs having extremely poor spatial and temporal locality, rendering the multicore's complex cache hierarchy ineffective. Moreover, GDRWs on multicore architectures have to be executed in stages, with barriers required at the end of each stage. Intermediate data need to be written to DRAM after the execution of one stage and read again for the next stage.

This paper studies an alternative architecture, FPGA, to address those issues in GDRWs. Different from the multicore CPUs that have fixed architectures with a deep cache hierarchy, FPGAs have the flexibility to customize hardware logic, which allows us to explore novel execution schemes

such as pipeline execution on chip with fine-grained inter-stage communication (referred to herein as fine-grained pipeline execution) to minimize data movement overhead and design specialized memory engines to handle random memory accesses in GDRWs efficiently. Specifically, we propose LightRW, the first FPGA-based acceleration framework for efficient GDRWs. LightRW creates a fully pipelined and parallelized FPGA-based accelerator system for GDRWs. The major contributions of our LightRW framework are as follows:

- We design a parallelized Weighted Reservoir Sampling (WRS) on FPGA, which is able to process multiple vertices per cycle and enables fine-grained communication between stages so that the different stages are executed in parallel and communicate within the chip without retrieving the DRAM.
- We propose two memory efficient techniques to handle notorious random accesses to neighbors of vertices: 1) a degree-aware cache keeps high-degree vertices associated data (e.g., address of their neighbors) into on-chip memory at runtime, and 2) a burst access engine adjusts the burst access size to minimize redundant data access for higher memory bandwidth utilization.
- Our evaluation of LightRW on a server with an FPGA board attached to the PCIe interface outperforms the state-of-the-art CPU-based implementation [54] by up to 9.55× and 9.10× on MetaPath [55] and Node2Vec [25], even with the consideration of the data transfer overhead over PCIe, respectively. Furthermore, LightRW is 15× ∼ 26× more power efficient than the state-of-the-art CPU-based implementation.

The rest of the paper is organized as follows. We introduce the background and motivation in Section 2. Section 3 presents the overview of LightRW, including the proposed pipelined GDRW algorithm and hardware architecture. While Section 4 illustrates the design of the parallel WRS sampler, Section 5 introduces two GDRW specific memory optimizations. We present the experimental results in Section 6 and related works in Section 7. This paper is concluded in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background of graph dynamic random walks, discuss the state-of-the-art CPU-based solutions and motivate our FPGA-based accelerator design according to the analysis on performance bottlenecks of CPU-based solutions.

2.1 Graph Dynamic Random Walks

Let $G = \{V, E\}$ represent a directed graph with $|V|$ vertices and $|E|$ edges. $N(u)$ is refered to a set of all neighbors of u, and $|N(u)|$ be the total number of neighbors (the degree). Let d_u^+ and d_u^- be the out-degree and in-degree of u respectively The undirected graphs are supported by representing each undirected edge with two directed edges with the same two vertices. The edge between the vertex u and the vertex v is represented as (u, v), and the corresponding edge weight is represented by $w_{u,v}^*$, in the case of unweighted graphs, $w_{u,v}^* = 1$.

A random walk on a graph begins at a starting vertex and moves to a randomly selected neighbor of the currently residing vertex at each step. Based on how the edge (or the neighbor) is selected, random walks are further categorized into *unbiased* or *biased* ones. The transition probability of an edge is defined as the probability of the corresponding edge being sampled, and we refer to the unnormalized form of the transition probability as the sampling weight. For unbiased random walks, the transition probability is uniformly distributed, while for biased random walks, the transition probability is generally related to the edge weight. Furthermore, for biased random walks, if the edge weights remain constant throughout the process, they are called *static* random walks. In contrast, *graph dynamic random walks* (GDRWs) take into account the walker's state by updating the edge weights at each step to recalibrate the transition probability. As in static random walks, per-edge

transition probabilities could be calculated offline, and many existing studies take advantage of this to simplify online calculation [44, 54, 65]. In comparison, GDRWs require more advanced and challenging optimizations at runtime, which are the focus of this paper. Moreover, with the modeling on the walker's state, GDRWs are able to extract higher-order structural information and temporal relationships in graphs. Therefore, GDRWs have attracted increasing attention for graph learning applications.

Next, we introduce two representative GDRW algorithms: MetaPath and Node2Vec. Let $w^t_{a,b}$ be the sampling weight of edge (a, b) to be sampled from the current vertex a at step t. $w^t_{a,b}$ is updated by an application-specific weight update function $\mathcal{F}$, i.e., $w^t_{a,b} = \mathcal{F}(w^*_{a,b}, V_{t-1})$, where V_{t-1} is the set of traversed vertices.

MetaPath [55] is an effective approach for data mining on heterogeneous graphs. A MetaPath M is defined as $L_1 \xrightarrow{R_1} L_2 \xrightarrow{R_2} ... \xrightarrow{R_t} L_{t+1}$, where L_i is the vertex label in the i-th step, and R_i is the relation between L_i and L_{i+1} (e.g., a book is cited by an author). A MetaPath random walk query returns a sampled path based on the given relation path $\mathcal{R} = R_1, R_2, ..., R_t$. The weight update function is shown in Equation (1). For the t-th step, For the t-th step, the weight of the path to be sampled is set according to whether the relationship is met, as shown in Equation (1a). Otherwise, the corresponding weight is set to zero, indicating that the path will not be sampled in this step (Equation (1b)).

$$w^t_{a,b} = \begin{cases} w^*_{a,b}, & \text{if } ((a,b) \in E) \wedge (R_{(a,b)} = \mathcal{R}[t]), \quad (1a) \\ 0, & \text{otherwise.} \quad (1b) \end{cases}$$

Node2Vec [25] is a second-order random walk [15] and has been widely used as a graph embedding technique [24]. The transition probability depends not only on the current vertex a^t but also on the previously traversed vertex a^{t-1}. The weight of neighbors depends on whether it is connected to the last traversed vertex. In cases where the neighbor b is the same as the vertex visited in the previous step, the sampling weight is equal to the scaled edge weight with a hyperparameter p (Equation (2a)). Apart from that, if there is an edge between a^{t-1} and b, the sampling weight is equal to the edge weight (Equation (2b)); otherwise, the sampling weight is scaled with another factor q in Equation (2c). Finally, for all non-neighboring vertices, the sampling weights are set to zero (Equation (2d)).

$$w^t_{a,b} = \begin{cases} \dfrac{w^*_{a,b}}{p}, & \text{if } b = a^{t-1}, \quad (2a) \\ w^*_{a,b}, & \text{if } (b \neq a^{t-1}) \wedge ((a^{t-1}, b) \in E), \quad (2b) \\ \dfrac{w^*_{a,b}}{q}, & \text{if } (b \neq a^{t-1}) \wedge ((a^{t-1}, b) \notin E), \quad (2c) \\ 0, & \text{otherwise.} \quad (2d) \end{cases}$$

2.2 GDRW on CPUs

The execution flow of the CPU-based GDRW [54] is shown in Algorithm 2.1. For an input query composed of a starting vertex v_{curr} and the requested length, the algorithm first ascertains the number of neighbors of the current vertex v_{curr} for indexing in Line 6. For each neighbor of v_{curr}, the algorithm first loads its properties (Line 8) and then updates the weight of each neighbor using the application-specific function (Line 9). Different GDRW applications can be implemented by customizing the application-specific weight update function. The updated weights are used

Algorithm 2.1: CPU-based GDRWs

Input: G: a given graph, $\mathbb{Q}$: a set of input queries.
Output: V_t: a path of traversed vertices.

```
1  res = ∅;
2  foreach Q ∈ ℚ do
3      v_curr = Q.v_start, Q.res = ∅;
4      loop
5          w_sum = 0, W = ∅, N_v = ∅ ;
           /* weight_calculation */
6          {address, degree} = get_neighbors_info(v_curr, G);
7          for i ← 1 to degree do
8              N_v^i = get_neighbor(address, i, G);
9              w_v^i = app_weight_update(N_v^i, v_curr);
10             W.push(w_v^i);
           /* weighted_sampling */
11         T = initialization(W) ;                    // O(n) time complexity
12         v_curr = generation(T);
13         Q.res.push(v_curr);
14         if (Q.is_end()) then
15             res.push(Q.res);
16             break;
17 return res;
```

to implicitly calculate the transition probability, which is the probability of the neighbors to be selected.

In the next stage, weighted sampling draws one sample based on the generated weight distribution W. This process is carried out in two phases: *initialization* and *generation*, which are commonly used in sampling methods including inverse transformation sampling [54] and alias sampling [29]. Since the updated weight is discretely distributed, the initialization stage constructs an intermediate table T that describes the distribution of W, which allows for the sampling of weighted items with uniformly distributed random numbers (Line 11). Finally, in Lines 12 and 13, the generation stage randomly draws one neighbor based on T and appends it to the output sequence $Q.res$, which is then set as v_{curr} for the next iteration. This process is repeated until v_{curr} satisfies the given terminal state (e.g., the required path length). The state-of-the-art CPU-based system [54] adopts multiple threads to process different batches of queries in parallel. Each stage of a query is executed sequentially and interleaved with memory accesses using one thread, such that the latency of random access overlaps with computation.

2.3 Inefficiencies of CPU-based GDRWs

To further investigate performance potentials on multi-core architectures, we have conducted a top-down performance analysis [17] of the state-of-the-art CPU-based system, ThunderRW [54]. In particular, we evaluated MetaPath and Node2Vec with two widely used graphs in previous studies, livejournal [35] and uk-2002 [9], on a server equipped with the latest Intel Xeon Gold 6246R CPU. We used vTune [17] to profile the last-level cache miss ratio (denoted as "LLC Miss"), the proportion of cycles stalled by memory accesses (denoted as "Memory Bound"), and the proportion of cycles

Table 1. Profiling results of the state-of-the-art CPU-based system [54] on MetaPath and Node2Vec.

Applications	Graphs	LLC Miss	Memory Bound	Retiring Ratio
MetaPath	liveJournal [35]	**58.2%**	**59.9%**	**8.2%**
	uk-2002 [9]	**61.8%**	**57.5%**	**13.7%**
Node2Vec	liveJournal [35]	**76.9%**	**31.2%**	**23.3%**
	uk-2002 [9]	**61.1%**	**31.7%**	**33.6%**

used for useful computations (denoted as "Retiring Ratio"). More details about the experimental setup can be found in Section 6.

