RidgeWalker: Perfectly Pipelined Graph Random Walks on FPGAs

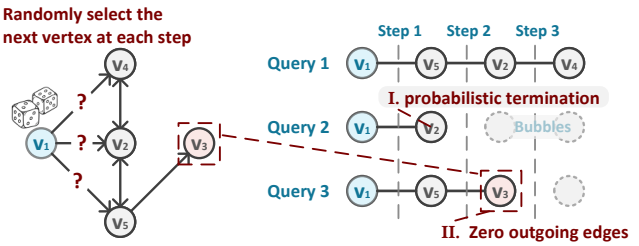
Hongshi Tan[†], Yao Chen[†], Xinyu Chen[‡], Qizhen Zhang[§], Cheng Chen[††], Weng-Fai Wong[†], Bingsheng He[†]
National University of Singapore[†], The Hong Kong University of Science and Technology (Guangzhou)[‡],
University of Toronto[§], ByteDance[††]
hongshi@u.nus.edu, yaochen@nus.edu.sg, xinyuchen@hkust-gz.edu.cn, qz@cs.toronto.edu,
chencheng.sg@bytedance.com, wongwf@comp.nus.edu.sg, hebs@comp.nus.edu.sg

***Abstract*—Graph Random Walks (GRWs) offer efficient approximations of key graph properties and have been widely adopted in many applications. However, GRW workloads are notoriously difficult to accelerate due to their strong data dependencies, irregular memory access patterns, and imbalanced execution behavior. While recent work explores FPGA-based accelerators for GRWs, existing solutions fall far short of hardware potential due to inefficient pipelining and static scheduling. This paper presents RidgeWalker, a high-performance GRW accelerator designed for datacenter FPGAs. The key insight behind RidgeWalker is that the Markov property of GRWs allows decomposition into stateless, fine-grained tasks that can be executed out-of-order without compromising correctness. Building on this insight, RidgeWalker introduces an asynchronous pipeline architecture with a feedback-driven scheduler grounded in queuing theory. This design enables perfect pipelining and adaptive load balancing. We prototype RidgeWalker on FPGAs and evaluate its performance across a range of GRW algorithms and real-world graph datasets. Experimental results demonstrate that RidgeWalker achieves an average speedup of $7.0\times$ over state-of-the-art FPGA solutions and $8.1\times$ over GPU solutions, with peak speedups of up to $71.0\times$ and $22.9\times$, respectively. The source code is publicly available at *https://github.com/Xtra-Computing/RidgeWalker*.**

I. Introduction

Modern data science and network analysis applications increasingly rely on *graph random walks* (GRWs) to efficiently explore and characterize complex network structures. GRW algorithms, based on Markov processes [1], traverse a graph by probabilistically selecting neighbors based solely on the current vertex, as shown in Figure 1a. This sampling strategy approximates key graph characteristics, such as centrality, influence propagation, and community structure, without exhaustive traversal [2]–[5]. Therefore, GRWs have been used as the core building blocks in databases [6], [7], machine learning [8]–[11], and graph-based reasoning [12]–[14].

GRWs have become the dominant performance bottleneck in many graph-learning workloads, often accounting for over 50% of total execution time [15]. This has spurred extensive research efforts aimed at developing high-performance GRW solutions on CPUs and GPUs [15]–[20]. However, general-purpose architectures like GPUs, designed for workloads with massive regular parallelism, are ill-suited for GRWs, which feature irregular computation and highly random memory access patterns [21]. As a result, even state-of-the-art GPU-based GRW [15] utilizes only 0.9% of total random access memory bandwidth (see Section VIII-C), exposing a fundamental mismatch between GRW workloads and conventional hardware.



(a) GRW query traversal on graphs. (b) Concurrent GRW query execution.

Fig. 1: Illustration of the inherent randomness and imbalanced traversal in GRWs.

This mismatch highlights the need for domain-specific architectures tailored to the unique characteristics of GRWs for improved bandwidth efficiency and performance. With architectural reconfigurability and access to *high-bandwidth memory* (HBM), modern FPGAs have attracted growing interest [18], [22], [23] for building customized GRW accelerators and adapting to rapidly evolving GRW applications. However, despite this potential, existing FPGA-based GRW accelerators still fall short of fully utilizing the hardware's capabilities. This performance gap stems from two inherent characteristics in GRW workloads that make high-performance accelerator design particularly difficult.

First, **GRWs exhibit strong data dependencies and unpredictable random memory accesses.** Each step of a walk requires a random access to retrieve the neighbors of the current vertex, and the next step cannot proceed until this retrieval is complete (shown in Figure 1a). Like traditional graph processing accelerators [24], [25], existing GRW design [22] caches frequently accessed vertices on-chip based on their access frequency. However, such strategies are largely ineffective for GRWs due to their probabilistic neighbor selection. As illustrated in Figure 1a, even walks starting from the same vertex v_1 rapidly diverge along distinct paths, offering minimal locality and thus limited cache reuse. Our profiling of FastRW [22] shows that once the graph exceeds on-chip cache capacity, random access memory bandwidth utilization drops by up to 42%.

Second, **concurrent GRW query execution introduces significant workload imbalance.** The lengths of each query vary greatly due to probabilistic termination and the structural sparsity of real-world graphs. For instance, vertex v_3 in Figure 1b has no outgoing edges, forcing any walk that reaches it to terminate immediately. Such behavior leads to unpredictable execution times across queries. Existing GRW architectures [18], [22], [23] rely on static scheduling and fail to adapt to runtime imbalances, leading to pipeline bubbles and under-utilization of the compute pipeline. Our evaluation shows that under such an imbalance, the effective random access memory bandwidth utilization can drop below 2.3% of the hardware's theoretical peak (see Section III).

We attribute the performance gap of existing designs to the lack of *perfectly pipelined* execution [26], which requires (a) proactively resolving data dependencies and (b) eliminating pipeline bubbles. Our key insight is that the Markovian nature of GRWs confines inter-step data dependencies to the current vertex, allowing queries to be decomposed into stateless, minimal tasks for fine-grained execution without compromising statistical correctness. This property motivates us to design a GRW accelerator architecture that supports out-of-order, asynchronous query execution to hide memory latency and adapts dynamically to workload imbalance.

In this paper, we present RidgeWalker, an efficient FPGA-based GRW accelerator that leverages the Markov property of GRWs to enable fine-grained task decomposition and achieve perfectly pipelined execution. First, RidgeWalker enables out-of-order GRW query execution through an asynchronous pipeline architecture. By interleaving tasks from different queries and executing them independently, queries are dynamically scheduled and redistributed across pipelines at runtime rather than being constrained by sequential input order or affinity to the processing pipeline. This flexible execution model effectively hides the latency of random memory accesses. Second, RidgeWalker incorporates a feedback-driven scheduler that adaptively resolves workload imbalance and pipeline bubbles. Grounded in *queuing theory*, it provides formal guarantees of zero pipeline bubbles, sustaining continuous data flow even under highly imbalanced GRW workloads. To summarize, this paper makes the following contributions:

- We propose RidgeWalker, the first GRW accelerator to achieve perfectly pipelined execution, aligning architectural design with the Markovian structure of GRW algorithms.
- We introduce an asynchronous GRW accelerator architecture that efficiently pipelines decomposed stateless GRW tasks, achieving fine-grained concurrency and near-optimal random-access bandwidth utilization.
- We introduce a feedback-driven scheduler based on queuing theory, offering formal guarantees for zero pipeline bubbles and sustained high utilization under irregular workloads and high parallelism.
- We implement RidgeWalker on various FPGAs and demonstrate its generality and performance across multiple GRW algorithms and datasets, achieving up to 71.0× and 22.9× speedup over prior FPGA and GPU solutions, respectively.

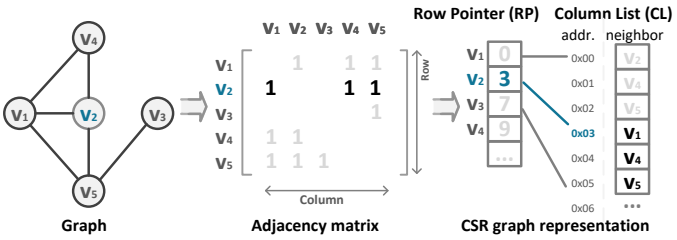


Fig. 2: An example of graph representation in CSR format.

Algorithm II.1: General GRW algorithm

Input: $\mathcal{G}$: a given graph, $\mathbb{Q}$: a set of input queries.
Output: *res*: paths of traversed vertices.

1 $res = \emptyset$;
2 **foreach** $Q \in \mathbb{Q}$ **do**
3 $v_{curr} = Q.v_{start}$, $path = \emptyset$;
4 **loop**
  /* Access the row pointer of CSR; */
5 $\{addr, deg\} =$ **row_access**$(v_{curr}, \mathcal{G})$;
  /* Application-specific sampling; */
6 $index =$ **sampling**$(addr, deg, \mathcal{G})$;
  /* Access the column list of CSR; */
7 $v_{curr} =$ **column_access**$(addr, index, \mathcal{G})$;
8 $path$.**push**(v_{curr});
9 **if** $(Q$.**is_end**$())$ **then**
10 res.**push**$(path)$;
11 **break**;
12 **return** *res*;

II. Background

A. CSR Graph Representation

Figure 2 illustrates an example graph encoded in the *compressed sparse row* (CSR) format, the adjacency representation most commonly used in GRW workloads. CSR uses two arrays: the row pointer array (RP) and the column list array (CL). Each entry RP$[i]$ specifies the starting offset of vertex v_i's neighbor list in CL, which stores the column indices of non-zero entries in the adjacency matrix. For example, RP$[2] = 3$ indicates that the neighbors of v_2 begin at CL address `0x03`. Storing every vertex's neighbors contiguously allows an $O(1)$ index lookup, making it well-suited and widely adopted for the random sampling in GRWs [15]–[20].

B. GRW Execution Flow

Algorithm II.1 outlines the execution flow of a general GRW algorithm. The input is a CSR graph $\mathcal{G}$, and the application launches a set of random walk queries $\mathbb{Q}$ with designated starting vertices. A query begins at $v_{curr} = Q.v_{start}$ and advances one step at a time. For each step, the algorithm first reads the degree of v_{curr} and the pointer to its neighbor list from the row pointer array of the CSR graph (Line 5). An application-specific sampling function then chooses a neighbor index within that list (Line 6). Using the selected index, the algorithm accesses the column list to fetch the next vertex to

visit. The walk terminates when it either reaches the maximum length or encounters a vertex with no outgoing edges.

