FlashWalker: An In-Storage Accelerator for Graph Random Walks

Fuping Niu[†], Jianhui Yue[‡], Jiangqiu Shen[‡], Xiaofei Liao[†], Haikun Liu[†], Hai Jin[†]
[†]*National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China*
[‡]*Department of Computer Science, Michigan Technological University, Houghton, Michigan, 49931, USA*
nfp@hust.edu.cn, jyue@mtu.edu, jshen2@mtu.edu, xfliao@hust.edu.cn, hkliu@hust.edu.cn, hjin@hust.edu.cn

***Abstract*—Graph random walk is widely used in the graph processing as it is a fundamental component in graph analysis, ranging from vertices ranking to the graph embedding. Different from traditional graph processing workload, random walk features massive processing parallelisms and poor graph data reuse, being limited by low I/O efficiency. Prior designs for random walk mitigate slow I/O operations. However, the state-of-the-art random walk processing systems are bounded by slow disk I/O bandwidth, which is confirmed by our experiments with real-world graphs.**

To address this issue, we propose FlashWalker, an in-storage accelerator for random walk that moves walk updating close to graph data stored in flash memory, by exploiting significant parallelisms inside SSD. Featuring a heterogeneous and parallel processing system, FlashWalker includes a board-level accelerator, channel-level accelerators, and chip-level accelerators. To address challenges posed by the tight resource constraints for processing large-scale graphs, we propose novel designs: storing a few popular subgraphs in accelerators, the pre-walking for dense walks, two optimizations to search the subgraph mapping table, and a subgraph scheduling algorithm. We implement FlashWalker in RTL, showing small circuit area overhead. Our evaluation shows FlashWalker reduces the execution time of random walk algorithms by up to 660.50×, compared with GraphWalker, which is the state-of-the-art system for random walk algorithms.

***Index Terms*—random walk, graph computing, in-storage processing, accelerator**

I. Introduction

Random Walk (RW) [1] is a fundamental graph processing algorithm widely used in the commercial fields, such as Google, Facebook, Alibaba, Tencent, LinkedIn, and Twitter [2]. Sim-Rank [3] computes the similarity of a vertex pair with RW, and random walk domination [4] measures the influence diffusion over the graph with RW. It generates small but representative samples from large-scale graphs because they are not publicly available or hard to analyze [5]. RW is also adopted in graph analytic tasks, such as Personalized PageRank [6], Graphlet Concentration [7], and Network Community Profiling [8]. More recently, the graph representation learning algorithms, such as DeepWalk [9] and Node2Vector [10], use RW to obtain negative nodes, to learn embeddings of nodes in a given graph. These learned node embeddings are used by the downstream machine learning tasks, such as node classification [10], [11], link prediction [10], and graph classification [12]. More importantly, training *graph neural networks* (GNN) also involves the RW to learn the weight parameters, achieving state-of-the-art performance in graph analytics [13].

The common execution pattern of random walk algorithms is that they start walks from massive vertices, and each walk randomly jumps to a neighbor of the vertex the walk landing in until some specific condition is met. The randomness of RW algorithm execution exacerbates the issue of poor data locality of graph processing caused by the irregular graph structures, and becomes a severe performance bottleneck. In response, there are many research efforts to address this issue. However, these prior works [12], [14]–[17] still suffer from low I/O utilization, low walk updating rate, slow disk I/O accesses, as well as high memory cost and energy consumption for managing graph and walks.

The execution pattern adopted by out-of-core graph processing for large-scale graphs features a high level of parallelism in walk updating and suffers from slow and large disk I/O operations. Processing a large-scale graph needs to first split it into subgraphs that can fit in the memory. Moving a walk w accesses the subgraph in which w resides. As a result, the randomness of the algorithm leads to the poor subgraph reuse, generating excessive disk I/O operations. To improve subgraph reuse, GraphWalker [17] proposes a state-aware subgraph scheduling algorithm that prioritizes subgraphs with more walks residing in it. However, our experimental results show that GraphWalker is bounded by slow disk I/O operations shown in Figure 1. This is because massive parallel walks may generate a large number of accesses to different subgraphs with poor locality. Transferring these subgraphs are limited by low PCIe bus bandwidth and narrow channel buses inside the SSD.