The profiling results show that the LLC miss ratio is very high (up to 76.9%), and the memory bound ranges from 31.2% to 59.9% for the two graphs in both applications. Subsequently, the retiring ratio is only 8.2% ∼ 33.6%, indicating poor utilization of CPU cores. In summary, memory accesses dominate the overall execution time for CPU-based GDRWs. Based on the execution flow on CPUs and the characteristics of GDRW, there are two key observations for the inefficiency:

Inefficiency 1: The sequential execution flow on CPUs introduces a large number of memory accesses. The three stages of GDRW are executed in sequence with barriers required at each stage on control flow-based multicore CPUs. Here, we provide a quantitative analysis of the number of memory accesses introduced by intermediate data access. The weight update stage of the neighbors requires storing updated weights with the size of $|N(v_{curr})|$ in memory. During the initialization stage, the updated weights are read and, a temporary data structure T is created with a size of $|N(v_{curr})|$. Therefore, in addition to the output results, the entire process ideally requires $2\times|N(v_{curr})|$ memory accesses. However, since the computations involved are typically lightweight, the costs of memory access cannot be overlapped. As a result, memory access tends to dominate the overall execution time.

Inefficiency 2: Irregular memory accesses are poorly handled. Due to the irregular nature of graphs, GDRW introduces massive irregular memory accesses, including random access to the addresses of neighbors of a vertex and access to varying numbers of neighbors, leading to poor data locality. Additionally, the size of intermediate data for existing weighted sampling methods (i.e., alias table [29] and inverse transform distribution table [61], etc.) is proportional to the number of neighbors. For large graphs, the space required to cache neighbors of different nodes varies significantly, which easily exceeds the capacity of any CPU cache, leading to cache thrashing. This results in significant cache misses and CPU core stalls. Moreover, with multiple queries executed with multiple threads, concurrent prefetching in the shared last-level cache exacerbates the memory access contention and cache thrashing.

2.4 Motivation and Design Rationales

The inefficiencies mentioned above are fundamentally caused by the sequential execution flow on multicore architectures and the mismatch between the memory access pattern of GDRW and the current cache hierarchy of multi-core architectures. It is clear that a "one size fits all" approach is not suitable. There is a need for a specialized solution in terms of both the algorithm and architecture. This paper studies radical and alternative architectures, starting with FPGA, a commodity hardware for specialized designs.

With the flexibility to customize hardware logic, FPGAs have demonstrated promising performance in many data-intensive applications [11–14, 27, 68]. Among them, a common optimization technique is fine-grained pipelining, which instantiates hardware units for different tasks (e.g.,

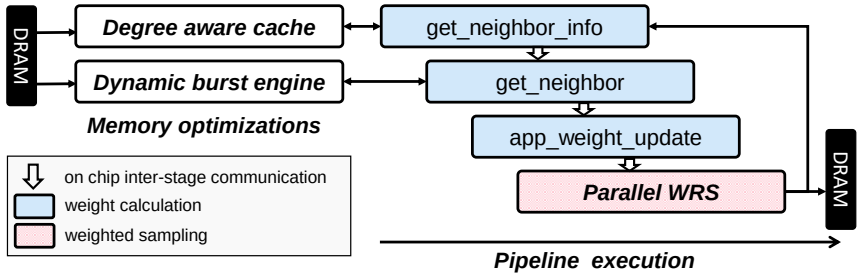


Fig. 2. Processing flow of LightRW.

functional stages in GDRW) on the chip and connects hardware units through FIFOs for fine-grained communication (e.g., a vertex per transfer). As a result, it minimizes the number of DRAM memory accesses and increases parallelism by simultaneous execution of hardware units in comparison to non-pipelined systems. Meanwhile, customization of the memory access engine (i.e., using scratch-pad instead of a complex cache hierarchy on CPUs) is another source of performance improvement. For example, the FPGA-based graph processing framework from Chen et al. [14] outperforms state-of-the-art solutions by pipelining the Scatter and Gather functions of the GAS model [14] and adopting application-specific memory access units. This motivates us to explore two directions to address the inefficiencies encountered in CPU-based GDRWs:

- Pipelining different stages of GDRW on FPGAs with fine-grained inter-stage communication to eliminate synchronization barriers and minimize data movement to DRAM.
- Designing specialized memory access engines with considerations on the irregular memory access patterns in GDRWs for high memory bandwidth utilization.

3 LIGHTRW

With the motivation and design rationales, we introduce LightRW, an FPGA-based accelerator for efficient GDRW.

3.1 Solution Overview

In short, the design of LightRW has two complementary approaches: 1) reducing the number of memory accesses to DRAM by enabling fine-grained pipeline execution of GDRW on FPGAs, and 2) handling random memory accesses efficiently with customized memory optimizations. Its processing flow is shown in Figure 2.

LightRW eliminates the synchronization barriers of existing GDRW algorithms to enable fine-grained pipeline execution on the chip by adopting the *weighted reservoir sampling* (WRS) technique that chooses a random sample in a single pass over the items. More importantly, we parallelize WRS to process multiple vertices per cycle for high throughput pipeline design. With fine-grained pipeline execution, LightRW reduces the number of memory accesses to DRAM and achieves higher spatial parallelism compared to existing GDRW solutions.

We also propose two memory-efficient techniques to handle notoriously random accesses to the neighbors of vertices. First, according to the power law distribution of graphs, vertices with high degrees dominate the connections and are frequently accessed. Thus, we propose a degree-aware cache that keeps information of high-degree vertices (e.g., the addresses of their neighbors) into on-chip memory at runtime. Second, we propose a dynamic burst access engine that adjusts the burst access size to minimize redundant data accesses and thus improve the use of available memory bandwidth, as the neighbors of different vertices have different lengths.

Algorithm 3.1: Dynamic Random Walk with WRS.

Input: G: a given graph, $\mathbb{Q}$: a set of input queries.
Output: V_t: a path of traversed vertices.

```
1  res = ∅;
2  foreach Q ∈ ℚ do
3      v_curr = Q.v_start, Q.res = ∅;
4      loop
5          {address, degree} = get_neighbors_info(v_curr, G);
6          for i ← 1 to degree do
               /* weight_calculation */
7              n_v = get_neighbor(address, i, G);
8              w = app_weight_update(n_v, G);
               /* weighted_sampling */
9              v_curr = WRS(n, w, R);
10         Q.res.push(v_curr);
11         if (Q.is_end()) then
12             res.push(Q.res);
13             break;
```

3.2 Pipelining GDRW with WRS for FPGAs

To explore the solution of fully pipelined GDRW for FPGAs, we revisit existing weighted sampling methods. We find that *weighted reservoir sampling* (WRS) meets our requirements [20] as it can choose n_{res} random samples from a population of unknown size in a single pass over the items. In particular, let w_i be the weight of the i-th item, and the probability that the i-th item is selected, denoted as p_i, is equal to its weight divided by the accumulated weights of all items passed, which is $p_i = \frac{w_i}{\sum_{m=1}^{i} w_m}$. If p_i is larger than a uniformly distributed random number r_i, the i-th item is then stored in a reservoir as a candidate for output, otherwise, it is discarded. As GDRWs only need to sample one vertex to move forward, n_{res} is set to one.

The streaming processing nature of WRS eliminates the synchronization barrier between the initialization and generation stages required in the existing commonly used sampling methods, such as inverse transformation sampling and bucket-based sampling approaches [29, 43, 54]. However, it introduces a huge computational cost, as WRS requires a random number for each item of the input, and the generation of random numbers on CPUs is time-consuming. This computational cost prevents WRS from being the best choice for the CPU-based GDRW system. For example, when we adopt WRS in optimal CPU-based implementations, we observe that the performance is 8.2 times worse. In contrast, generating massive random numbers on FPGAs is no longer a problem. With a novel state sharing technique, ThundRiNG [56] is able to generate high-quality and high-throughput random numbers with efficient resource utilization on FPGAs.

In this paper, we adopt WRS for GDRWs to enable fine-grained pipeline execution on FPGAs, as shown in Algorithm 3.1. Unlike CPU-based GDRWs shown in Algorithm 2.1, the weighted sampling method is changed to WRS. This enables us to fuse all the neighbor weight update and sampling functions into one loop, and each function can produce and consume a single item at a time to allow fine-grained communication between functions. We describe the detailed execution flow as follows: First, get_neighbors_info loads $\{address, degree\}$ of v_{curr} from graph G, where *address* is the location of v_{curr}'s neighbors in global memory, and *degree* is the total number of

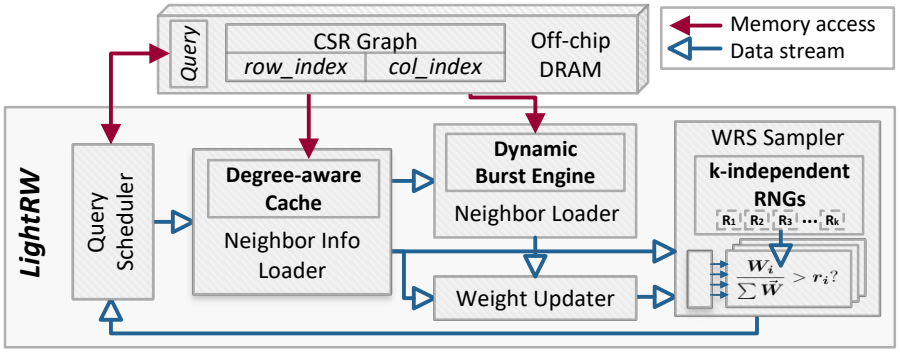


Fig. 3. Overview of hardware architecture.

v_{curr}'s neighbors (Line 5). Then, a for loop scans all v_{curr}'s neighbors and randomly selects one item from them (Lines 6 to 9). In the i-th iteration, it first loads the property of the i-th neighbor, N_v, and then calculates the sampling weight with app_update_weight, which is an application-specific function. WRS takes the updated weight as input to temporally sample one item as v_{next}. After all the items are enumerated, the selected v_{next} is updated to $Q.v_{curr}$ for the next step.

The intermediate variables of these functions, which are n_v, w, and v_{next}, consume $O(1)$ space and are stored on-chip. This is significantly smaller than the state-of-the-art execution flow required (i.e., $O(|N(v_{curr})|)$ [54]). The proposed algorithm eliminates the synchronization barrier among different functions. It also guarantees fine-grained pipeline execution on FPGAs, which reduces the number of memory accesses to DRAM by fine-grained communication on-chip and improves spatial parallelism by concurrently executing the computing stages.