GRWs are naturally difficult to perfectly pipeline, because their execution cannot be structured as a perfectly nested loop [27], since the number of inner loop iterations is not known in advance. This unpredictability prevents the straightforward adoption of conventional pipelining techniques that rely on static scheduling, such as Finite State Machine with Datapath (FSMD) [28], SDC-based modulo scheduling [29], and similar approaches, from achieving perfect pipelining and optimal bandwidth utilization.

III. Motivation and Design Principles

A. The Need for Specialized Accelerators for GRWs

While extensive research has focused on accelerators for general graph processing [24], [25], [30]–[38], these designs are fundamentally mismatched with the needs of GRWs. General graph processing typically involves scanning all neighbors of a vertex to perform aggregation or message propagation. This has motivated optimizations like update coalescing (e.g., ACTS [30], Swift [32]) and sequential buffering (e.g., GraphPulse [35]) that efficiently utilize on-chip memory. However, GRWs are structurally different, as they traverse the graph by sampling a single, randomly selected neighbor at each step, without aggregation. This sparsity in access renders such buffering and aggregation techniques ineffective. As a result, there is a pressing need for specialized architectures that can match the stochastic, data-dependent behavior of GRWs.

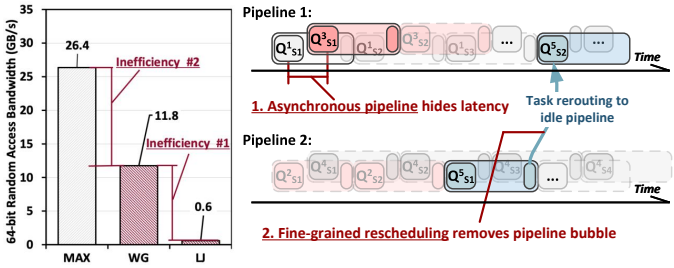
B. Inefficiencies in Existing GRW Accelerators

While several prior works have explored FPGA-based GRW acceleration [18], [22], [23], current designs fall far short of the hardware's potential. To explore design challenges of GRW accelerators, we conduct in-depth performance analysis for FastRW [22] and LightRW [18]. We measure its bandwidth utilization as $B_{\text{measured}}/B_{\text{peak}}$, where B_{measured} is the effective bandwidth observed during GRW traversal and is calculated by dividing the total memory footprint of traversed edges by the overall execution time. B_{peak} denotes the theoretical peak bandwidth provided by DRAM or HBM, estimated following Asifuzzaman *et al.* [39]. Since each GRW step typically triggers a DRAM row-buffer miss, we consider this to compute peak 64-bit random-access bandwidth B_{peak}, aligned with the granularity of vertex-level random accesses in GRWs:

$$B_{\text{peak}} = \frac{f_{\text{mem}}}{t_{\text{RRD}}} \times N_{\text{chn}} \times \frac{\text{64-bit}}{8}, \tag{1}$$

where f_{mem} is the memory operating frequency, t_{RRD} is the row-to-row delay, and N_{chn} denotes the number of available memory channels. We conclude two key observations:

Observation #1: Memory access latency is inherently difficult to hide due to GRW data dependency. Figure 3a shows that FastRW [22] sustains 11.8 GB/s of effective bandwidth on the small `WG` graph, where the row-pointer array fits entirely in on-chip SRAM. However, performance drops drastically to 0.6 GB/s, merely 2.3% of peak, on the larger `LJ` graph.



(a) Bottom-up analysis. **(b) Proposed perfectly pipelined solution.**

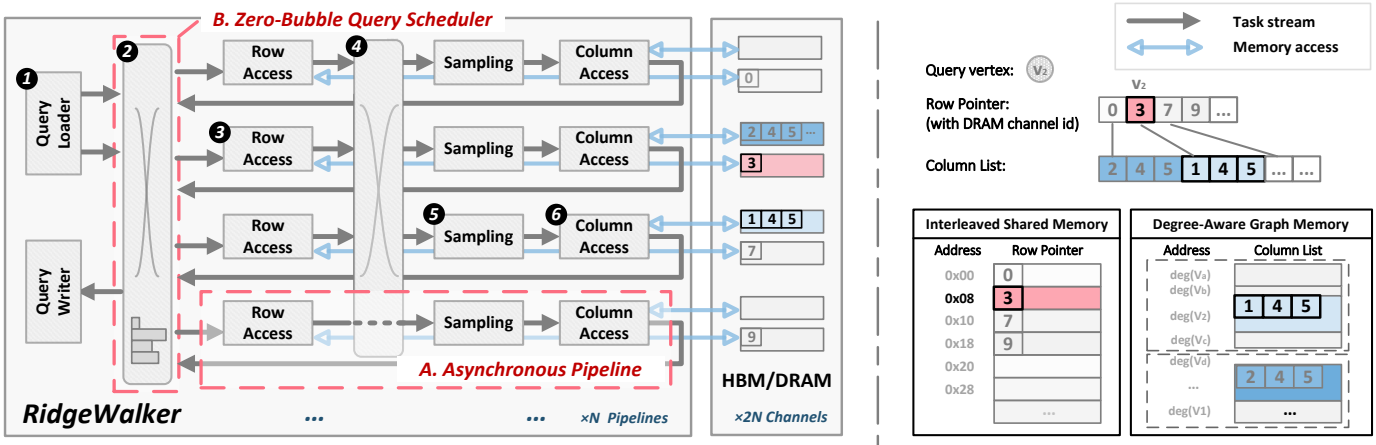
Fig. 3: (a) Bandwidth analysis of SOTA FPGA accelerator FastRW [22], highlighting underutilization. (b) Perfectly pipelined parallel GRW execution on two pipelines, Q^x_{sy} denotes the y-th step of traversal for query x.

This drop reveals a fundamental challenge: GRWs impose strict, step-by-step data dependencies, where each step must complete a random memory access to determine the next vertex before proceeding. These accesses are not only random but also sequentially chained, making it extremely difficult to prefetch or parallelize across steps. *Thus, optimizing the memory hierarchy alone is insufficient; instead, resolving GRW's inter-step dependencies is essential for hiding latency and achieving high performance.*

Observation #2: Static scheduling cannot help the workload imbalance problem in GRW. Figure 3a compares FastRW's observed bandwidth with the theoretical peak `MAX` derived in Equation (1). Even when the row-pointer array fits entirely within the on-chip RAM, FastRW reaches only 45% of the peak. The shortfall is not a memory issue but query scheduling. Individual queries finish at different times, so idle cycles accumulate while the fixed schedule waits for the slowest query. LightRW [18] improves locality by batching queries in a ring buffer, yet still issues every step in a predetermined order; when a walk terminates early, its reserved slots remain empty. Our analysis on LightRW reveals bubble ratios up to 37%, confirming that static scheduling cannot cope with GRW's highly imbalanced runtime behavior. *To close this gap, a scheduler must adapt on the fly, redirecting ready tasks to a free processing pipeline, to keep all hardware resources busy.*

C. RidgeWalker: Towards Perfectly Pipelined GRWs

Our analysis shows that efficient GRW acceleration requires rethinking both the algorithm and architecture to break data dependencies and dynamically balance workloads. Our key insight builds on the Markov property of GRWs, which asserts that the next step in a random walk depends solely on the current vertex, independent of prior history. *This property allows us to decompose each GRW query into stateless, fine-grained tasks involving only localized memory access and computation.* These tasks are inherently independent and can be executed without maintaining global walk state or requiring inter-task synchronization, allowing the architecture to issue massive outstanding memory requests and balance workloads at a much finer granularity. Based on this insight, we pro-



(a) RidgeWalker architecture (b) Data layout

Fig. 4: RidgeWalker architecture overview.

pose RidgeWalker, a GRW accelerator that achieves perfectly pipelined execution through two co-design innovations:

Out-of-order Query Execution. Each GRW traversal hop is further decomposed into several independent, stateless tasks, such as memory access and sampling, that can be issued as soon as their required vertex data becomes available. Tasks from different queries interleave freely, allowing memory stalls in one query to be masked by ready tasks from others (e.g., Q^1_{s1}, Q^1_{s2}, and Q^3_{s1} share Pipeline 1 in Figure 3b), allowing for fully overlapping memory latency with computation.

Workload-Adaptive Scheduling. To eliminate pipeline bubbles in concurrent GRW query execution, RidgeWalker continuously monitors pipeline utilization at cycle-level granularity and dynamically redirects ready tasks to idle modules through a feedback-driven scheduling. For example, in Figure 3b, when Q^5_{s1} completes in Pipeline 2 and Pipeline 1 becomes idle, the scheduler immediately dispatches Q^5_{s2} to fill the gap. Such adaptive scheduling eliminates pipeline bubbles and sustains peak throughput even under highly imbalanced GRWs.

IV. RidgeWalker Overview

A. Architecture Overview

Figure 4 illustrates the architecture of RidgeWalker, a novel GRW accelerator with perfectly pipelined execution. RidgeWalker adopts two tightly integrated components:

A) Asynchronous Pipeline (§V): To support out-of-order GRW query execution, RidgeWalker introduces an asynchronous pipeline design within a scalable architecture. Each asynchronous pipeline independently processes the stages of a GRW step using three key modules: *Row Access*, *Sampling*, and *Column Access*. The *Row Access* and *Column Access* modules are assigned with independent HBM channels, which avoids inter-channel arbitration and contention, ensuring that random accesses issued by one stage do not interfere with those from another.

Within each pipeline, RidgeWalker integrates an asynchronous memory-access engine (see Section V-B) that provides up to 128 outstanding, non-blocking requests. The engine is fully pipelined with an initiation interval of one cycle, enabling it to saturate the outstanding-request capacity of the HBM controller and align memory throughput with the pipeline's processing rate. The number of HBM channels assigned to each access engine is selected to match the random-access bandwidth demand of the asynchronous pipeline. For example, a single HBM2 channel can sustain roughly 284 million 64-bit random transactions per second. A memory-access engine, operating at approximately 300 MHz, is therefore well matched to the capacity of one HBM channel and is able to fully saturate its available random-access bandwidth.