To address the disk I/O overhead of random walks, we propose the in-storage accelerator for the random walks for large-scale graphs, referred to as FlashWalker. We move walks updating close to graph stored in flash memory, to avoid graph structure data transferring over narrow channel data bus inside SSD. Different from prior in-storage accelerators, the low arithmetic computing intensity of random walks makes it feasible to deploy walk updating units to a large number of low-level storage units, including flash channels and flash chips, with reasonable circuit area overhead, which is

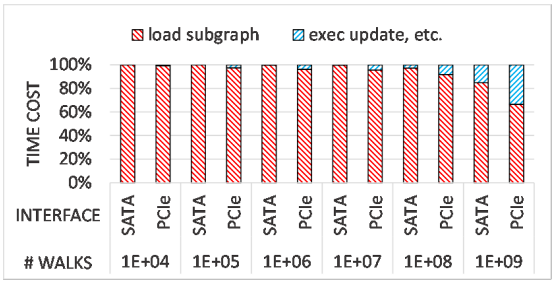


Fig. 1: GraphWalker time cost breakdown on ClueWeb

confirmed by our evaluation. The proposed design not only efficiently exploits the significant I/O parallelisms in SSD but also executes a large number of concurrent walk updating. To this end, we design a board-level accelerator, channel-level accelerators, and chip-level accelerators, and map the random walks to these accelerators, to efficiently utilize the resources in SSD.

There are several challenges to design FlashWalker under the tight resource budget constraints. A large graph is divided into many subgraphs, and the updating walks require their subgraphs to be loaded by accelerators.

First, a board-level accelerator or a channel-level accelerator has much more resources than a chip-level accelerator, but suffers from the data transferring from flash memory arrays. It is challenging to map random walks to board-level and channel-level accelerators to effectively use their resources and minimize data transferring from the underlying flash memory arrays. To solve this issue, we propose to store a few subgraphs with top in-degrees so that most walks can be updated in board-level and channel-level accelerators, as vertices' degrees follow the power-law distribution.

Second, the capacity of subgraph buffer limits the number of edges each subgraph can hold. However, a dense vertex may have a large number of edges, making it impossible to be fully accommodated by the accelerator. To solve this problem, we distribute a dense vertex's outgoing edges into several subgraphs so that each one of them can be loaded by the accelerator. We propose the pre-walking that only loads required graph structural information, rather than all these subgraphs, and selects one of the subgraphs before choosing the next hop for a walk.

Third, walks are organized by their destination subgraphs so that walks are buffered in a buffer entry associated with a subgraph. Determining the destination subgraph for a walk requires looking up the subgraph mapping table, which involves excessive accesses to the mapping table, leading to a performance bottleneck. To solve this issue, we proposes two methods. The first one is we perform the subgraph range query in channel-level accelerators to significantly reduce walk queries in board-level accelerator. The second method designs several small walk query caches to further reduce accesses to the mapping table.

Fourth, walk buffer entries that overflow are written to flash memory, leading to low throughput. To reduce flash memory writes caused by the buffer overflow, we propose a subgraph scheduling algorithm that prioritizes subgraphs that more likely induce walk buffer overflow.

To summarize, the main contributions of our work are as follows.