3.3 Hardware Architecture

LightRW efficiently realizes the proposed GDRW on FPGAs with a highly optimized microarchitecture. While existing CPU-based GDRWs mainly focus on multi-query parallelization with task interleaving and multithreading, LightRW explicitly explores parallelism within the query by processing multiple neighbors per cycle.

Figure 3 shows an overview of LightRW's hardware architecture, which consists of the *query controller*, *degree-aware cache*, *neighbor info loader*, *dynamic burst engine*, *weight updater*, and *WRS sampler*. The target graphs are stored in the FPGA's DRAM with compressed sparse row (CSR) format, which consists of a *row_index* array and a *col_index* array. The *col_index* array records adjacent edges (sorted by destination vertex) of each vertex, while the *row_index* records the offset of adjacent edges of each vertex in the *col_index* array. The Query Controller loads multiple queries and prepares query metadata (e.g., the v_{curr}, query index, and application-specific parameters) for the next stage. The Neighbor Info Loader reads the address of the neighbors (adjacent edges) of v_{curr}, namely the *row_index* of v_{curr}. As v_{curr} is randomly selected, the accesses to the addresses of neighbors are random. Our proposed *Degree-aware Cache* caches vertices with high degrees to improve memory efficiency (details in 5.1). The Neighbor Loader sequentially loads the neighbors of v_{curr} by dereferencing the address of neighbors. However, the number of neighbors varies according to the vertices. The memory coalescing with a fixed burst length may result in redundant memory accesses. Therefore, we also propose a *Dynamic Burst Engine* that adopts hybrid burst lengths to maximize memory bandwidth utilization (details in 5.2). The Weight Updater calculates the sampling weight of the fetched neighbors using an application-specific update function. The *WRS Sampler* consumes multiple neighbors per cycle with throughput at the line rate of memory bandwidth and samples the vertex for the next step of the current query. Details on how to parallelize WRS

are given in Section 4. The Query Controller updates v_{curr} with the sampled vertex. All the above hardware modules are pipelined and run concurrently.

4 PARALLELIZING WRS

To process multiple neighbors per cycle for GDRWs, WRS has to be parallelized. However, data dependency in calculating the probability of the item in the reservoir prevents parallelization. As discussed in Section 3.2, the probability of the i-th item, p_i, is calculated as its weight divided by the accumulated weight of all passed items, i.e., $p_i = \frac{w_i}{\sum_{m=1}^{i} w_m}$. In other words, the probability calculation of the current item is dependent on the weights of the previously processed items, which prevents straightforward parallelization of the probability calculation. For instance, simple loop unrolling cannot be applied due to this dependency [28]. Existing works on paralleling WRS [29, 30] aim to reduce the number of required random numbers for each machine in a distributed environment, as random number generation is usually time-consuming for CPUs. On the contrary, FPGAs can easily generate multiple independent random numbers benefiting from spatial parallelism, high-performance bitwise operations, and flexible on-chip communication [56]. Therefore, in this paper, we propose a new parallelized WRS algorithm and its FPGA-based implementation.

4.1 Parallel WRS Algorithm

The key idea of our proposal is to consume multiple items per cycle while carefully handling their dependency to ensure the independence and correctness of sampling. As each item in the stream requires the accumulated weight of all items passed, namely $w_{sum}^{i} = \sum_{m=1}^{i} w_m$, we employ a parallelized prefix sum to calculate the accumulated weights for multiple items in parallel. Specifically, assuming that we process k items per cycle and i items have passed, the accumulated weights for the current k items can be represented by the set $\{\sum_{m=1}^{i+j} w_m\}_{j=1}^{k}$. By decomposing the sum of the weights of the passed i items, we have the following representation of the set $\{\sum_{m=1}^{i+j} w_m\}_{j=1}^{k}$,

$$\{\sum_{m=1}^{i+j} w_m\}_{j=1}^{k} = \{\sum_{m=1}^{i} w_m + \sum_{m=i+1}^{i+j} w_m\}_{j=1}^{k}. \tag{3}$$

As $w_{sum}^{i} = \sum_{m=1}^{i} w_m$, which is a constant for current k items, we have

$$\{\sum_{m=1}^{i+j} w_m\}_{j=1}^{k} = \{w_{sum}^{i} + \sum_{m=i+1}^{i+j} w_m\}_{j=1}^{k} \tag{4}$$

$$= w_{sum}^{i} + \{\sum_{m=i+1}^{i+j} w_m\}_{j=1}^{k}, \tag{5}$$

where $\{\sum_{m=i+1}^{i+j} w_m\}_{j=1}^{k}$ is the prefix sum of $\{w_{i+1}, w_{i+2}, ...w_{i+k}\}$ that can be calculated in parallel.

Algorithm 4.1 shows the proposed parallelized WRS that can process k items per cycle. The inputs consist of the item stream $N_v = \{n_{v1}, n_{v2}, ...\}$, weight stream $W = \{w_1, w_2, ...\}$, the degree of parallelism k, and k-wise independent random variables uniformly distributed in the interval $[0, 1]$, $R = \{R_1, R_2, ..., R_k\}$. The algorithm reads k items and the corresponding weights per cycle (Line 3). Line 4 calculates the prefix sum of the k items, which corresponds to the second term $\{\sum_{m=i+1}^{i+j} w_m\}_{j=1}^{k}$ in Equation (5), and stores it in the array W_{ps}. Then, k items are sampled concurrently. For each item in $\hat{N_k}$, Line 7 calculates the probability of the j-th item being selected. Next, a random number r is sampled from the j-th random variable in R (Line 8). If p is smaller than r, its index is stored in *candidate* (Lines 9 to 10), indicating that the j-th item is temporarily

Algorithm 4.1: Parallel Weighted Reservoir Sampling.

Input:

$N_v = \{n_{v_1}, n_{v_2}, ...\}$: item stream to be sampled;
$W = \{w_1, w_2, ...\}$: corresponding weight stream;
k : degree of parallelism;
$R = \{r_1, r_2, ..., r_k\}$: k-wise independent random variables.

Output:

n_s : item sampled with probability of $\frac{w_s}{\sum W}$.

1 $w^i_{sum} = 0$, $reservoir = \emptyset$, $n_s = \emptyset$;
2 **do**
  /* get k items and corresponding weights from streams */
3   **if** $(\hat{N_k} = \textbf{receive}(N_v, k)) \wedge (\hat{W_k} = \textbf{receive}(W, k))$ **then**
    /* calculate the prefix sum array of $\hat{W_k}$ */
4     $W_{ps} = \textbf{prefix\_sum}(\hat{W_k})$;
5     $candidate = \emptyset$;
6   **parallel for** $j \leftarrow 1$ **to** k
      /* calculate the probability of selecting $\hat{W_k}[j]$ */
7       $p = \frac{\hat{W_k}[j]}{w^i_{sum} + W_{ps}[j]}$;
8       $r = R[j].\textbf{sample}(0, 1)$;
9   **if** $p > r$ **then**
10        $candidate[j] = j$;
    /* find the max index in candidate */
11    $sel = \textbf{max}(candidate)$;
    /* update the reservoir */
12   **if** $sel \neq 0$ **then**
13      $reservoir = \hat{N_k}[sel]$;
    /* update w^i_{sum} value for next k items */
14    $w^i_{sum} = w^i_{sum} + \textbf{sum}(\hat{W_k})$;
15    $n_s = reservoir$;
16   **output** n_s;
17 **while** N_v *is end*;
18 **return** n_s

selected. After all k items are processed, the maximum value in *candidate*, which is the index of the latest sampled item, is outputted to update the *reservoir* (Lines 11 to 13). Meanwhile, w^i_{sum} is accumulated with $sum(\hat{W_k})$ (Line 14) for the next batch of sampling. For a stream with n items, the time complexity of the proposed algorithm is $O(n/k + \log k)$, where the functions *prefix_sum()*, *max()*, and *sum()* introduce the $O(\log k)$ time complexity. Since only *reservoir* needs to be stored for each k input, the space complexity of the proposed algorithm is $O(1)$.

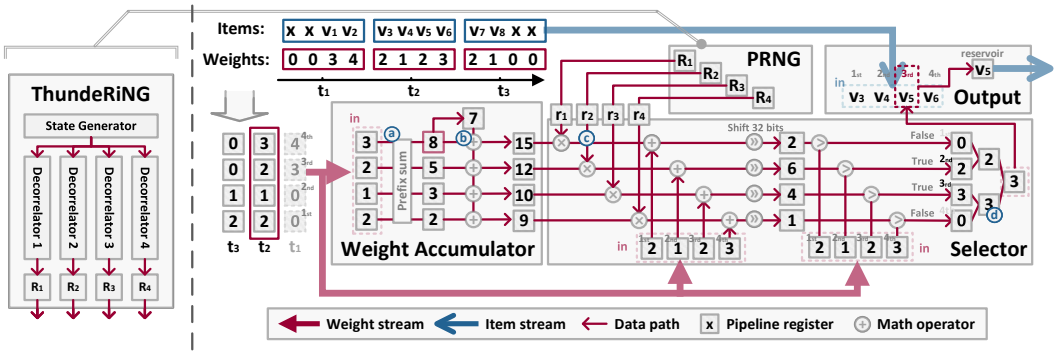


Fig. 4. Hardware architecture of WRS Sampler.

4.2 WRS Sampler

Deploying our proposed parallel WRS algorithm on FPGAs highly relies on high-quality and high-throughput random number generation (RNG). Therefore, we first select a suitable RNG module. After that, we introduce the hardware architecture of the WRS Sampler module for our proposed parallel WRS algorithm.

Selection of RNGs. RNGs can be classified into two types: true random number generations (TRNGs)[52] and pseudo-random number generations (PRNGs)[40]. TRNGs produce unpredictable and unreproducible outputs, while PRNGs are commonly used in randomized algorithms such as Monte Carlo simulations [48] and graph random walks [51] to ensure the reproducibility of results. In recent years, FPGA-based PRNGs have gained attention from both industry and academia [5, 37].

To enable high-performance WRS, we have summarized the following criteria for selecting PRNGs: First, the PRNGs must be capable of generating multiple independent sequences of random numbers to ensure the independent selection of the candidates in Lines 6 to 10 of Algorithm 4.1. Second, the generation of random numbers should be fast enough to avoid blocking the overall processing pipeline. Third, the hardware resources required to construct the PRNGs should not constitute a severe conflict with respect to the resource demand for GDRW processing.