B) Zero-Bubble Query Scheduler (§VI): RidgeWalker instantiates multiple asynchronous pipelines to fully utilize the available random-access memory bandwidth when executing parallel GRW queries. To support this parallelism at the data level, the CSR graph is randomly partitioned and distributed across all HBM channels. Unlike BFS or PageRank, where traversal is tightly tied to partition boundaries and often causes severe load imbalance, GRWs spread their accesses much more evenly. Prior analyses on GRW mixing and meeting times [40], [41] show that any imbalance lasts only for a short window of steps, after which access probabilities across partitions become nearly uniform (typically in a few tens of steps for million-scale graphs).

To handle these short-lived fluctuations, RidgeWalker allows each walker to be flexibly reassigned at every hop. Multiple parallel pipelines operate asynchronously, and the *Task Router*, implemented with a butterfly interconnect, directs each task to the correct memory channel based on the vertex it needs to access. The zero-bubble scheduler continuously monitors pipeline status and immediately fills open slots with ready tasks, ensuring that temporary bursts do not reduce throughput. Together, the flexible rescheduling and high-throughput routing allow RidgeWalker to sustain line-rate execution even under fully random GRW access patterns.

B. Query Execution Flow

Figure 4a illustrates how a GRW query executes a single step in RidgeWalker, with the corresponding graph memory layout shown in Figure 4b. To match GRW access patterns, the CSR graph is mapped and distributed across HBM channels: the row pointer array is partitioned and assigned to the Row Access channels, while the neighbor lists are shuffled across the Column Access channels in a round-robin manner to reduce the potential of bank conflicts. Each row pointer entry encodes both the target channel ID and the starting address of the corresponding neighbor list, facilitating direct access.

For a query starting from vertex V_2, the *Query Loader* (❶) fetches it from host memory; the *Scheduler* (❷) assigns it to Pipeline 2; the *Row Access* module (❸) retrieves V_2's row pointer. If the neighbor list resides in a different channel, the *Task Router* (❹) redirects the task accordingly. The *Sampling* module (❺) selects a neighbor based on a configurable distribution, and the *Column Access* module (❻) fetches the neighbor and determines the termination of the query. Each module communicates via shallow FIFOs within the AXI-Stream protocol, enabling backpressure-based flow control. The final list of traversed vertices is collected using the query index upon query termination. This process operates in a streaming-window manner. When the accumulated path length reaches the predefined write granularity, the corresponding paths are sequentially written back to global or host memory that can be used for downstream applications without interrupting GRW execution.

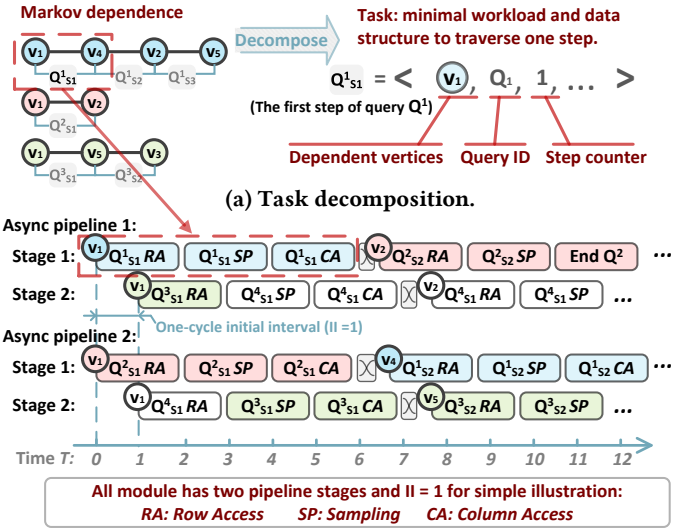
V. Out-of-order Concurrent Query Execution

This section describes how the proposed asynchronous pipeline enables out-of-order query execution and resolves data dependencies, and how the asynchronous memory access engine amortizes memory latency to maximize bandwidth utilization under GRW's highly random access patterns.

A. Decomposing Queries to Stateless Tasks

Unlike conventional dataflow architectures [18], [22], [42], where execution is strictly governed by data availability, and each module maintains local state to enforce sequential control, RidgeWalker eliminates this constraint through Markov-based task decomposition, followed by stateless, out-of-order pipeline execution. Each decomposed task flows independently through the pipeline without requiring additional control logic or global synchronization. Tasks are tagged with a unique query index for result tracking, allowing the system to correctly associate sampled vertices with their corresponding queries. Each module executes based solely on the availability of input tasks and is fully decoupled from the state of other modules. As a result, task execution and rescheduling are non-blocking and interference-free, enabling fully out-of-order query processing.

Figure 5a illustrates how RidgeWalker decomposes a GRW query into minimal, independent tasks. Each task is represented as a tuple $Q^y_{sx} = \langle v_{\text{last}}, \text{ID}_y, x, \dots \rangle$, where v_{last} denotes



(b) Out-of-order query execution.

Fig. 5: Markov-based task decomposition and illustration of out-of-order query execution across two pipelines.

the most recently visited vertex (or two vertices for higher-order walks like Node2Vec), ID_y uniquely identifies the query, and x tracks the current hop count. This tuple representation fits within a single pipeline word. Each module consumes the word in one cycle, performs its computation, updates the word, and forwards it to the next module. All modules are designed to process one task per cycle, sustaining fully pipelined throughput.

Figure 5b demonstrates out-of-order execution of decomposed steps across multiple pipelines. Each pipeline comprises a Row-Access (RA), Sampling (SP), and Column-Access (CA) module, and each module is simplified with two pipeline stages and an initiation interval of one cycle for illustration. Benefiting from task independence, they can be dynamically routed across pipelines based on runtime availability, enabling flexible and efficient utilization of all processing resources. For instance, at $T = 6$, Pipeline 1 has already completed Q^1_{s1}, and its successor step Q^1_{s2} is immediately dispatched to Pipeline 2, eliminating inter-pipeline blocking. Similarly, the first hop of Q^3 begins in Pipeline 1, but subsequent steps are redirected to Pipeline 2 to access graph data mapped to its associated memory. By fully decoupling pipeline modules and removing centralized control via an out-of-order execution model, RidgeWalker seamlessly overlaps module-level access with computation.

B. Asynchronous Memory Access for Concurrent Execution

The Asynchronous Access Engine is at the core microarchitectural component underpinning both the *Row Access* and *Column Access* modules. Figure 6 presents the architecture. An incoming task first enters the *Request Proxy*, which extracts the target address and accompanying metadata (e.g., vertex index and query ID). The vertex index is translated into a physical address and forwarded to the *Memory Engine*, which issues DRAM accesses at the memory's minimum granularity.

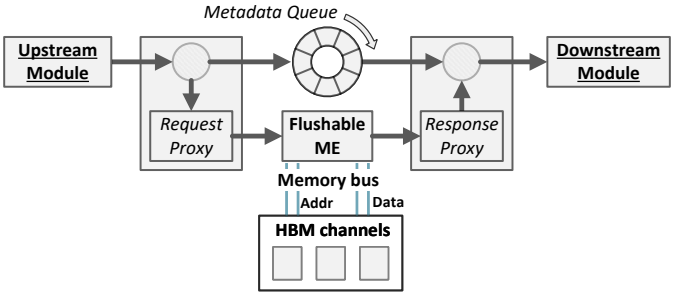


Fig. 6: Architecture of asynchronous memory access engine.

Metadata bypasses the data path and is enqueued separately to be reunited with the returned data.

Conventional hardware pipelines stall when input data is unavailable, causing serialization and poor latency hiding. To avoid this, the memory engine operates independently of input readiness signals, allowing access requests to proceed and flush without delay. Each memory request is issued via the AXI protocol and tagged with a unique transaction ID. The associated metadata are enqueued into a BRAM-based *Metadata Queue*, sized to cover the round-trip latency (up to 512 entries, sufficient for 100 cycles at 320 MHz). Since the AXI protocol ensures in-order responses per transaction ID, our memory engine includes an on-chip buffer supporting up to 64 transaction IDs to reconstruct out-of-order returns. The *Response Proxy* then reassembles the data and metadata into a complete task for downstream modules. This design eliminates pipeline stalls, fully overlaps memory latency, and supports high-throughput, fine-grained random accesses critical for GRWs.

C. Dynamic Query Reassignment Over Parallel Pipelines

To effectively scale GRW execution across a large number of parallel pipelines and memory channels, RidgeWalker must support dynamic per-hop reassignment of queries. The reassignment flexibility in RidgeWalker architecture arises from two orthogonal dimensions.

First, our Markov-based, per-hop task decomposition ensures that each GRW hop is represented as an independent, stateless task. Because no historical state or dependency chain is carried across hops, every decomposed task can be executed by any pipeline without violating correctness. This provides the foundation for fully flexible, per-hop redistribution. Second, the hardware overhead of dynamic task routing on FPGA fabrics is minimal. Each decomposed task is compact, no larger than 512 bits, and can be transferred in a single cycle through an AXI-Stream interface. These interfaces require only lightweight resources; even the handshaking FIFO used for flow control can be implemented with a single CLB (32 entries), which already suffices to sustain pipelining among modules. This compact and self-contained task format makes it practical to route and reassign work across many parallel pipelines using only lightweight on-chip interconnects and simple scheduling logic.

Given this flexibility, the key design goal is to maintain balanced execution by considering two factors: (1) each task must be routed to the correct memory channel that stores the adjacency list of its current vertex, and (2) no pipeline should become idle, thereby maximizing throughput. To achieve this, RidgeWalker integrates two architectural mechanisms that leverage task independence. The *Task Router*, implemented as a butterfly interconnect, performs data-aware routing by directing each task to the memory channel holding the required adjacency list. In parallel, the *Zero-Bubble Scheduler* continuously tracks pipeline availability and immediately issues ready tasks into newly freed pipeline slots. Because tasks contain no mutable state, they can be reassigned every cycle without rollback, coordination, or synchronization. As a result, our architecture seamlessly absorbs short-lived workload variations and prevents the formation of bubbles.