- We propose an in-storage accelerator for graph random algorithms for large-scale graphs, to address graph data movement overhead caused by low I/O bandwidth of the underlying SSD. The proposed in-storage accelerator moves the walk updating close to graph data, avoiding data transmission over narrow data bus. Furthermore, it can efficiently exploit the untapped large I/O parallelism inside SSD, to boost walk updating rate.
- To address constraints of tight resource budget in SSD, we propose to store a few popular subgraphs in board-level and channel-level accelerators, achieving high walk updating rate.
- To update walks in dense vertices, we propose the pre-walking for these walks, without loading the complete subgraphs for a dense vertex.
- To reduce the overhead of walk queries, we propose the approximate walk query that reduces search operations for a query, by limiting the search range. To further reduce the overhead, we design the walk query cache.
- We propose a novel subgraph scheduling algorithm to reduce flash memory write operations caused by the walk buffer overflow. In addition, we design an efficient method to implement the proposed scheduling algorithm with less overhead.
- We implement FlashWalker in RTL and evaluate it using a cycle-level microarchitectural simulator. We use both large scale real-world graphs and synthetic graphs to evaluate our design. Experiment shows that FlashWalker achieves 51.56× speedup on average over the state-of-the-art random walk engine GraphWalker.

II. Background and Motivation

In this section, we first introduce the graph random walk algorithm, describe challenges faced by existing graph analysis systems when executing random walk algorithms, and analyzes the limitations of random walk specific systems. Finally, we introduce the architecture of modern SSDs, and discuss their potentials to accelerate random walk algorithms.

A. Graph Random Walk

The general framework of random walk algorithm for graphs is as follows. In the initialization phase, it selects a number of vertices in the graph as starting points, and initiates random walks from these selected vertices. During the execution, each walk randomly jumps to a neighbor of the vertex that the walk lands in, based on the neighbor-sampling probability distribution specified by both the graph and algorithm. This process is repeated until a walk reaches the termination condition.

The difference among variant random walk algorithms mainly lies in the neighbor-sampling probability distribution

and the walk termination condition. For weighted graphs, the edges' weights may determine the neighbor-sampling probability distribution. For example, the vertices pointed to by edges with greater weights are more likely to be selected as the destination, which is referred to as the biased random walk algorithm. The algorithm is *unbiased* if the next hop of a walk is uniformly sampled from its neighbors. A random walk algorithm is *static* if its neighbor-sampling probability distribution remains constant throughout the execution, while a random walk algorithm is *dynamic* if its neighbor-sampling probability distribution depends on the state of a walk. The walk termination condition could be: 1) a walk terminates after it has completed a specified number of hops; or 2) a walk terminates according to some probability.

B. Drawbacks of Existing Systems

The out-of-core graph processing systems [18] divide a large graph into a number of partitions so that a partition can fit in the memory. After a partition is processed, the graph processing system proceeds to the next partition. Since most graph analytic algorithm are iterative, the next iteration can be started after all partitions of the current iteration are finished. This is because the next iteration depends on the latest values updated in the current iteration.

They suffer from extremely low I/O utilization and walk updating rate, when performing random walks. The iteration-wise synchronization forces updated walks to be written back to disks before walks are completed, incurring significant slow disk operations. Moreover, the iteration-wise synchronization prevents finished partitions of current iteration from being initiated. However, the inherent asynchronous nature of random walks can allow walks to be independently updated, without requiring the iteration-wise synchronizations.

To improve the performance of random walk, GraphWalker [17] proposes the asynchronous walk updating scheme, by taking advantage of asynchronous nature of random walks. Instead of updating walks in the loaded blocks only once and then putting them back to disk, it keeps updating them until they leave these blocks or have reached the termination conditions. In addition, it gives preference to blocks with a higher number of walks inside to load into the memory. In this way, a higher I/O efficiency and walk updating rate is realized. However, GraphWalker still fails to solve the performance problem caused by the slow I/O bandwidth, which is especially obvious for large-scale graphs. As shown in Figure 1, time spent on loading graph structure data still accounts for the majority of total execution time. This inspires us to implement an architecture inside the SSD to shorten the data-path and thus mitigate the impact of I/O operations on the overall performance.

C. Modern SSD Architecture

The architecture of an SSD [19]–[24] consists of four main parts: a SSD controller, an on-board DRAM, flash channel controllers, and flash chips mounted on each flash channel. The SSD controller has two modules: the *host-interface logic* (HIL), and the *flash translation layer* (FTL). HIL decodes and executes I/O commands (SATA or NVMe) from the external host device. FTL manages resource of the entire SSD: 1) the dynamic allocation of physical address space to support the out-place-update, 2) caching of stored data with the on-board DRAM, 3) the logical-to-physical address mapping, 4) the transaction scheduling, 5) the garbage collection, and 6) the wear-leveling.