To meet all of the aforementioned criteria, we have selected ThunderRiNG [56] and integrated it into our WRS pipeline, as shown in Figure 4. ThunderRiNG allows for the generation of multiple independent sequences of random numbers while sharing the costly state generation among different instances. Additionally, each instance adopts a decorrelator to generate a high-quality random number sequence. ThunderRiNG also provides an easy-to-use interface for integration. Our evaluation has demonstrated that ThunderRiNG is capable of simultaneously generating up to 64 sequences of random numbers, consuming only 1.2% of hardware resources, and passing the most stringent empirical randomness tests [33].

WRS Sampler Hardware Architecture. Figure 4 illustrates the hardware architecture of the WRS Sampler that realizes the proposed parallelized WRS. It includes four hardware modules: *Weight Accumulator*, *Selector*, *PRNG*, and *Output*. To facilitate presentation, we set k to four. The *Weight Accumulator* reads k items from the weight stream in one cycle (e.g., $\{2, 1, 2, 3\}$ at time t_2) and calculates the accumulated weights of k items in two steps. First, it calculates W_{ps} using pipelined prefix sum logic (Step ⓐ). Then, in Step ⓑ, it calculates $\{w^i_{sum} + W_{ps}[j]\}^k_{j=1}$ using four adders in parallel, and the last prefix sum (e.g., eight in the figure) is added to $wsum$ to calculate the next k items. The *Selector* performs the sampling procedure of k items per cycle. To avoid costly division operations in calculating p, we rewrite the comparison between p and the random number in the interval of $[0, 1]$, r, as follows: First, let $r*$ be a 32-bit integer random number generated from

ThundeRiNG. The condition for sampling the j-th item is as follows,

$$p > r \Rightarrow \frac{\hat{W}_k[j]}{w^i_{sum} + W_{ps}[j]} > \frac{r^*}{2^{32}-1}. \tag{6}$$

Then, by moving the denominators from both sides to the numerators on the other side, we obtain the following inequality:

$$\Rightarrow \hat{W}_k[j](2^{32}-1) > r^*(w^i_{sum} + W_{ps}[j]). \tag{7}$$

We further reorganize the left term,

$$\Rightarrow 2^{32} \cdot \hat{W}_k[j] > r^*(w^i_{sum} + W_{ps}[j]) + \hat{W}_k[j]. \tag{8}$$

The comparison in Equation (8) can be further simplified as follows. Multiplication in the left term can be obtained by light-weight bit shifting on FPGAs. As $\{w^i_{sum} + W_{ps}[j]\}^k_{j=1}$ is the output of the Weight Accumulator, the right term only needs one multiplication and one addition (Step ⓒ), which can be completed simultaneously by the DSP slice in FPGAs [46]. If the comparison condition is true, which means the item is a candidate for output, we record the indices of these items (e.g., two and three in this example). Finally, we determine the latest candidate by comparing their indices. This is implemented using a tree-based comparator (Step ⓓ), where each level compares two adjacent items. Based on the index of the selected item, *Output* extracts the corresponding item from the input item stream and updates *reservoir*. In the example, the third item, v_5, is selected in the *reservoir*. When the stream ends, *reservoir* is output as the sampled result.

5 MEMORY OPTIMIZATIONS

Memory accesses to DRAM are highly optimized in LighRW. First, LightRW employs a degree-aware cache that stores only the high-degree vertices in on-chip memory. Second, since neighbors of different vertices have varying lengths, we propose a dynamic burst access engine that adjusts the burst access size to minimize redundant data access, thereby improving memory bandwidth utilization.

5.1 Degree-aware Cache

Accesses to the addresses of neighbors of the current vertex, namely the accesses to the *row_index* array in the Neighbor Info Loader component, are inherently random, as the current vertices from different steps are selected randomly. Due to the large reuse distance [8], existing cache policies (i.e., LRU and FIFO, etc.) that are designed to retain the most recent or frequently-used data items in on-chip memory are ineffective in handling this access. In this paper, we provide an analysis of how the degree of a vertex can indicate the reuse ratio of a vertex in GDRWs. Based on this analysis, we propose a degree-aware cache to cache the high-degree vertices at runtime with zero initialization overhead.

Buffering high-degree vertices into fast memory for graph processing has been adopted in many recent research works [3, 6, 67]. These approaches are based on the observation that graph processing applications frequently access the properties of vertices with a larger number of neighbors. For instance, Zhao et al.[67] build a hash lookup table for high-degree vertices during graph partitioning, while Balaji and Lucia [6] sort the vertices by their degrees in the preprocessing phase and then reindex all vertices during the graph data preprocessing. However, all of these existing works introduce additional initialization overhead and are only applicable for graph processing. Instead, we explore the degree-aware caching method for GDRWs.

Probability Analysis of Vertex Accesses: We begin by presenting the analysis that the degree of a vertex has a positive correlation with the probability of the vertex being traversed in GDRWs,

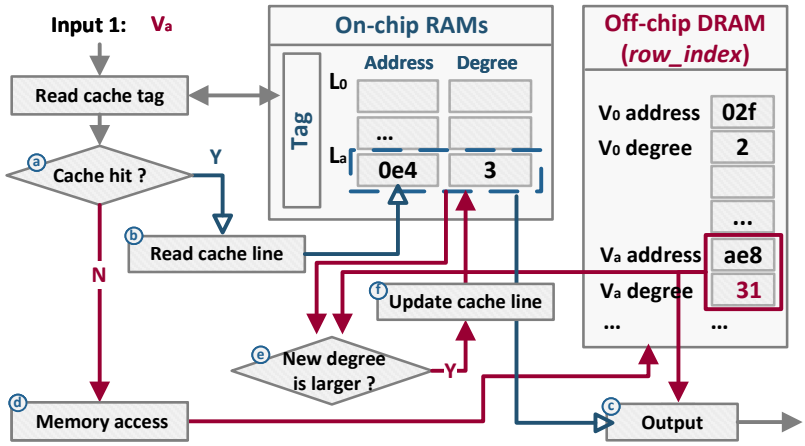


Fig. 5. Hardware architecture of degree-aware cache.

by extending the existing conclusion on static random walks. Let $Pr[v]$ be the probability that v is traversed by multiple independent random walk queries on a static weighted graph and follows a stationary distribution [51]. Then, $Pr[v]$ can be represented as follows:

$$Pr[v] = \frac{\sum_{u \in N(v)} w_{v,u}}{\sum_{i \in V} \sum_{j \in N(i)} w_{i,j}}. \tag{9}$$

Next, we consider dynamic random walks, where the sampling weight from vertex u to vertex v at time t is $w^t_{u,v}$. The stationary distribution of v based on the sampling weights at time t is then given by

$$Pr[v] = \frac{\sum_{u \in N(v)} w^t_{v,u}}{\sum_{i \in V} \sum_{j \in N(i)} w^t_{i,j}}. \tag{10}$$

Let $\mathcal{W}$ be defined as $\sum_{i \in V} \sum_{j \in N(i)} w^t_{i,j}$, and let α be a scaling factor that scales the minimal non-zero weight to one. Since graphs have a finite number of edges, α exists and is constant for a given graph. Similarly, $\mathcal{W}$ is also a constant value. Therefore, we have:

$$Pr[v] = \frac{\sum_{u \in N(v)} \alpha w^t_{v,u}}{\alpha \mathcal{W}}. \tag{11}$$

Meanwhile, there exists a scaling factor α such that the numerator $\sum_{u \in N(v)} \alpha w^t_{v,u}$ is larger than $N(v)$ and holds for any t. Since $1/\alpha\mathcal{W}$ is a constant value, we can conclude that $Pr[v] = \Omega(N(v))$. Hence, the degree of vertices can be an admissible heuristic that estimates the probability of the corresponding vertex being traversed.

Degree-aware Cache Policy: The above analysis supports us in designing a cache replacement policy that replaces low-degree vertices in fast memory with high-degree vertices for a high cache hit ratio. Figure 5 shows an example of our cache replacement policy and the hardware architecture of the proposed degree-aware cache. The input to our degree-aware cache is a vertex index, and the corresponding output is a tuple that contains the starting address of the neighbors (adjacent edges) and the degree of the vertex. When an input vertex v_a is received in Step ⓐ, our degree-aware cache looks up whether v_a exists in the cache by comparing the tag of the cache line and the tag of the input. If v_a is already in the cache (cache hit), the corresponding cache line is read in Step ⓑ, and the output is directly sent to the Neighbor Loader module within one cycle (Step ⓒ). If v_a is not in the cache (cache miss), one memory access is issued to the off-chip DRAM in Step ⓓ. When the data for v_a is received from DRAM, the cache immediately outputs the returned data and

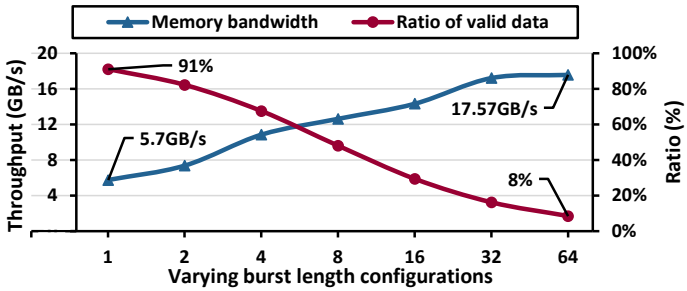


Fig. 6. Memory bandwidth and the ratio of valid data of MetaPath on livejournal with varying burst lengths.

compares the degree of the vertex in the corresponding cache line with the degree of v_a in Step ⓔ. If the degree of v_a is higher, the cache updates the cache line with the data for v_a; otherwise, it retains the original data.

5.2 Dynamic Burst Engine

GDRWs need to load all neighbors of the current vertex to move one step forward. Since neighbors of a vertex are stored in consecutive addresses, burst access that reads multiple data sequentially with a single memory request has been an effective method to improve memory bandwidth utilization [4, 12, 16]. The data size of each burst access, namely the burst size, is equal to the data width of the memory interface, N_{bus}, multiplied by the burst length, S. For example, if the data width of the memory interface is 512-bit, burst access with a burst length of four will request 2048-bit data per request. Figure 6 shows memory throughput benchmark results (blue line) with different burst lengths on our FPGA platform (Section 6.1). The results indicate that the utilized memory bandwidth increases significantly with increasing burst length until reaching the peak bandwidth (17.57 GB/s).