Through this combination of stateless task decomposition, data-aware routing, and runtime task dispatch, RidgeWalker sustains high utilization across all pipelines and memory channels, while providing the flexibility required for scalable, perfectly pipelined GRW execution.

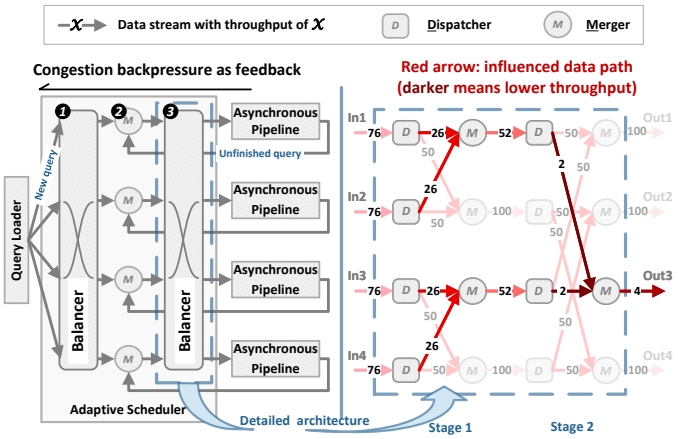
VI. Zero-Bubble Scheduling for Parallel GRW

Figure 7 illustrates the architecture of the *Zero-Bubble Scheduler*, which dynamically dispatches GRW tasks to pipelines based on real-time execution status. Our design is grounded in a queuing-theoretic model that guarantees full utilization under asynchronous execution, ensuring that tasks are executed without introducing pipeline bubbles.

A. Problem Formulation

Hardware View: As shown in Figure 7, GRW execution is organized as N *asynchronous* pipelines connected through stream FIFOs and an N-to-N balancer. Each GRW query is decomposed into a sequence of stateless tasks, but the *per-query execution time is data dependent*: the number of inner-loop iterations (neighbor traversals, rejection retries, early termination, etc.) is unknown a priori. Consequently, different pipelines complete queries at different times, and back-pressure propagates through FIFOs and the balancer. The central challenge is therefore a *hardware scheduling* problem: in every cycle, the scheduler must decide which pipelines to feed using only real-time availability signals (empty/full), such that: i) no pipeline is left idle when work exists (*zero bubbles*), and ii) no subset of pipelines is systematically favored under congestion (*fairness*).

Queuing Model: We model the scheduler as a bulk-service queue, specifically an $M/M/1[N]$ system [43], to reflect the hardware constraint that the scheduler can issue to multiple pipelines concurrently. Without loss of generality, we assume that the incoming tasks arrive according to a Poisson process with rate λ, capturing the stochastic injection of GRW tasks from upstream modules. Task service times are modeled as exponential with rate μ, reflecting the randomized, graph-structure-dependent completion time of GRW work units. The single "server" corresponds to the scheduler/balancer logic, which can dispatch up to N tasks in one decision epoch



(a) Zero-bubble scheduler. **(b) *N*-to-*N* balancer.**

Fig. 7: Zero-bubble query scheduler architecture and example of balancing workloads over $N = 4$ pipelines.

(i.e., one cycle), matching the N parallel pipelines; thus, the maximum batch size in our queuing model is N.

Back-pressure and Observation Delay: In hardware, the scheduler does not observe instantaneous pipeline idleness; it observes the system through FIFO occupancy and back-pressure signals that are delayed by the interconnect and pipeline depth. We capture this effect with a C-cycle delayed observation: at cycle t, scheduling decisions are made based on availability signals reflecting the workload state at $t - C$. This delayed feedback, together with stochastic service times, is what creates bubbles in naive static schedules. Our goal is to design a hardware-realizable policy that remains stable (i.e., does not build unbounded queues) for high load and, whenever tasks are present, sustains full utilization of the N pipelines despite the C-cycle delay.

B. Buffering Requirement for Zero-Bubble Scheduling

Given the hardware execution model in Figure 7, the scheduler observes pipeline availability through FIFO back-pressure with a finite feedback delay (up to C cycles). Under this delayed observation, insufficient buffering can transiently starve ready pipelines even when work exists upstream, creating bubbles. We therefore apply the queuing-theoretic result of Lu *et al.* [44] (formalized in Theorem VI.1), which characterizes the minimum queue depth required between a dispatcher (server) and downstream service nodes to maintain a steady state under delayed feedback. In particular, provisioning a queue of depth at least $N + N\mu C$ ensures that the scheduler can continuously supply tasks despite the C-cycle delay and stochastic service-time variation. As a result, whenever the system is backlogged, all N pipelines remain fully utilized, achieving zero-bubble scheduling and sustaining peak throughput.

Theorem VI.1 (Minimum Buffer Queue Depth)**.** *Consider a system with N independent servers processing tasks, and each server is capable of processing up to μ tasks per cycle. Tasks are scheduled from a queue by a scheduler that receives feedback about server availability with a maximum delay of C_{max} time. To ensure that all servers remain fully utilized without idleness, the minimum queue depth D must satisfy:*

$$D = N + O(\mu C_{max} N). \tag{2}$$

C. From Queuing Guarantees to Hardware Scheduler Architecture

Theorem VI.1 provides the buffering condition under which bubbles caused by delayed feedback can be eliminated in principle. We next translate this condition into a hardware-realizable scheduler: a fully pipelined, branch-free dispatch fabric that uses FIFO full/empty signals to route tasks and leverages the derived queue depth to absorb workload imbalance across the N asynchronous pipelines.

The Zero-Bubble Scheduler consists of three core functional modules: ❶, ❷, and ❸ as illustrated in Figure 7a. Together, these modules form a pipelined scheduling datapath designed to achieve high throughput and bubble-free execution. Module ❶ serves as the initial task balancer, adaptively distributing newly injected queries from the query loader to available scheduling paths based on pipeline availability. It ensures that incoming tasks are scheduled without stalls introduced by bulk query loading. Module ❷ acts as the task merger, combining newly scheduled queries from module ❶ with unfinished queries returned from downstream pipelines. This module prioritizes in-flight unfinished queries and enables their immediate redirection into the scheduling pipeline without incurring additional waiting delay. Finally, module ❸ functions as the backpressure-aware dispatcher, routing ready-to-run tasks to the appropriate processing pipelines based on real-time availability signals. This multistage coordination enables the scheduler to dynamically balance load and fill execution slots as they are released, thereby eliminating bubbles and sustaining maximum pipeline utilization.

1) N-to-N Task Balancer Design: Conventional methods for scheduling N tasks across N processors, including the Completely Fair Scheduler [45]–[47], require atomic scheduling to assign all tasks in $O(N \log N)$ time complexity. The scheduler has to select and commit a global scheduling decision by polling the state of all processors and assigning tasks in a centralized, indivisible operation. Each scheduling action involves $O(\log N)$ complexity per task and introduces strong synchronization dependencies among processors.

In contrast, our task balancer decomposes the scheduling process into independent pairwise comparisons between tasks using lightweight *Dispatcher* and *Merger* modules, each operating in $O(1)$ time. These modules are further scaled with a multistage topology based on a butterfly network, as illustrated in Figure 7b. In each stage, the result of a two-task comparison is propagated forward and participates in the next stage's arbitration to resolve the imbalanced distribution. Every module is fully pipelined, allowing line-rate scheduling that continuously adapts to the runtime workload distribution.

Figure 7b exemplifies how the task balancer smooths out downstream load imbalance. Assume the third output channel is limited to 4 pkt/s while the others sustain 100 pkt/s. In

Algorithm VI.1: Balanced task dispatch.

```
   Input:  in : input task stream.
   Output: out_1 : the first output task stream;
           out_2 : the second output task stream.
1  last_selection = 0;
2  do
3      if task = non_blocking_read() then
           scode = build_scode(last_selection,
4                              out_1.is_full(),
                               out_2.is_full());
5          switch scode do
6              case 0b001 do
                   // Both have space; pick
                      not-last-served to alternate
                      (select out_1).
7              case 0b111 do
                   // Both full; block on not-last-served
                      (out_1) to guarantee fairness.
8              case 0b101, 0b100 do
                   // Only one channel can accept (out_2
                      full); route to out_1 to avoid
                      stalling.
9                  out_1.blocking_write(task);
10                 last_selection = 0;
11                 break;
12             otherwise do
13                 out_2.blocking_write(task);
14                 last_selection = 1;
15                 break;
16 while True;
```

Algorithm VI.2: Balanced task merge.

```
   Input:  in_1 : the first input task stream;
           in_2 : the second input task stream.
   Output: out : output stream.
1  last_selection = 0;
2  do
       scode = build_scode(last_selection,
3                          in_1.is_empty(),
                           in_2.is_empty());
4      switch scode do
5          case 0b111, 0b110 do
               // Both inputs empty.
6              break;
7          case 0b101, 0b100 do
               // Only one input has valid data;
                  forward it directly to the output
                  (select in_1).
8          case 0b001 do
               // Both inputs are valid; alternate
                  based on not-last-served to avoid
                  starvation under congestion (select
                  in_1).
9              task = in_1.non_blocking_read();
10             out.blocking_write(task);
11             last_selection = 0;
12             break;
13         otherwise do
14             task = in_2.non_blocking_read();
15             out.blocking_write(task);
16             last_selection = 1;
17             break;
18 while True;
```

the second stage, each *Dispatcher* feeding the slow channel receives traffic from one fast and one slow path, averaging $(100+4)/2 = 52$ pkt/s. This imbalance is further averaged in the first stage, equalizing all four inputs at 76 pkt/s. Thus, the multistage routing network spreads local congestion upstream and keeps earlier stages uniformly loaded even when a single downstream channel is throttled.

2) Task Dispatcher: Algorithm VI.1 details the *Dispatcher* that routes tasks from a single input stream to two independent output channels while honoring back-pressure and preserving fairness. The Dispatcher maintains a one-bit state, `last_selection`, initialized in Line 1, to record which output was served most recently. In each iteration (Line 2), it first attempts a *non-blocking* read from the input (Line 3). If no task is available, the Dispatcher simply skips the iteration without stalling upstream. Once a task is obtained, the Dispatcher constructs a compact three-bit scheduling code, *scode*, using the `build_scode()` function (Line 4) by packing `last_selection` with the `is_full` status of *out_1* and *out_2*. Specifically, `last_selection` occupies the least significant bit (LSB), while the full flag of *out_2* is placed in the most significant bit (MSB).