TABLE I: SSD architectural characteristics

SSD Organization	32 Channels, 4 Chips per Channel
Flash Channel	ONFI 3.1 (NV-DDR2), Width: 8 bit, Rate: 333 MB/s
Flash Microarchitecture	4KB Page, 4 Planes per Die, 2 Dies per Chip
Flash Access Parameters	Read Latency: 35 μs, Program Latency: 350 μs

Flash channel controllers execute commands to read data from, write, and erase data on flash chips, with the *Open NAND Flash Interface* (ONFI) protocol. A flash chip includes multiple dies, and each die contains multiple planes. A plane consists of a number of flash blocks, and an SRAM-based page register. A flash block contains a number of flash pages, which is a basic unit for reading and writing flash memory.

As shown in Table I, SSD has significant I/O parallelisms and has great potential to achieve high data transmission rate. For example, the SSD has 32 channels, and further each channel has four chips with two dies, which consists of four planes. SSD could deliver huge data transmission bandwidth if all planes work in parallel. However, the narrow bandwidth of a channel bus and a PCIe bus is a roadblock to realize such large bandwidth. First, the flash channel of ONFI 3.1 NV-DDR2 only supports up to 333MB/s, while the aggregation bandwidth of all planes in this channel reaches 1786 MB/s. Second, the bandwidth of four PCIe lanes is 4GB/s, while the aggregate bandwidth of 32 channels is 57.1GB/s. This observation motivates us to design an in-storage accelerator that takes advantage of the internal parallelism to unleash high bandwidth of SSDs.

III. FlashWalker Design

A. System Overview

1) Design Philosophy: Excessive slow disk I/O operations and the low arithmetic intensity of graph random walks make it fit to in-storage computing. Due to poor locality, RW requires a large amount of data transferring from the slow SSD. The narrow data buses, including SSD channel buses and a PCIe bus, become data transmission bottleneck, making it challenging to efficiently exploit parallelism of many flash memory devices. In addition, the updating walks of RW only involve low arithmetic operations and can be executed on customized hardware units with low area/power overhead.

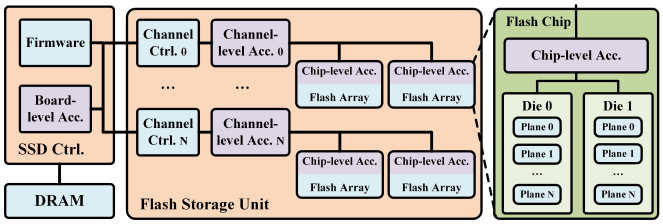


Fig. 2: Overall design of FlashWalker

These observations motivate us to pursue the in-storage accelerating RW algorithms for large-scale graphs. To reduce data transmissions, we move walk-update close to flash memory devices as much as possible, avoiding data transmissions over narrow data buses. Performing walk-update near flash memory naturally exploits SSD's inherent parallelisms offered by its many flash memory devices.

2) Overall Architecture: Figure 2 shows the architecture of FlashWalker which includes chip-level accelerators, channel-level accelerators, and a board-level accelerator. These accelerators can fetch, and update walks with loaded subgraphs in them. To update walks, each accelerator loads corresponding subgraphs to its subgraph buffer. After fetching walks from flash planes and on-board DRAM, a chip-level accelerator updates walks in its walk buffer if the accelerator has loaded the subgraph in which these walks land. When going beyond loaded subgraphs in the accelerator, an uploaded walk is buffered for the further processing and this walk is referred to as a roving walk. Similar to a chip-level accelerator, a channel-level accelerator updates walks after fetching roving walks from the chip-level accelerators connected to the channel. Besides fetching roving walks from channel accelerators and updating walks, a board-level accelerator writes overflow walks, completed walks, and foreigner walks to flash memory.