Issues of fixed burst length: Burst access with a long burst length is not effective for power-law graphs as vertices have varying numbers of neighbors. If the size of the neighbors is narrower than the burst size, the remaining data returned from memory will be unused, resulting in poor effective memory bandwidth utilization. We define *the ratio of valid data* as the percentage of data used in the computation compared to the total number of data loaded by the memory engine. The red line in Figure 6 shows the ratio of valid data of MetaPath on livejournal with an increasing burst length configuration. It turns out that the ratio of valid data is the highest with a burst length of one and decreases with larger burst lengths. The existing study, DynaBurst proposed by Asiatici and Ienne[4], adopts variable-length bursts to the memory interface to improve the effective memory bandwidth for irregular applications. The main idea is to cache and reorder numerous narrow memory requests in a non-blocking cache and explore data reuse of burst accesses. However, it requires a large amount of on-chip memory and logic resources for the cache and reordering function, respectively. Moreover, the latency is non-deterministic and varies significantly, which is not suitable for GDRWs.

Our Design: To utilize the high memory bandwidth of the long burst and maintain a high ratio of valid data of the short burst simultaneously, we propose a dynamic burst engine that schedules memory access requests with varying sizes to access pipelines with different burst length configurations at runtime. Let S_1 and S_2 be the number of bytes that can be loaded by one burst access of the long burst pipeline and short burst pipeline, respectively. With a total of c bytes of neighbors to be loaded, the number of long burst accesses is set to $\lfloor c/S_1 \rfloor$, while the number of short bursts is set to $\lceil (c - \lfloor c/S_1 \rfloor S_1)/S_2 \rceil$. In this way, the majority of requested data is handled by the long burst pipeline with high memory bandwidth utilization, and the remaining data that cannot form a long

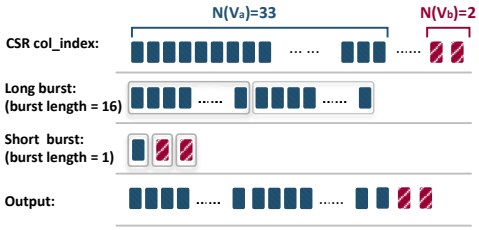


Fig. 7. Example of dynamic burst strategy.

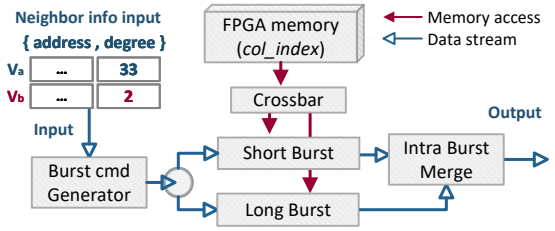


Fig. 8. Hardware architecture of dynamic burst engine.

burst is loaded by the short burst pipeline, achieving a high ratio of valid data. As the total bytes loaded is equal to $\lfloor c/S_1 \rfloor S_1 + \lceil (c - \lfloor c/S_1 \rfloor S_1)/S_2 \rceil S_2$, which is equal to $\lceil c/S_2 \rceil S_2$, the loaded unused data is no larger than S_2.

Due to the highly skewed distribution of node degrees in real graphs, it is very hard to build the model of burst length configurations to graphs with different structures. Hence, we determine the configurations of the burst length from practical evaluation (more details in Section 6.3.2). Figure 7 presents an example of our dynamic burst strategy given $S_1 = 16$ and $S_2 = 1$. Assuming the request to neighbors with a size of 33, the request is scheduled into two ($\lfloor 33/16 \rfloor$) long burst accesses and one ($(33 - 2 \times 16)/1$) short burst access. The request to neighbors with a size of two consists of zero ($\lfloor 2/16 \rfloor$) long burst and two ($(2 - 0)/1$) short burst accesses.

Figure 8 presents the hardware architecture of our dynamic burst engine, which contains four hardware modules: Burst cmd Generator, Long Burst, Short Burst, and Intra Burst Merge. The Burst cmd Generator module takes a data pair, *address*, *degree*, as input, where *address* is the starting address of neighbors in *col_index*, and *degree* is the number of neighbors of the vertex. The Burst cmd Generator follows the above scheduling method to generate burst access commands and dispatches commands to the Long Burst and Short Burst modules. Long Burst and Short Burst are connected to a memory crossbar and access the *col_index* array in FPGA's DRAM independently. Their burst length is configured to S_1 and S_2, respectively. The Intra Burst Merge module returns data from two burst modules and outputs them to the Weight Updater.

6 EVALUATION

In this section, we present the evaluation of LightRW. In Section 6.1, we introduce the hardware setup and the graph dataset used in our evaluation. In Section 6.2, we evaluate the throughput and efficiency of the proposed weighted reservoir sampling module, PWRS sampler. We further demonstrate the impact of parameters in the dynamic burst engine and degree-aware cache in Section 6.3. Finally, we compare LightRW with the state-of-the-art CPU-based implementation in Section 6.5.

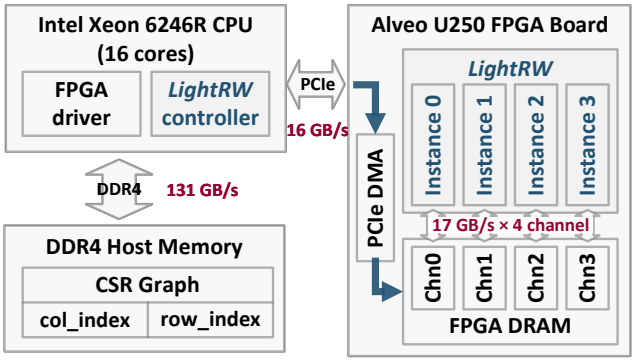


Fig. 9. Deployment of LightRW.

6.1 Experimental Setup

6.1.1 Hardware Platform. LightRW is built on a machine where an Xilinx Alveo U250 accelerator card is attached to the motherboard via the PCIe bus, as shown in Figure 9. The FPGA board has 2,000 BRAMs, 11,508 DSP slices, 1,341,000 LUTs, and four DRAM channels with a total capacity of 64 GB. We used the Vitis HLS Toolchain 2021.2 as the development environment.

6.1.2 Graph Datasets. Table 2 shows the graph datasets used in our evaluation. These real-world graphs come from different categories, including the web, citations, and social networks. The RMAT synthetic graphs are produced by the RMAT generator [10].

Table 2. The graph datasets.

Graphs	$\|V\|$	$\|E\|$	$\|D\|$	Type	Categories
youtube (*YT*) [35]	1.14M	2.99M	5	Undirected	Web
us-patents (*UP*) [35]	3.78M	16.52M	9	Directed	Citation
liveJournal (*LJ*) [35]	4.8M	68.9M	14	Undirected	Social
orkut (*OR*) [47]	3.1M	117.2M	38	Undirected	Social
uk2002 (*UK*) [9]	18.52M	298.11M	32	Directed	Social
rmat-12~22 (*RMAT*) [34]	2^{12}~2^{22}	2^{15}~2^{25}	8	Directed	Synthetic

6.1.3 Workloads. We implement and evaluate two representative GDRW algorithms: MetaPath [55] and Node2Vec [25]. These algorithms are widely adopted as benchmarks in existing graph random walk systems [19, 25, 42, 57].

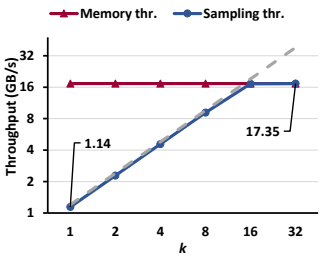
6.1.4 Parameter Settings. We follow the experimental settings of previous works [54, 65]. The number of queries is set to the number of vertices with non-zero degrees, and each query has a unique starting vertex. All queries are shuffled following the approach in ThunderRW [54]. The sampling method of ThunderRW is configured to use inverse transformation sampling following the authors' recommendation. The graph datasets are initialized with random edge weights and vertex labels. The query length is set to five for MetaPath and 80 for Node2Vec, and we set $p = 2$ and $q = 0.5$ for Node2Vec. All experiments are repeated five times, and the median results are presented.

6.1.5 Implementation Details. As shown in Figure 9, we implemented LightRW on a CPU-FPGA platform, where a Xilinx Alveo U250 FPGA accelerator card is attached to the motherboard through the PCIe bus. To run GDRW queries on a given graph, the LightRW controller on the host CPU side issues DMA requests to transfer random walk queries and graph data in the compressed sparse row (CSR) format to the DRAM of the FPGA platform and then invokes the accelerator. The LightRW controller then enters a waiting state until the accelerator completes the computation.

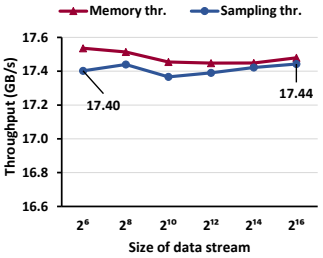
Modern FPGAs generally use multiple DRAM channels to provide higher memory bandwidth [14, 31]. To utilize multiple DRAM channels, we instantiate multiple independent LightRW instances and connect each of them to one DRAM channel, as shown in Figure 9. Each LightRW instance has a private and independent copy of the graph data and is configured to fully utilize the bandwidth of one channel. Meanwhile, we evenly distribute random walk queries to all instances.

6.2 Evaluation on WRS Sampler

The throughput of the WRS Sampler determines the overall performance of LightRW. Therefore, we conduct an evaluation of the WRS Sampler module with different degrees of parallelism and varying workloads individually. We randomly generate weights and items as the input data to be sampled by the WRS Sampler. The pre-generated data is stored in the one FPGA's DRAM. During



(a) Varying degree of parallelism.



(b) Varying length of stream.

Fig. 10. Throughput evaluation of WRS sampler.

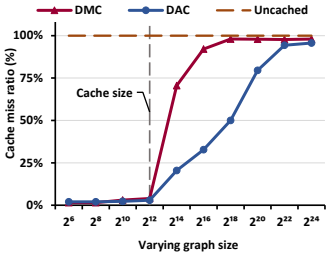


Fig. 11. Comparison of the cache miss ratio between DAC and DMC as the size of graphs increases.

the evaluation, the weights are sent to the WRS Sampler in a stream, and we measure the number of traversed items per second as throughput.