Guaranteeing Balance under Worst-case Congestion. This encoding enables a simple decode via a `switch` (Lines 5-15) rather than complex runtime control branching. In the case that the last task is sent to *out_2*, when both outputs are available, the Dispatcher selects the *not-last-served* channel to alternate service and balance load at Line 6. When both outputs are full, it blocks on the *not-last-served* channel to prevent persistent preemption of one side, guaranteeing fairness under worst-case congestion (Line 7). When only one output can accept new data, the task is routed directly to the available channel (e.g., Line 8). After each `blocking_write`, the Dispatcher updates `last_selection` to reflect the chosen output (Lines13–14), ensuring consistent alternation in subsequent iterations. Overall, the Dispatcher is fully pipelined with a one-cycle initiation interval and a fixed latency of two cycles.

3) Task Merger: Algorithm VI.2 presents the algorithm implemented in *Merger*, which combines two input task streams into a single output stream while maintaining balanced service under back-pressure. The Merger tracks the most recently selected input using a one-bit state, `last_selection` (Line 1). In each iteration (Line 2), it

constructs a three-bit scheduling code, *scode* (Line 3), in which `last_selection` occupies the LSB, and the empty flags of *in_1* and *in_2* are stored in the next two bits, respectively.

Guaranteeing Fairness under Worst-case Congestion. The scheduling policy is as follows: If exactly one input contains valid data, it forwards that input directly to the output to maximize throughput (e.g., Line 7 and the default case, regardless of the last selection). When both inputs are valid, the Merger selects the *not-last-served* stream based on `last_selection` to alternate input (Line 8). This prevents the output from being continuously occupied by a single input under congestion, avoids starvation, and bounds the worst-case waiting latency for the other stream. After each successful forward, `last_selection` is updated to reflect the chosen input (Lines 11), ensuring persistent balance in subsequent iterations. Similar to the *Dispatcher*, the Merger is fully pipelined with a one-cycle initiation interval and a fixed latency of two cycles.

D. Achieving Perfect Pipelining

We now analyze the feedback delay to apply Theorem VI.1 to eliminate any pipeline bubbles via scheduling. In the butterfly topology, each task traverses $\log N$ *Dispatcher* and $\log N$ *Merger* units on the task balancer. Since each unit is fully pipelined with at most two cycles of latency, the total delay through the task balancer is bounded by $2\log N$ cycles. Adding in the round-trip delay from the scheduler to the selected pipeline and back, the total scheduling latency is at most $4\log N$ cycles. Given that each pipeline sustains an ideal throughput of one GRW step per cycle (i.e., $\mu = 1$), Theorem VI.1 establishes that a queue depth of $D = N + 4N\log N$ is the minimum required between the scheduler and pipelines to guarantee zero-bubble execution. This corresponds to a FIFO per pipeline with a depth of $1 + 4\log N$, ensuring that no pipeline is idle due to input starvation.

Throughput and Latency Analysis. Our scheduler is designed to eliminate runtime throughput overhead. It employs a fully pipelined scheduling datapath with a one-cycle initiation interval, ensuring that both newly injected and redirected queries are concurrently scheduled among multiple processing pipelines at maximum throughput without stalling. The fixed latency incurred by redirected queries is small (e.g., eight cycles for 16 pipelines), fully overlapped, and amortized with pipeline execution. From the perspective of downstream pipelines, queries remain continuously available, preventing idle cycles due to scheduler delays.

VII. Adaptation to Different GRWs

RidgeWalker is designed to be modular and extensible, supporting a wide range of GRW algorithms used in graph-learning applications. The sampling module communicates via a standard AXI-Stream interface, enabling flexible integration of custom sampling logic. Each instance is paired with ThundeRiNG [48], an FPGA-optimized, high-throughput random number generator. The task input format encapsulates full query session context, allowing algorithm-specific sampling behavior. Additionally, the graph representation is template-based to support weighted edges and extended metadata. In particular, the row pointer entry (RP_{entry}) size is configurable at compile time, allowing customization to accommodate auxiliary structures such as alias tables.

TABLE I: The supported sampling algorithm and GRWs.

GRWs	Weighted $\mathcal{G}$?	Sampling algorithm	$\lvert RP_{entry}\rvert$
URW [49], PPR [50]	No	Uniform sampling	64-bit
DeepWalk [5]	Yes	Alias sampling	256-bit
Node2Vec [9]	No	Rejection sampling	64-bit
Node2Vec [18]	Yes	Reservoir sampling	128-bit
MetaPath [8]	Yes	Reservoir sampling	128-bit

RidgeWalker supports all commonly used sampling algorithms for both unweighted and weighted graphs, as summarized in Table I. Uniform sampling is used by URW [49] and PPR [50]. DeepWalk [5] employs alias sampling [51], [52], which requires each neighbor list to maintain an alias table. To accommodate this, the RP_{entry} format is extended to 256 bits, storing the alias table pointer and its size. For Node2Vec [9], RidgeWalker supports both rejection sampling and reservoir sampling, which are suitable for unweighted and weighted graphs, respectively.

To configure the sampling module, RidgeWalker exposes memory-mapped `AXI4-Lite` control registers over PCIe, allowing the host to program algorithm-specific parameters such as the teleport probability α in PPR or the bias factors p and q in Node2Vec. All configuration fields are accessible via lightweight 32-bit register writes. Additionally, sampling modes (e.g., weighted vs. unweighted) can be selected via a mode bit, enabling rapid switching between GRW variants without requiring full resynthesis.

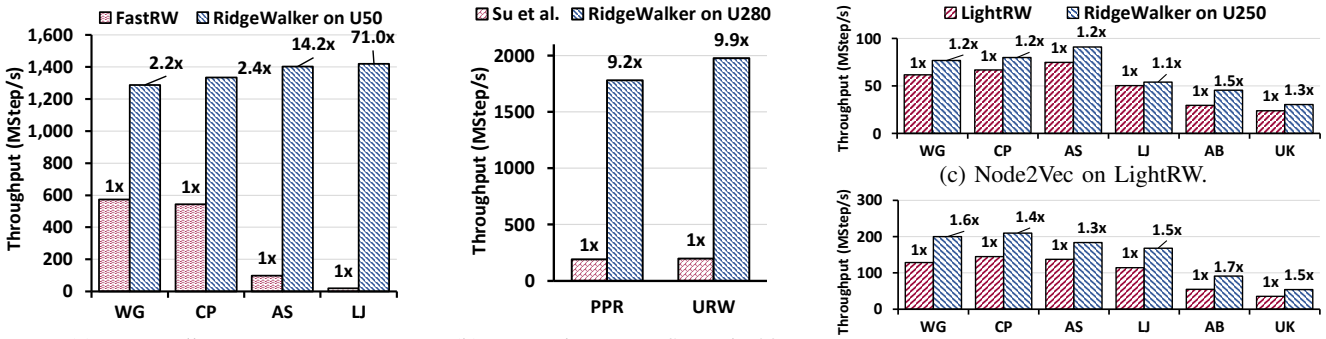
VIII. Evaluation

In this section, we present a comprehensive evaluation of RidgeWalker, focusing on its asynchronous execution and perfect pipelining. We compare it against state-of-the-art solutions and structure our evaluation around the following objectives:

1. Evaluate the performance of RidgeWalker against state-of-the-art FPGA-based accelerators and highlight the benefits of its perfectly pipelined architectural design (§VIII-B).
2. Compare RidgeWalker with high-performance GPU-based solutions, and demonstrate its architectural advantages despite operating under lower available bandwidth (§VIII-C).
3. Quantitatively analyze the effectiveness of the proposed asynchronous access engine and zero-bubble scheduler through microbenchmarking and an ablation study, demonstrating how we address Observation #1 and #2 (§VIII-D).

A. Experimental Setup

1) Hardware Setup: We prototype the RidgeWalker design using high-level synthesis (HLS) and mainly evaluate its performance on AMD Alveo U55C FPGA boards. The performance experiments are conducted on a dual-socket AMD EPYC 7V13 128-core CPU server. Since each U55C FPGA supports up to 32 HBM memory channels, and each asynchronous pipeline occupies two HBM channels, we instantiate



(a) DeepWalk on FastRW. (b) PPR and URW on Su et al. [23]. (c) Node2Vec on LightRW. (d) MetaPath on LightRW.

Fig. 8: Comparison of throughput between RidgeWalker and SOTA FPGA-based GRW accelerators.

TABLE II: The evaluated real-world graph datasets.

Graphs	$\|V\|$	$\|E\|$	Size	Categories	δ
web-Google (WG) [53]	0.9 M	5.1 M	48 MB	Web	21
cit-Patents (CP) [53]	3.8 M	16.5 M	0.2 GB	Citation	26
as-Skitter (AS) [53]	1.7 M	22.2 M	0.2 GB	Network	31
soc-LiveJournal (LJ) [53]	4.9 M	69.0 M	0.6 GB	Social	28
arabic-2005 (AB) [54]	22.7 M	0.6 B	5.0 GB	Web	133
uk-2005 (UK) [54]	39.6 M	0.8 B	6.7 GB	Web	45

$32/2 = 16$ asynchronous processing pipelines, following the architecture design in Section IV. The Zero-Bubble Scheduler is correspondingly configured with 16 outputs connecting to the asynchronous processing pipeline.

2) FPGA Baselines: The design of Su *et al.* [23], FastRW [22] and LightRW [18] are state-of-the-art accelerators for PPR, URW, DeepWalk, Node2Vec and MetaPath, respectively. To ensure a fair evaluation, RidgeWalker is re-synthesised on the same FPGAs used by these baselines as the AMD Alveo U50 (FastRW) and AMD Alveo U250 (LightRW). Because RidgeWalker is architecture-agnostic, porting involved only retiming the memory interface number of memory channels.

3) GPU Baseline: For GPU-based solutions, we benchmark RidgeWalker against the state-of-the-art solution, gSampler [15], as gSampler provides the best acceleration performance across all evaluated applications. The experiments are conducted on a server equipped with four NVIDIA H100 PCIe GPUs (H100) features 80 GB of HBM2e memory with a bandwidth of 2093 GB/s running CUDA 12.1.