Figure 10a shows the throughput of the proposed WRS Sampler with varying degrees of parallelism, k (the number of vertices consumed per cycle). The blue line shows the measured throughput, while the gray dashed line shows the theoretical throughput. We observe that the throughput of the WRS Sampler increases linearly with k and matches the theoretical throughput when $k \leq 16$. At $k = 16$, the sampling throughput reaches the maximum bandwidth of the FPGA DRAM (17 GB/s), indicating that the proposed sampler can fully utilize the available memory bandwidth. Moreover, the results demonstrate the good scalability of our proposed parallel weighted reservoir algorithm and hardware architecture design.

Figure 10b shows the throughput of the WRS Sampler with varying workloads. Specifically, we generate items of different sizes to be sampled in data streams ranging from 2^6 to 2^{16}. To ensure the full bandwidth utilization of the FPGA's DRAM by the WRS Sampler, we set the degree of parallelism to 16. The results show that the throughput of the WRS Sampler is slightly less than the maximum memory throughput when the workload size is small. This is due to the initialization overhead of the pipeline execution, but such a performance difference is negligible. Overall, the proposed WRS Sampler is capable of fully utilizing the memory bandwidth with varying workloads. In other words, the WRS Sampler is not the bottleneck of the entire accelerator pipeline.

6.3 Impact of Memory Optimizations

In this section, we first evaluate the impact of different dynamic burst strategies in the dynamic burst engine. We then demonstrate the benefits of our degree-aware cache in reducing the overhead of random accesses in GDRWs.

6.3.1 Degree-aware Cache. Figure 11 depicts the cache miss ratio of the proposed degree-aware cache (DAC) compared to the direct-mapped cache (DMC) for MetaPath on RMAT graphs with an increasing number of vertices from 2^6 to 2^{24}. The evaluated caches are capable of storing up to 2^{12} vertices.

We can make the following observations. First, for graphs with fewer than 2^{12} vertices, the cache miss ratio is close to zero, as all vertices can be cached. Second, our degree-aware cache has a much lower cache miss ratio than a direct-mapped cache. Specifically, as the size of the graph increases, the cache miss ratio of the direct-mapped cache grows significantly, while our cache still maintains a comparatively lower miss ratio. For example, when the graph has 2^{18} vertices, the cache miss ratio of the direct-mapped cache is close to 100%, while our degree-aware cache only has a 49% cache miss.

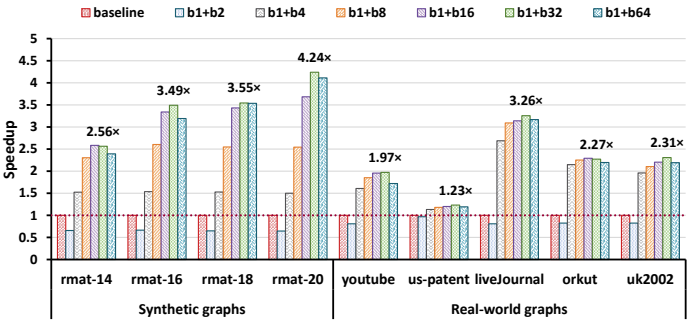


Fig. 12. Speedup of different dynamic burst strategies on MetaPath with synthetic graphs and real world graphs.

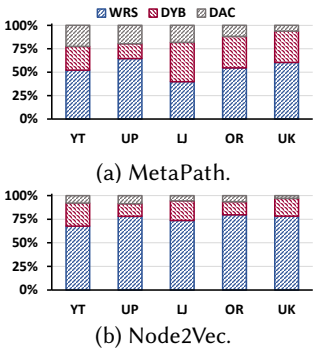
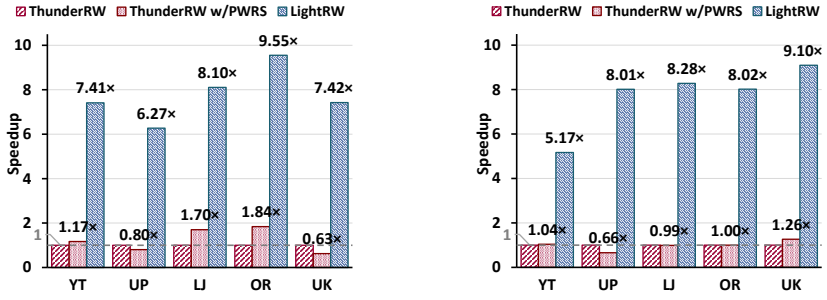


Fig. 13. Performance breakdown of WRS, DYB, and DAC on MetaPath and Node2Vec.

6.3.2 Dynamic Burst Engine. Figure 12 shows the throughput comparison on MetaPath with different dynamic burst strategies. The burst strategies are represented as 'b{x} + b{y}'. For example, given a strategy b1 + b16, it indicates that the length of a short burst and a long burst is set to 1 and 16, respectively. The baseline is b1 + b0, i.e., the burst length of the access pipeline is only configured to one, which is the common setting to handle irregular workloads [4]. We set the length of the short burst to one since it achieves minimal overhead in loading unused data, as we have discussed in Section 5.2. Based on the results in Figure 12, we have the following highlights. First, performance varies significantly with different strategies. The burst strategy b1 + b32 achieves the best performance and outperforms the baseline up to 4.24×. Second, the strategy b1 + b2 delivers the worst performance, because the benefits of long burst with length of two cannot amortize the overhead of the burst plan generation. Third, although similar trends are observed on both synthetic and real-world graphs, the performance improvements from dynamic burst strategies on real-world graphs are not as significant. This is because real-world graphs are generally more irregular than synthetic graphs. Nonetheless, our dynamic burst strategies can still provide up to 3.26× throughput improvement compared to the baseline solution. As the strategy b1 + b32 outperforms others in all graphs, we use it as the dynamic burst strategy for the subsequent evaluation.

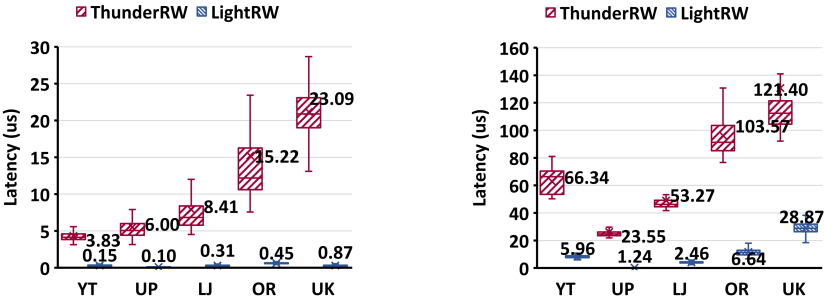
6.4 Performance Breakdown

Figure 13 analyzes the performance breakdown of the three proposed techniques, weighted reservoir sampling (WRS), dynamic burst engine (DYB), and degree-aware cache (DAC). Specifically, each proposed technique is disabled one at a time, and the resulting impact on performance is measured relative to the implementation with all techniques enabled. On the basis of the results, we have the following observations. First, WRS, which enables pipelined execution, contributes the most performance improvement (up to 41% to 79%) for the two GDRW algorithms, particularly for Node2Vec. Second, the benefit from the dynamic burst engine on Node2Vec is comparatively small. This is because Node2Vec requires additional memory access on the neighbors of the last traversed vertex, which in turn decreases the available memory bandwidth for the dynamic burst engine. Third, the degree-aware cache contributes more improvement to MetaPath than to Node2Vec. This is because that the degree-aware cache aims to improve the performance on accessing the *col_index* array, but Node2Vec has more memory accesses on the *row_index* array. Even though our cache only occupies several URAMs, it still achieves up to 6% improvement on the largest graph *uk2002*, demonstrating the effectiveness of the proposed cache replacement policy.



(a) MetaPath. (b) Node2Vec.

Fig. 14. Performance comparison between LightRW and ThunderRW on MetaPath and Node2Vec.



(a) MetaPath. (b) Node2Vec.

Fig. 15. Comparison of latency between LightRW and ThunderRW on MetaPath and Node2Vec.

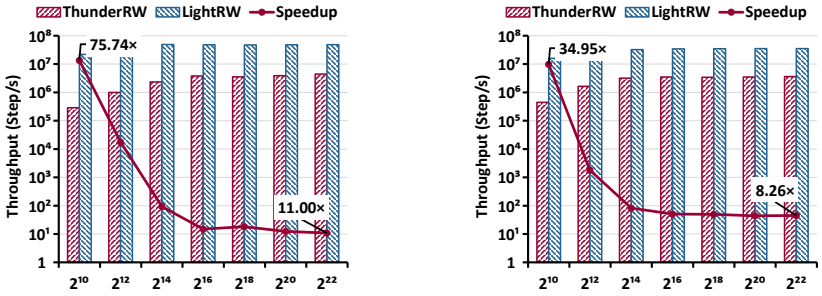
6.5 Comparison with the State-of-the-art CPU Implementation

We compare LightRW with the state-of-the-art CPU-based implementation, ThunderRW [54]. ThunderRW is executed on a server equipped with an Intel Xeon Gold 6246R CPU, which has 16 physical cores and a 19-stage pipeline, with a total cache capacity of 35.75MB [2].

First, we present the speedup of MetaPath and Node2Vec on five real-world graphs. Next, we compare the latency of executing small number of queries. Finally, we conduct a sensitivity evaluation by varying the query length and number of queries.

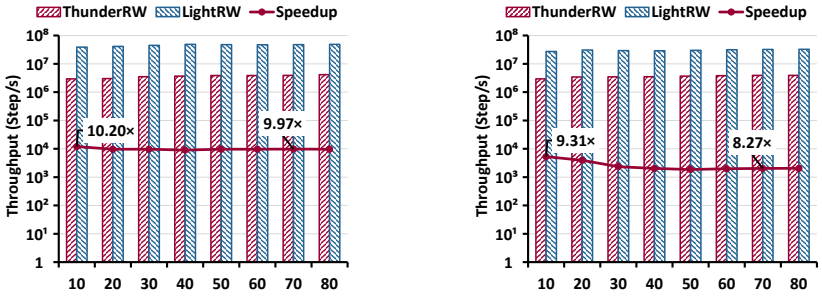
6.5.1 Throughput comparison. Figure 14 presents the speedup in performance of LightRW over ThunderRW for both MetaPath and Node2Vec. Besides the explicit memory prefetching, task level parallelism, and parallelized PRNG [50] that have already been enabled in ThunderRW, we further implement and integrate the proposed parallel weighted reservoir sampling algorithm into ThunderRW (ThunderRW w/PRWS) to make a fair comparison with LightRW.

The impact of parallel WRS algorithm on ThunderRW varies on different graphs. For example, 1.84× speedup is observed on *OR* dataset on MetaPath random walk, while significant performance degradation is also observed on the *UP* and *UK* datasets. There are two potential reasons. First, the maximum number of CPU pipeline stages is limited to 19, which is not suitable for the proposed parallel WRS algorithm that relies on long pipeline execution. Second, the computational cost of random number generation is not sufficiently amortized over the benefit of reduced memory accesses. LightRW outperforms ThunderRW, even running at a 10× slower frequency and 2× lower sequential memory bandwidth. In particular, LightRW delivers 6.27×~9.55× speedup over ThunderRW on MetaPath and 5.17×~9.10× speedup on Node2Vec. This is attributed to the proposed fine-grained pipelining with WRS and memory optimizations. It is worth noting that the speedup



(a) MetaPath. (b) Node2Vec.

Fig. 16. Comparison of throughput between ThunderRW and LightRW on *LJ* with varying numbers of queries on MetaPath and Node2Vec.



(a) MetaPath. (b) Node2Vec.

Fig. 17. Comparison of throughput between ThunderRW and LightRW on *LJ* with varying query lengths on MetaPath and Node2Vec.

on the *youtube* graph is smaller than others because it is small enough to fit into the CPU last-level cache.

6.5.2 Latency evaluation. Figure 15 shows the execution latency of LightRW and ThunderRW. Latency is calculated as the time from the start to the end of a query. For LightRW, we implement a cycle counter inside the accelerator to measure the exact hardware cycles. We measure the latency of 8192 randomly selected queries for each graph using both systems. In Figure 15, the rectangular box of each data series indicates the range from the lower quartile (25th percentile) to the upper quartile (75th percentile), and the horizontal line inside the box represents the median measurement. The whiskers with the horizontal lines at the top and bottom represent the maximum and minimum measurements, respectively.

On the whole, LightRW achieves much lower latency than ThunderRW. This suggests that FPGA-based GDRW is more suitable for handling real-time graph analytic applications [21, 32, 38]. Additionally, the latency of LightRW is more consistent across different graphs compared to ThunderRW. This is because the FPGA-based design has deterministic latency, CPU-based designs often suffer from uncontrolled latency and non-deterministic resource sharing, such as hardware interrupt preemption, user/kernel space context switching, and synchronization between multiple threads, etc.

6.5.3 Sensitivity evaluation on query parameters. To further investigate the performance of LightRW and ThunderRW, we conduct a sensitivity analysis on a representative dataset, the liveJournal (*LJ*) graph. This graph has a moderate number of edges and its average degree is the median of all evaluated real-world graphs. Specifically, we vary the number and length of queries in our analysis.

Table 3. The comparison of power efficiency between LightRW and ThunderRW on all tested graphs.

App.	Power consumption (Watt)		Power efficiency improvement
	LightRW	ThunderRW	
MetaPath	41~45	103~124	15.05×~26.42×
Node2Vec	39~42	110~126	16.28×~24.10×

Figure 16 presents the throughput of LightRW and ThunderRW with the number of queries ranging from 2^{10} to 2^{22} of random walk on the LJ graph. We start the number of queries from 2^{10} because ThunderRW requires at least 2^{10} queries to fully utilize the memory bandwidth of the target CPU. First, we can see that LightRW delivers almost constant throughput regardless of the number of queries. The throughput is up to 4.8×10^7 steps per second on MetaPath and 3.5×10^7 steps per second on Node2Vec. Next, we observe that LightRW's speedup over ThunderRW is 11.00×~75.74× on MetaPath and 8.26×~34.95× on Node2Vec. In particular, the speedup is more significant when the number of queries is relatively small (e.g., up to 75.54× with 2^{10} queries). This is because ThunderRW has constant initialization overhead, such as memory allocation and thread allocation, before the execution of multiple queries.

Figure 17 shows the throughput of LightRW and ThunderRW with varying query lengths from 10 to 80. The trends show that LightRW delivers constant throughput with different query lengths. The performance speedup of LightRW over ThunderRW is around 9.97×~10.20× for MetaPath, and 8.28×~9.31× for Node2Vec. Therefore, we conclude that LightRW stably outperforms ThunderRW on MetaPath and Node2Vec with different query settings.

6.5.4 Comparison of power efficiency. Table 3 compares the power consumption of LightRW and ThunderRW. The power consumption of CPU-based execution is measured using the CPU Energy Meter [41] during the execution of each benchmark, while the power consumption of our LightRW FPGA accelerator is measured using xbutil [63].

The power efficiency improvement is calculated as the ratio of end-to-end execution time per Watt of LightRW to that of ThunderRW. Overall, LightRW consumes less energy than ThunderRW and outperforms ThunderRW by 15×~26× on MetaPath and 16×~24× on Node2Vec in power efficiency. Since the power of LightRW on the FPGA platform is only 2.29× less than that of ThunderRW on CPUs, and LightRW runs to 9.55× fast ThunderRW, we conclude that the improvement in power efficiency is mainly due to our design rather than a power-efficient hardware platform. The significant performance and energy efficiency improvements also demonstrate the efficacy of customizing accelerators for GDRWs.

6.6 Other Results

In this section, we analyze the PCIe overhead of FPGA acceleration and present the hardware resource utilizations of LightRW for different GDRW algorithms

Table 4. The proportion of PCIe data transfer time over the end-to-end execution time of MetaPath and Node2Vec.

App.	youtube	us-patents	liveJournal	orkut	uk2002
MetaPath	16.5%	15.3%	20.5%	33.5%	23.3%
Node2vec	0.07%	1.10%	0.54%	0.56%	0.25%

6.6.1 PCIe Overhead Analysis. Table 4 shows the percentage of time taken for PCIe data transfer over the end-to-end execution time. The results demonstrate that graph data transfer generally occupies only a small portion of the total end-to-end execution time, ranging from 0.07% to 33.5%.

This suggests that data transfer is not a bottleneck for the entire system, and justifies the offloading of GDRWs to the FPGA accelerator.

Table 5. The consumption of hardware resources (percentage) and frequency (MHz) of MetaPath and Node2Vec on Alveo U250 FPGA borad.

App.	LUTs	REGs	BRAMs	DSPs	Frequency
MetaPath	33.52%	29.76%	17.24%	5.16%	300MHz
Node2Vec	20.84%	18.20%	36.12%	2.62%	300MHz

6.6.2 FPGA resource utilization. Table 5 presents the resource utilization and frequency of LightRW for MetaPath and Node2Vec. The results show that LightRW utilizes only a small portion of the resources available on the U250 FPGA, leaving ample resources for downstream processing logic, such as graph learning applications. Additionally, LightRW runs at a frequency of up to 300MHz benefiting from the modular design methodology used.

6.7 Case Study: Link Prediction

As a case study, we integrated our accelerator into the Stanford Network Analysis Platform (SNAP) to accelerate the link prediction application [62]. SNAP is a popular graph analysis platform with a core library written in C++, optimized for maximum performance and compact graph representation [36]. Link prediction is a widely used function in applications such as social networks, bioinformatics, and e-commerce [62]. The goal is to predict the existence of a link between two entities in a given network. A common approach is to use the Node2Vec random walk to generate different paths, followed by an unsupervised learning method, such as Word2Vec [25], to process the random walk paths and identify vertices with minimal cosine similarity to form the predicted edges.

Figure 18 presents the execution time breakdown of LightRW-accelerated SNAP (SNAP w/LightRW) and CPU-based SNAP for link prediction on the liveJournal graph. The results show that the Node2Vec random walk is the most time-consuming part of the link prediction application. With the acceleration of the Node2Vec random walk by LightRW, the total execution time of the link prediction is reduced to half the original execution time. Furthermore, although graph data and results need to be copied between CPU and FPGA memory, the impact is negligible compared to the overall execution time. The results also indicate the potential for further performance improvements if the learning process can also be offloaded to the FPGA side for acceleration.

7 RELATED WORKS

Graph Random Walks on CPU/GPUs. Knightking [65] adopts a vertex-centric model for a single query and processes multiple queries with the BSP execution model, i.e., waiting for all queries to complete the current step before starting the next step. To leverage the parallel processing capability of GPUs, C-SAW [43] parallelizes the initialization and generation stage using the inverse transformation sampling method among multiple GPU processing cores. Multiple queries are processed using the BSP model. ThunderRW [54] proposes a step-centric model that interleaves memory accesses of subtasks to improve the parallelism of multiple queries, achieving state-of-the-art performance and outperforming the GPU-based C-SAW.

Graph Random Walks on FPGAs. Su et al. [53] propose an FPGA-based accelerator for static random walks that utilizes *high bandwidth memory* (HBM) and multiple processing elements to handle multiple queries concurrently. They develope a degree-aware sampler that selects the appropriate division arithmetic units based on the degree of the vertices. This work is designed to support a specific MetaPath query by partitioning the graph into subgraphs based on vertex labels and performing a static random walk on each subgraph. However, their architecture is highly

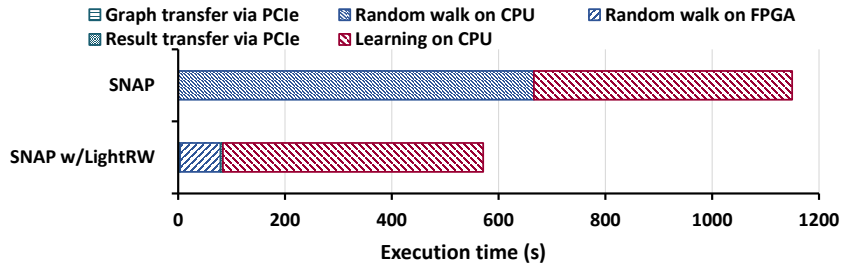


Fig. 18. Execution time breakdown for link prediction on *LJ* graph between SNAP and LightRW accelerated SNAP (SNAP w/LightRW)

specific to MetaPath queries with only two types of vertex labels. Because it only supports uniform random sampling, the proposed accelerator is limited to unweighted MetaPath random walks and cannot be generalized to any other GDRW algorithms. On the other hand, LightRW is an accelerator for dynamic random walks and supports not only generic MetaPath queries but also other GDRW algorithms. Our performance improvements are mainly due to the novel FPGA architecture design that efficiently realizes the proposed parallel weighted reservoir sampling algorithms.