4) GRW Workloads: Table II presents the graph datasets evaluated in our experiments, arranged in ascending order based on the number of edges. These datasets encompass a variety of real-world graphs, including web, network, citation, and social networks. The fifth column of the table (δ) displays the graph's diameter. That is the longest shortest path between any two vertices.

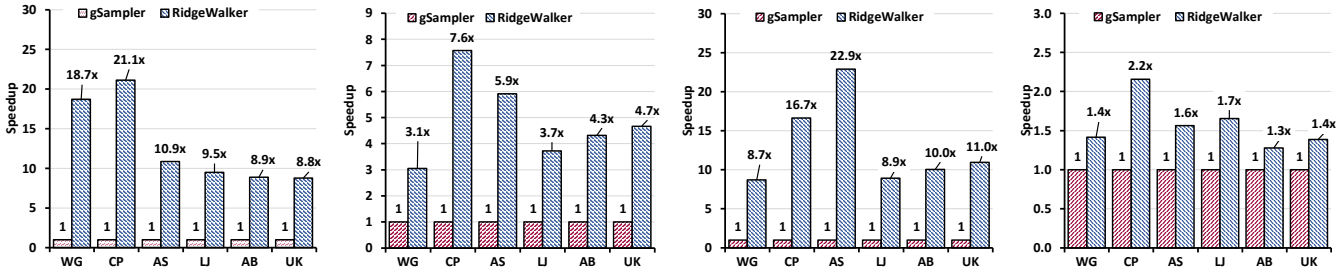
GRW Algorithms. We evaluate four commonly used GRW algorithms from graph database and graph ML application: PPR [50], URW [49], DeepWalk [5], and Node2Vec [9], where DeepWalk and Node2Vec are extensively used in GNN workloads. To ensure consistency with existing benchmarks [15]–[17], we set the query length to 80. For Node2Vec, the parameters are set $p = 2$ and $q = 0.5$, as employed in other state-of-the-art works [15]–[17]. The edge weights are generated according to the ThunderRW method [16]. For a fair comparison with existing systems, we consistently employ the same state-of-the-art sampling algorithms and configurations, without altering their algorithmic semantics.

Performance Metrics. We evaluate GRW execution performance based on effective throughput, which is independent of variations in workload characteristics such as the number of input queries and graph sizes. Throughput is measured in millions of steps per second (MStep/s) and is calculated by dividing the *total count of visited vertices* by the *end-to-end query-processing time*. The processing time is measured as follows: each GRW system is first warmed up and operates as a runtime service with the input graph pre-loaded into main memory. Queries are issued as a continuous stream to emulate real-world application behavior. Each experiment is repeated five times using the same query input, and the median result is reported to reduce the influence of outliers.

B. Comparison to SOTA FPGA-based Solutions

Figure 8a compares the DeepWalk throughput of RidgeWalker against FastRW [22], a state-of-the-art GRW accelerator. Since FastRW's code is not publicly available, we implement RidgeWalker on the same hardware platform, the Alveo U50 FPGA, and compare its performance using datasets reported in their paper. RidgeWalker consistently outperforms FastRW across all datasets, with speedups increasing with graph size. On the largest dataset (*LJ*), RidgeWalker achieves a $70.98\times$ improvement. This is because FastRW relies on caching to store small graphs in limited on-chip memory, which becomes ineffective for large graphs due to GRW's inherently poor locality. In contrast, RidgeWalker is optimized for random access over large graphs residing in HBM.

Even on small graphs like *WG*, which FastRW can cache most of the graph to the on-chip fast memory, RidgeWalker also achieves a $2.24\times$ speedup. This is attributed to our out-of-order execution and pipelined pseudo-random number generation using ThundeRiNG [48], which avoids additional HBM traffic. FastRW, by contrast, pre-generates random numbers



(a) Personalized PageRank (b) Uniform random walk (c) DeepWalk (d) Node2Vec

Fig. 9: Comparison of normalized throughput to gSampler on four GRW applications.

on the CPU and has to load them to HBM, consuming the bandwidth that could otherwise be used for graph access.

Figure 8b compares throughput for PPR and URW on the *WG* graph with the accelerator proposed by Su *et al.* [23]. Since their code and evaluations are limited to this dataset, we restrict the comparison accordingly. RidgeWalker achieves $9.21\times$ and $9.94\times$ higher throughput for PPR and URW, respectively, primarily due to its efficient memory subsystem that fully utilizes HBM bandwidth.

Figure 8c compares RidgeWalker's Node2Vec throughput to LightRW [18], using reservoir sampling, the same sampling method used by LightRW, both implemented on the Alveo U250 FPGA. RidgeWalker delivers $1.1\times$–$1.5\times$ higher performance. While both designs are efficient, LightRW uses batched execution, which introduces pipeline stalls. In contrast, RidgeWalker adopts fine-grained scheduling, eliminating bubbles and achieving better resource utilization.

A similar trend is observed for MetaPath random walks on weighted graphs. As shown in Figure 8d, RidgeWalker achieves a $1.3\times$ to $1.7\times$ speedup over LightRW, due to the higher likelihood of early termination. In MetaPath walks, the next-hop must match a specific type [16]; if no such neighbor exists, the walk ends early. RidgeWalker leverages the *Zero-Bubble Scheduler* to maintain high pipeline utilization under this runtime irregularity, consistently outperforming LightRW.

C. Comparison to SOTA GPU-based Solutions

1) Analysis on Real-world Graphs: Figure 9a shows that RidgeWalker outperforms gSampler on PPR by $8.8\times$–$21.1\times$ across all six graphs, with a peak gain of $21.1\times$ on `CP`. The key differentiator is RidgeWalker's Zero-Bubble Scheduler. By re-routing ready tasks every cycle, it keeps all pipelines busy even when PPR walks terminate at wildly different lengths, thereby preserving a perfectly filled pipeline and near-peak random access memory bandwidth. While gSampler relies on *super batching* to reduce kernel-launch overhead on the GPU, each warp must still wait for the slowest thread. When a random walk ends early, the idle threads induce significant divergence overhead.

Figure 9b reports the speedup of RidgeWalker over gSampler on URW. Across all datasets RidgeWalker is faster, with gains from $3.1\times$ on `WG` to $7.6\times$ on `CP`. The largest

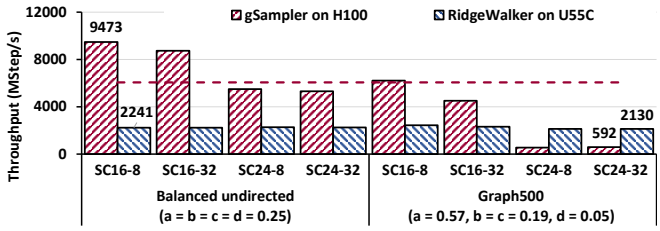


Fig. 10: Performance comparison on RMAT graphs under balanced and Graph500 configurations.

improvements on `CP` and `AS` arise because our Asynchronous Memory Access Engine saturates random-access bandwidth. The smaller gain on `WG` reflects its compact size, which fits largely in GPU cache, and the moderate gain on `LJ` is due to its undirected structure, which reduces workload imbalance. Even so, RidgeWalker delivers consistent speedups across graphs.

Figure 9c compares RidgeWalker with gSampler on DeepWalk. RidgeWalker delivers speedups from $8.7\times$ (`WG`) to $22.9\times$ (`AS`); gains exceed $10\times$ on `CP` and `AB`. DeepWalk relies on alias sampling, which doubles the number of pseudo-random numbers and increases GPU instruction count, limiting gSampler to just 0.9–2.4 % of peak bandwidth. RidgeWalker fully pipelines sampling and random-number generation, achieving URW-level throughput. These results highlight the importance of pipeline optimization and bandwidth utilization for large-scale graph embedding.

Figure 9d shows the speedup of RidgeWalker over gSampler on Node2Vec using rejection sampling [15]. Here the speedups are more modest, ranging from $1.28\times$ on `AB` to $2.16\times$ on `CP`. This outcome is expected because Node2Vec's biased walks introduce more structured sequential access on the neighbor list, allowing GPU hardware to capture locality from bulked access.

2) Analysis on Synthetic Graphs: To investigate and further breakdown the performance benefits of RidgeWalker, we compare it against DeepWalk in gSampler on H100 GPU using synthetic RMAT [55] graphs with varying size and density. Two initiator configurations are evaluated: the balanced undirected setup (RMAT probability distribution is set as a=b=c=d=0.25) and the Graph500 configuration (a=0.57, b=c=0.19, d=0.05) [56]. Each graph is labeled as SCX-Y, where X represents the scale factor and Y denotes the edge

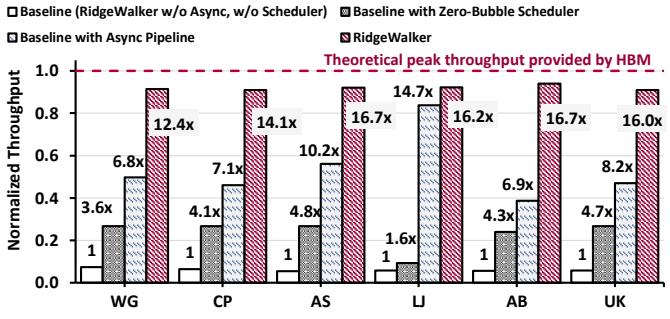


Fig. 11: Breakdown of performance gains from the Asynchronous Pipeline and Zero-Bubble Scheduler

factor (e.g., SC24-32 corresponds to scale 24 and edge factor 32).

The red dashed line in Figure 10 represents the benchmarked upper bound of GRW throughput on the H100 GPU, derived from its measured random-access memory bandwidth (excluding cache effects). derived from the random-access bandwidth benchmark [57]. On SC24 balanced graphs, gSampler's throughput closely approaches this line, showing that GPU execution achieves near-peak random-access efficiency when accesses are evenly distributed.

Although the GPU achieves high absolute throughput on balanced RMAT graphs, the picture changes dramatically under the skewed Graph500 configuration. Graph500 introduces a strong structural imbalance, causing traversing lengths to vary significantly across queries. Under the SIMT execution model, GPU warps execute in lockstep, so threads that finish early must stall until the longest walk completes. This leads to heavy divergence and underutilization, reducing throughput by more than an order of magnitude compared with the balanced case. Increasing the number of queries does not help, as millions of concurrent walks are already issued and all SMs are fully saturated.

RidgeWalker maintains consistently high throughput across all RMAT configurations. Our stateless task decomposition and zero-bubble scheduling enable fully independent per-hop execution, allowing short and long walks to proceed without blocking one another. As a result, RidgeWalker continues to deliver around 2,130 MSteps/s even on the heavily skewed graphs, demonstrating that architectural tolerance to workload imbalance can outweigh raw bandwidth advantages. This highlights a key strength of our design: RidgeWalker converts irregular GRW workloads into perfectly pipelined execution, allowing it to fully exploit the hardware potential.

D. Breakdown Evaluation

Figure 11 reports the normalized throughput across all real-world graphs, with all values shown relative to the theoretical HBM-supported throughput (indicated by the red dashed line and computed using Equation (1)). Each bar reflects both the achieved performance and its speedup over our breakdown baseline. The baseline retains the overall RidgeWalker architecture but disables both asynchronous execution and dynamic scheduling: queries are statically bound to fixed pipelines, memory requests are issued directly to HBM via a standard AXI interface, and execution proceeds in bulk-synchronous batches without early-termination handling, similar to LightRW [18] and FastRW [22]. We then progressively enable individual optimizations to isolate their contributions.

First, enabling only the zero-bubble scheduler allows early-terminating queries to be dynamically redistributed across pipelines. This eliminates pipeline bubbles and improves performance by $1.6\times$–$4.8\times$. The improvement is small on *LJ*, whose undirected structure results in very few early terminations. Given that nearly 80% of all 418 graphs from KONECT [58] are directed, the zero-bubble scheduler is crucial for sustaining high utilization in practical GRW workloads, directly addressing Observation #2, mitigating the substantial performance loss caused by early termination in GRW queries.

Second, enabling the asynchronous pipeline and the asynchronous memory-access engine, while keeping the scheduler disabled, isolates the effect of latency hiding and out-of-order task execution. This design improves baseline performance by $6.8\times$ to $14.7\times$, demonstrating that decoupled memory access and per-hop task independence are essential for mitigating memory-latency bottlenecks in GRW execution. It addresses Observation #1 over prior work by efficiently amortizing pointer-chasing latency across multiple concurrent queries.

Finally, enabling both optimizations together yields perfectly pipelined execution across massive numbers of concurrent queries. Under this configuration, RidgeWalker achieves up to $16.7\times$ speedup over the baseline and reaches up to 88% of the theoretical HBM random-access peak.

TABLE III: Average URW throughput of all graph datasets across FPGA with different memory configurations

	U250	VCK5000	U50	U55C
Memory type	DDR4	DDR4-NoC	HBM2	HBM2
Bandwidth (GB/s)	77	102	316	460
# of memory channel	4	4	32	32
Throughput (MStep/s)	258	202	1463	2098
BW utilization	81%	87%	88%	88%

E. Support on Different FPGAs

To demonstrate the generalizability of RidgeWalker's architecture, we evaluate its performance on four FPGA platforms: U250, VCK5000, U50, and U55C. These devices vary in available memory bandwidth and the number of independent memory channels, as summarized in the second and third rows of Table III. We measure both the throughput (fourth row) and random-access bandwidth utilization (fifth row) of URW across all graph datasets listed in Table II, and report their average values. The Versal VCK5000 includes a hardened network-on-chip (NoC) memory subsystem with four DDR4 channels, providing up to 102 GB/s aggregate bandwidth. To support irregular access patterns, we disable default NoC channel interleaving, typically optimized for sequential workloads, and deploy RidgeWalker with four processing pipelines. Overall, RidgeWalker sustains bandwidth utilization from 81% to 88% across all evaluated FPGA platforms. RidgeWalker

TABLE IV: The consumption of hardware resource (percentage) and frequency (MHz) of different GRWs on U55C FPGA.

App.	LUTs	REGs	BRAMs	DSPs	Frequency
PPR	61.1%	29.8%	19.5%	2.2%	320MHz
URW	50.1%	24.0%	19.5%	2.2%	320MHz
DeepWalk	67.5%	32.3%	39.1%	4.4%	320MHz
Node2Vec	79.1%	41.6%	36.0%	7.3%	320MHz

requires only independent memory channels that support the standard AXI protocol, facilitating portability across different FPGAs.

F. Resource Utilization and Frequency Optimization

Table IV reports the resource utilization and operating frequency of RidgeWalker for four GRW kernels implemented on the U55C FPGA. Because GRWs are fundamentally memory bound, the design focuses on saturating the available random access memory bandwidth and channels rather than on exhausting logic resources, leaving ample capacity for additional, downstream accelerators. Resource usage varies with the sampling method and the size of each row-pointer entry (RP_{entry}, see Section VII). Benefiting from the asynchronous execution model, operators and modules are decoupled, simplifying timing closure and supporting a frequency up to 320 MHz.

RidgeWalker implements shallow FIFOs with LUTs, streamlining placement and routing for high-frequency. BRAM is used for deeper buffers, as each block can store up to 512 entries. Two components rely on BRAM-based FIFOs: (1) a 128-entry metadata queue in the asynchronous memory-access engine, sized to absorb the HBM round-trip latency, and (2) 65-entry FIFOs between the scheduler and the pipelines, which maintain a steady query processing and prevent stalls.

We further optimize the zero-bubble scheduler to prevent the butterfly interconnect from becoming a critical path. We inserted registers to break long combinational logic paths, and the entire module is designed as a free-running module without global control signals such as start/stop triggers. This eliminates high-fanout broadcast logic, improving timing for higher frequency. Independent profiling on the U55C FPGA shows that the scheduler operates at up to 450 MHz while consuming only 1.8% of available LUTs, demonstrating its potential scalability beyond 32 HBM channels.

IX. Related Work and Discussion

ASIC and In-Memory Designs. Several works tackle the memory-bound nature of random walks by bringing computation closer to data. FlashWalker [59] embeds walk logic within solid-state drives, leveraging NAND-level parallelism to eliminate host-device transfers. Other efforts explore *processing-in-memory* (PIM) using emerging technologies such as ReRAM, enabling graph processing directly within memory arrays to reduce data movement and latency [60]. However, current PIM designs support only very small graph sizes and remain limited in scalability. While ASIC and PIM approaches offer performance potential, they face challenges in adaptability to evolving algorithms and long development cycles. FPGA-based solutions are flexible to deploy and adaptive to the fast evolution of GRW algorithms.

Supercomputing Architectures for Graph Processing. General-purpose multithreaded architectures such as the Cray XMT (originally Tera MTA) [61], [62] and Lucata Emu [63], [64] have been explored for irregular graph processing workloads, including BFS and SSSP. These systems tolerate memory latency by maintaining hundreds of hardware thread contexts and issuing instructions from any ready thread at each time. RidgeWalker takes a different approach by adopting a *domain-specific architecture* tailored for GRWs. Leveraging the Markov property, it allows query tasks to flow directly through the pipeline as fine-grained stateless units, enabling out-of-order execution without the need to maintain thread contexts or global walk state in memory. Furthermore, RidgeWalker employs a scheduler that is formally grounded in queueing theory, sustaining near-optimal throughput even under highly diverse and imbalanced GRW workloads.

Sparse Data Structure Fetchers. Widx [65] accelerates database hash-index lookups using a stateful design that offloads pointer chasing and key hashing to programmable on-chip units with per-key state in dispatcher tables and local caches. Fifer [66] employs coarse-grained time-multiplexed reconfiguration to tolerate memory latency and balance irregular workloads. TMU [67] targets data-movement–intensive tensor operators using reconfigurable address abstractions for coarse- and fine-grained data rearrangement. Terminus [68] generalizes sparse data structure acceleration with per-operation and per-partition state to support fine-grained updates on hash tables and trees. Aurochs [69] extends dataflow accelerators to execute irregular structures via stateful fine-grained hardware threading. These designs are mainly stateful, maintaining per-query or per-task context in registers, caches, or buffers to overlap memory access and computation. In contrast, RidgeWalker adopts a stateless task decomposition model based on the Markov property, enabling flexible scheduling and massive parallelism. The architecture is specialized for GRW workloads, achieving near-optimal random-access throughput and high efficiency for emerging large-scale graph applications.

Discussion. RidgeWalker employs standard architecture primitives rather than FPGA-specific features, hence it can be easily ported to ASICs and complements existing logic. Furthermore, our perfect pipelining strategy generalizes to other probabilistic workloads, such as Bayesian networks and Monte Carlo Markov Chain applications, where runtime dependencies and random access latency critically affect performance.

X. Conclusion

This paper introduces RidgeWalker, a perfect pipelined FPGA-based accelerator that leverages the Markovian property of GRWs to decompose queries into stateless tasks. By combining several asynchronous pipelines with the proposed zero-bubble scheduler on FPGAs, RidgeWalker maximizes parallelism and achieves 88% runtime utilization of the theoretical random-access memory bandwidth. RidgeWalker delivers up

to 71.0× and 22.9× speedup over state-of-the-art FPGA and GPU solutions, respectively. Moreover, RidgeWalker's modular design supports diverse GRW algorithms, offering a scalable and efficient architecture for high-performance GRWs.

ACKNOWLEDGEMENTS

This research/project is supported by the Ministry of Education AcRF Tier 2 grant (No. MOE-T2EP20224-0020) and Tier 1 grant (No. T1 251RES2315) in Singapore, the Google South & Southeast Asia Research Award 2025, and the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant. We also thank the AMD Heterogeneous Accelerated Compute Clusters (HACC) program [70] for the generous hardware donation. Yao Chen from the National University of Singapore is the corresponding author.

REFERENCES

[1] R. Lambiotte, J.-C. Delvenne, and M. Barahona, "Random walks, markov processes and the multiscale modular organization of complex networks," *IEEE Transactions on Network Science and Engineering*, vol. 1, no. 2, pp. 76–90, 2015.