8 CONCLUSION AND FUTURE WORK

This paper proposes LightRW, the first FPGA-based accelerator for GDRWs. Unlike existing CPU-based approaches that require synchronization barriers among different stages, LightRW parallelizes the weighted reservoir sampling for GDRWs to enable a fine-grained pipeline execution on the chip, allowing for better spatial parallelism and reduced memory access to the DRAM. Additionally, LightRW contains two novel memory optimizations to handle memory access patterns specific to random walks with better efficiency. Experimental results demonstrate that LightRW achieves up to 9.55× and 9.10× performance speedup compared to the state-of-the-art CPU-based implementation on two representative dynamic random walk algorithms. However, processing large graphs (e.g., in Terabyte scale) may require multiple FPGA boards with sufficient computation power and DRAM.

Future work: First, we plan to develop a distributed version of LightRW to leverage the increased availability of high-speed network interfaces (e.g., InfiniBand and 100G Ethernet) and open-source network frameworks on FPGAs (e.g., OpenNIC [64] and Corundum [22]). Second, LightRW demonstrates the advantage of a fine-grained pipelined accelerator over generic CPU hardware. Specifically, our pipelined architecture eliminates the high cost of the random number generation stage and the weighted random sampling initialization stage. This idea is not limited to GDRWs but further enables the possibility of more efficient computing for both graph-based and non-graph-based applications that rely on randomized approaches, such as Markov chain Monte Carlo [7], Bayesian networks [18], and physics simulations [45].

ACKNOWLEDGMENTS

This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002), the Ministry of Education AcRF Tier 2 grant (No. MOE-000242-00 / MOE-000242-01), and Google South & Southeast Asia Research Award 2022. We also thank the AMD Heterogeneous Accelerated Compute Clusters (HACC) program (formerly known as the XACC program - Xilinx Adaptive Compute Cluster program) [1] for the generous hardware donation. Bingsheng He and Yao Chen from the National University of Singapore are the corresponding authors.

REFERENCES

[1] AMD. 2023. Heterogeneous Accelerated Compute Clusters (HACC) Program. https://www.amd-haccs.io/index.html.

[2] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. 2019. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro* 39, 2 (2019), 29–36.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[4] Mikhail Asiatici and Paolo Ienne. 2019. DynaBurst: Dynamically Assemblying DRAM Bursts over a Multitude of Random Accesses. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 254–262. https://doi.org/10.1109/FPL.2019.00049

[5] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, and Abdelkrim Kamel Oudjida. 2018. Survey on hardware implementation of random number generators on FPGA: Theory and experimental analyses. *Computer Science Review* 27 (2018), 135–153.

[6] Vignesh Balaji and Brandon Lucia. 2019. Combining data duplication and graph reordering to accelerate parallel graph processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 133–144.

[7] Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2019. AcMC2 : Accelerating Markov Chain Monte Carlo Algorithms for Probabilistic Models. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 515–528. https://doi.org/10.1145/3297858.3304019

[8] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–386.

[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[11] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2019. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 67–73.

[12] Xinyu Chen, Feng Cheng, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2022. ThunderGP: Resource-Efficient Graph Processing Framework on FPGAs with HLS. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* (2022).

[13] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. Skew-Oblivious Data Routing for Data Intensive Applications on FPGAs with HLS. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 937–942.

[14] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 69–80.

[15] Wai-Ki Ching, Ximin Huang, Michael K Ng, and Tak-Kuen Siu. 2013. Higher-order markov chains. In *Markov Chains*. Springer, 141–176.

[16] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. 2021. Hbm connect: High-performance hls interconnect for fpga hbm. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 116–126.

[17] Intel Corporation. 2022. *Intel VTune Profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[18] Petros Dellaportas, Jonathan J Forster, and Ioannis Ntzoufras. 2002. On Bayesian model and variable selection using MCMC. *Statistics and Computing* 12, 1 (2002), 27–36.

[19] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[20] Pavlos S Efraimidis and Paul G Spirakis. 2006. Weighted random sampling with a reservoir. *Information processing letters* 97, 5 (2006), 181–185.

[21] James Fairbanks, David Ediger, Rob McColl, David A Bader, and Eric Gilbert. 2013. A statistical framework for streaming graph analysis. In *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013)*. IEEE, 341–347.

[22] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*.

[23] Marco Gori and Augusto Pucci. 2006. Research paper recommender systems: A random-walk based approach. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06)*. IEEE, 778–781.

[24] Martin Grohe. 2020. word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 1–16.

[25] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[26] Yu He, Yangqiu Song, Jianxin Li, Cheng Ji, Jian Peng, and Hao Peng. 2019. Hetespaceywalk: A heterogeneous spacey random walk for heterogeneous information network embedding. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 639–648.

[27] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating graph linear algebra on HBM-equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[28] Jung-Chang Huang and Tau Leng. 1999. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*. IEEE, 244–248.

[29] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Parallel Weighted Random Sampling. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[30] Rajesh Jayaram, Gokarna Sharma, Srikanta Tirthapura, and David P Woodruff. 2019. Weighted reservoir sampling from distributed streams. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 218–235.

[31] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. 2020. Do {OS} abstractions make sense on {FPGAs}?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 991–1010.

[32] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[33] Pierre L'Ecuyer and Richard Simard. 2007. TestU01: AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)* 33, 4 (2007), 1–40.

[34] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research* 11, 2 (2010).

[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[36] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[37] Yuan Li, Paul Chow, Jiang Jiang, Minxuan Zhang, and Shaojun Wei. 2013. Software/Hardware Parallel Long-Period Random Number Generation Framework Based on the WELL Method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 5 (2013), 1054–1059.

[38] Xi Liu, Ping-Chun Hsieh, Nick Duffield, Rui Chen, Muhe Xie, and Xidao Wen. 2019. Real-time streaming graph embedding through local actions. In *Companion Proceedings of The 2019 World Wide Web Conference*. 285–293.

[39] Linyuan Lü and Tao Zhou. 2011. Link prediction in complex networks: A survey. *Physica A: statistical mechanics and its applications* 390, 6 (2011), 1150–1170.

[40] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8, 1 (1998), 3–30.

[41] SoSy Lab LMU Munich. 2021. CPU Energy Meter. https://github.com/sosy-lab/cpu-energy-meter.

[42] Giannis Nikolentzos and Michalis Vazirgiannis. 2020. Random walk graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 16211–16222.

[43] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[44] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[45] SJ Plimpton, SG Moore, A Borner, AK Stagg, TP Koehler, JR Torczynski, and MA Gallis. 2019. Direct simulation Monte Carlo on petaflop supercomputers and beyond. *Physics of Fluids* 31, 8 (2019), 086101.

[46] Juan J Rodriguez-Andina, Maria J Moure, and Maria D Valdes. 2007. Features, design tools, and application domains of FPGAs. *IEEE Transactions on Industrial Electronics* 54, 4 (2007), 1810–1823.

[47] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.
[48] Reuven Y Rubinstein and Dirk P Kroese. 2016. *Simulation and the Monte Carlo method*. Vol. 10. John Wiley & Sons.
[49] Ilya Safro, Paul D Hovland, Jaewook Shin, and Michelle Mills Strout. 2009. Improving Random Walk Performance.. In *CSC*. 108–112.
[50] Mutsuo Saito and Makoto Matsumoto. 2008. SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer, 607–622.
[51] Frank Ludvig Spitzer. 1976. *Principles of random walk / Frank Spitzer* (2d ed. ed.). Springer-Verlag New York. xiii, 408 p. ; pages.
[52] Mario Stipčević and Çetin Kaya Koç. 2014. True random number generators. In *Open Problems in Mathematics and Computational Science*. Springer, 275–315.
[53] Chunyou Su, Hao Liang, Wei Zhang, Kun Zhao, Baole Ai, Wenting Shen, and Zeke Wang. 2021. Graph Sampling with Fast Random Walker on HBM-enabled FPGA Accelerators. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 211–218.
[54] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: an in-memory graph random walk engine. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1992–2005.
[55] Yizhou Sun and Jiawei Han. 2013. Mining heterogeneous information networks: a structural analysis approach. *Acm Sigkdd Explorations Newsletter* 14, 2 (2013), 20–28.
[56] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2021. *ThundeRiNG: Generating Multiple Independent Random Number Sequences on FPGAs*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/3447818.3461664
[57] Fatemeh Vahedian, Robin Burke, and Bamshad Mobasher. 2017. Weighted Random Walk Sampling for Multi-Relational Recommendation. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization* (Bratislava, Slovakia) *(UMAP '17)*. Association for Computing Machinery, New York, NY, USA, 230–237. https://doi.org/10.1145/3079628.3079685
[58] Fatemeh Vahedian, Robin D Burke, and Bamshad Mobasher. 2016. Weighted Random Walks for Meta-Path Expansion in Heterogeneous Networks.. In *RecSys Posters*.
[59] Guojia Wan, Bo Du, Shirui Pan, and Gholameza Haffari. 2020. Reinforcement learning based meta-path discovery in large-scale heterogeneous information networks. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 34. 6094–6101.
[60] Nian Wang, Min Zeng, Yiming Li, Fang-Xiang Wu, and Min Li. 2021. Essential Protein Prediction Based on node2vec and XGBoost. *Journal of Computational Biology* 28, 7 (2021), 687–700.
[61] Wikipedia contributors. 2022. Inverse transform sampling — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Inverse_transform_sampling&oldid=1115190568 [Online; accessed 16-October-2022].
[62] Wikipedia contributors. 2022. Link prediction — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Link_prediction&oldid=1100501520 [Online; accessed 14-October-2022].
[63] Xilinx. 2020. Vitis Unified Software Development Platform 2020.2 Documentation. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/index.html.
[64] Xilinx. 2020. *Xilinx OpenNIC Shell*. https://github.com/Xilinx/open-nic-shell
[65] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537.
[66] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. 2014. Personalized entity recommendation: A heterogeneous information network approach. In *Proceedings of the 7th ACM international conference on Web search and data mining*. 283–292.
[67] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. 2021. LCCG: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[68] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. 2019. HitGraph: High-throughput graph processing framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2249–2264.

Received July 2022; revised October 2022; accepted November 2022