[2] Y. Shao, S. Huang, X. Miao, B. Cui, and L. Chen, "Memory-aware framework for efficient second-order random walk on large graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1797–1812. [Online]. Available: https://doi.org/10.1145/3318464.3380562

[3] M. Liao, R.-H. Li, Q. Dai, H. Chen, H. Qin, and G. Wang, "Efficient personalized pagerank computation: The power of variance-reduced monte carlo approaches," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.

[4] X. Yu, X. Ren, Y. Sun, Q. Gu, B. Sturt, U. Khandelwal, B. Norick, and J. Han, "Personalized entity recommendation: A heterogeneous information network approach," in *Proceedings of the 7th ACM international conference on Web search and data mining*, 2014, pp. 283–292.

[5] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[6] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, 2013, pp. 141–147.

[7] Z. Li, D. Fu, and J. He, "Everything evolves in personalized pagerank," in *Proceedings of the ACM Web Conference 2023*, ser. WWW '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3342–3352. [Online]. Available: https://doi.org/10.1145/3543507.3583474

[8] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 135–144.

[9] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[10] L. Cappelletti, T. Fontana, E. Casiraghi, V. Ravanmehr, T. J. Callahan, C. Cano, M. P. Joachimiak, C. J. Mungall, P. N. Robinson, J. Reese *et al.*, "Grape for fast and scalable graph processing and random-walk-based embedding," *Nature Computational Science*, vol. 3, no. 6, pp. 552–568, 2023.

[11] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[12] Q. Liu, L. Jiang, M. Han, Y. Liu, and Z. Qin, "Hierarchical random walk inference in knowledge graphs," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, 2016, pp. 445–454.

[13] X. Wang, X. He, and T.-S. Chua, "Learning and reasoning on graph for recommendation," in *Proceedings of the 13th international conference on web search and data mining*, 2020, pp. 890–893.

[14] B. Jimenez Gutierrez, Y. Shu, Y. Gu, M. Yasunaga, and Y. Su, "Hipporag: Neurobiologically inspired long-term memory for large language models," *Advances in Neural Information Processing Systems*, vol. 37, pp. 59 532–59 569, 2024.

[15] P. Gong, R. Liu, Z. Mao, Z. Cai, X. Yan, C. Li, M. Wang, and Z. Li, "gsampler: General and efficient gpu-based graph sampling for graph learning," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 562–578.

[16] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, "Thunderrw: an in-memory graph random walk engine," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 1992–2005, 2021.

[17] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: a fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 524–537.

[18] H. Tan, X. Chen, Y. Chen, B. He, and W.-F. Wong, "Lightrw: Fpga accelerated graph dynamic random walks," *Proc. ACM Manag. Data*, vol. 1, no. 1, may 2023. [Online]. Available: https://doi.org/10.1145/3588944

[19] H. Yin, Y. Shao, X. Miao, Y. Li, and B. Cui, "Scalable graph sampling on gpus with compressed graph," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 2383–2392.

[20] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using gpus," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 311–326.

[21] P. Wang, C. Xu, C. Li, J. Wang, T. Wang, L. Zhang, X. Hou, and M. Guo, "Optimizing gpu-based graph sampling and random walk for efficiency and scalability," *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 2508–2521, 2023.

[22] Y. Gao, T. Wang, L. Gong, C. Wang, X. Li, and X. Zhou, "Fastrw: A dataflow-efficient and memory-aware accelerator for graph random walk on fpgas," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[23] C. Su, H. Liang, W. Zhang, K. Zhao, B. Ai, W. Shen, and Z. Wang, "Graph sampling with fast random walker on hbm-enabled fpga accelerators," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 211–218.

[24] X. Chen, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Hls-based graph processing framework on fpgas," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 69–80.

[25] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W.-F. Wong, "Regraph: Scaling graph processing on hbm-enabled fpgas with heterogeneous pipelines," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1342–1358.

[26] L. Stasytis and Z. István, "Optimization techniques for hestenes-jacobi svd on fpgas," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2023, pp. 144–150.

[27] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, "Dmazerunner: Executing perfectly nested loops on dataflow accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–27, 2019.

[28] J. Davis and R. Reese, *Finite State Machine Datapath Design, Optimization, and Implementation*. Springer Nature, 2022.

[29] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 211–218.

[30] W. Jaiyeoba, N. Elyasi, C. Choi, and K. Skadron, "Acts: a near-memory fpga graph processing framework," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 79–89.

[31] O. Jaiyeoba and K. Skadron, "Dynamic-acts-a dynamic graph analytics accelerator for hbm-enabled fpgas," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 3, pp. 1–29, 2024.

[32] O. Jaiyeoba, A. T. Mughrabi, M. Baradaran, B. Gul, and K. Skadron, "Swift: A multi-fpga framework for scaling up accelerated graph analytics," 2024.

[33] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu, "HitGraph: High-throughput graph processing framework on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2249–2264, 2019.

[34] Y. Hu, Y. Du, E. Ustun, and Z. Zhang, "Graphlily: Accelerating graph linear algebra on hbm-equipped fpgas," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[35] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, 2020, pp. 908–921.

[36] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "Jetstream: Graph analytics on streaming data with event-driven hardware accelerator," in *Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*, 2021, pp. 1091–1105.

[37] P. Yao *et al.*, "Scalagraph: A scalable accelerator for massively parallel graph processing," in *Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 199–212.

[38] X. Chen, F. Cheng, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "Thundergp: Resource-efficient graph processing framework on fpgas with hls," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

[39] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, "Demystifying the characteristics of high bandwidth memory for real-time systems," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[40] V. Kanade, F. Mallmann-Trenn, and T. Sauerwald, "On coalescence time in graphs: When is coalescing as fast as meeting?" *ACM Transactions on Algorithms*, vol. 19, no. 2, pp. 1–46, 2023.

[41] R. I. Oliveira and Y. Peres, "Random walks on graphs: new bounds on hitting, meeting, coalescing and returning," in *2019 Proceedings of the Sixteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. SIAM, 2019, pp. 119–126.

[42] O. Moreira, A. Yousefzadeh, F. Chersi, A. Kapoor, R.-J. Zwartenkot, P. Qiao, G. Cinserin, M. A. Khoei, M. Lindwer, and J. Tapson, "Neuronflow: A hybrid neuromorphic–dataflow processor architecture for ai workloads," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2020, pp. 01–05.

[43] J. H. Dshalalow, "An anthology of classical queueing methods," in *Advances in Queueing Theory, Methods, and Open Problems*. CRC Press, 2023, pp. 1–42.

[44] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, and X. Liu, "Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, 2003, pp. 208–217.

[45] Y. Sun, C. E. Koksal, and N. B. Shroff, "Near delay-optimal scheduling of batch jobs in multi-server systems," 2023. [Online]. Available: https://arxiv.org/abs/2309.16880

[46] F. D. Croce and R. Scatamacchia, "Longest processing time rule for identical parallel machines revisited," 2018. [Online]. Available: https://arxiv.org/abs/1801.05489

[47] P. Yu, Y. Qiu, X. Jin, and M. Chowdhury, "Orloj: Predictably serving unpredictable dnns," 2022. [Online]. Available: https://arxiv.org/abs/2209.00159

[48] H. Tan, X. Chen, Y. Chen, B. He, and W.-F. Wong, *ThundeRiNG: Generating Multiple Independent Random Number Sequences on FPGAs*. New York, NY, USA: Association for Computing Machinery, 2021, p. 115–126. [Online]. Available: https://doi.org/10.1145/3447818.3461664

[49] R.-H. Li, J. X. Yu, L. Qin, R. Mao, and T. Jin, "On random walk based graph sampling," in *2015 IEEE 31st international conference on data engineering*. IEEE, 2015, pp. 927–938.

[50] G. Hou, X. Chen, S. Wang, and Z. Wei, "Massively parallel algorithms for personalized pagerank," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1668–1680, 2021.

[51] A. J. Walker, "New fast method for generating discrete random numbers with arbitrary frequency distributions," *Electronics Letters*, vol. 8, no. 10, pp. 127–128, 1974.

[52] F. Zhang, M. Jiang, and S. Wang, "Efficient dynamic weighted set sampling and its extension," *Proceedings of the VLDB Endowment*, vol. 17, no. 1, pp. 15–27, 2023.

[53] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[54] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[55] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[56] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, no. 45-74, p. 22, 2010.

[57] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump up the volume: Processing large data on GPUs with fast interconnects," in *SIGMOD*. New York, NY, USA: ACM, 2020, pp. 1633–1649.

[58] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd international conference on world wide web*, 2013, pp. 1343–1350.

[59] F. Niu, J. Yue, J. Shen, X. Liao, H. Liu, and H. Jin, "Flashwalker: An in-storage accelerator for graph random walks," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1063–1073.

[60] D. Choudhury, L. Xiang, A. Rajam, A. Kalyanaraman, and P. P. Pande, "Accelerating graph computations on 3d noc-enabled pim architectures," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 3, pp. 1–16, 2023.

[61] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *2006 International Conference on Parallel Processing (ICPP'06)*. IEEE, 2006, pp. 523–530.

[62] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–9.

[63] E. R. Hein, S. Eswar, A. Yaşar, J. Li, J. S. Young, T. M. Conte, Ü. V. Çatalyürek, R. Vuduc, J. Riedy, and B. Uçar, "Programming strategies for irregular algorithms on the emu chick," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 4, pp. 1–25, 2020.

[64] E. Smith, S. Kuntz, J. Riedy, and M. Deneroff, "Concurrent graph queries on the lucata pathfinder," 2022. [Online]. Available: https://arxiv.org/abs/2209.11889

[65] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 468–479.

[66] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[67] W. Zhou, Z. Wang, C. Chen, Y. Li, Y. Yang, Z. Wu, and A. Chattopadhyay, "Tensor manipulation unit (tmu): Reconfigurable, near-memory tensor manipulation for high-throughput ai soc," *arXiv preprint arXiv:2506.14364*, 2025.

[68] H. R. Lee and D. Sanchez, "Terminus: A programmable accelerator for read and update operations on sparse data structures," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 1233–1246.

[69] M. Vilim, A. Rucker, and K. Olukotun, "Aurochs: An architecture for dataflow threads," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 402–415.

[70] AMD, "Heterogeneous accelerated compute clusters (hacc) program," https://www.amd-haccs.io/index.html, 2023.