Walks can be updated if the accelerator has loaded subgraphs where they land. To facilitate accesses to walks, walks in the same subgraph are buffered in a walk queue entry. After updating a walk, the accelerator moves the walk to its destination buffer. The walk-updating requires the accelerator to provide as many walk queue entries as the number of loaded subgraphs in an accelerator. However, constrained resource in the accelerators necessitates small size subgraph and limited number of subgraphs, and makes it impossible to accommodate all subgraphs and walks at the same time. Therefore, we divide a graph into graph partitions and limit the number of subgraphs in each partition. FlashWalker proceeds to the next graph partition when there is no walk in the current partition. In addition, the board-level accelerator provides the partition walk buffer to keep walks in subgraphs in the current graph partition.

B. Chip-level Accelerator

Figure 3 shows a chip-level accelerator that fetches subgraphs and their walks, updates walks, and processes updated walks. After receiving a loading subgraph command issued by the board-level accelerator, a chip-level accelerator reads the

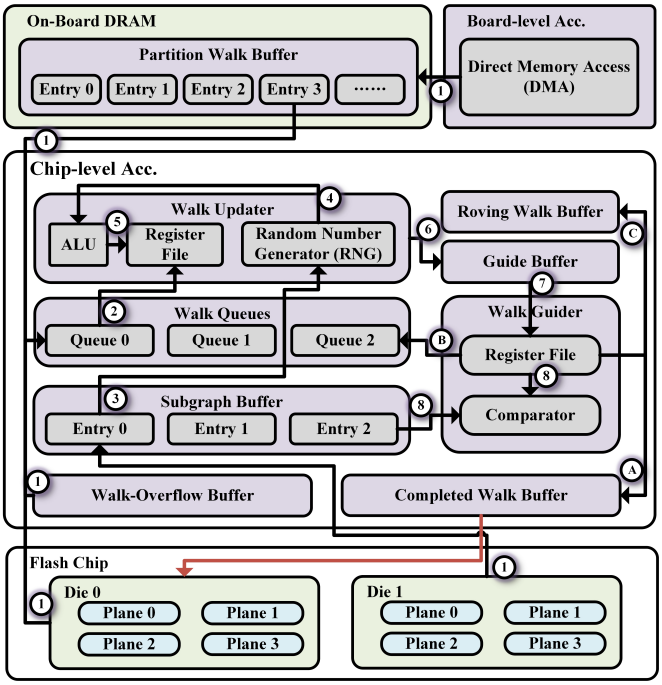


Fig. 3: Chip-level design of FlashWalker

subgraph from flash planes in this chip, and collects its walks from partition walk buffer in the on-board DRAM and from the flash planes. The subgraph are then stored in a subgraph buffer entry and the walks are put in its associated walk queue entry (step ①). A subgraph is stored in CSR format, which contains an offsets array and an edges array. A walk, w, state includes the ID of its source vertex, the offset of the current vertex in the subgraph, and the number of hops, indicated by $w.src$, $w.cur$, and $w.hop$, respectively.

The walk updater first fetches a walk w from a walk queue and stores it in a register (step ②). In unbiased random walk, updating a walk can be formulated as randomly choosing an out-going edge of a given vertex. The updater instructs the random number generator to produce a random number $rnd0$ (step ③), and calculates $outDegree$ (step ④) at the same time, where $outDegree$ is the out-degree of $w.cur$. The $rnd0$ is fed to the ALU to produce the integer random number, $rnd1$, between 0 and $outDegree-1$. The updater fetches the vertex ID of w's next stop from the subgraph's edges array using $rnd1$ as the offset relative to the edge list of $w.cur$ (step ⑤), and changes $w.cur$ to be the fetched vertex's full ID (step ⑥). In addition, $w.hop$ is decremented. After updating w, the walk updater moves it to the guide buffer (step ⑦). The walk updater then fetches the next walk from the walk queue if the queue is not empty. Otherwise, it enters the idle state.

FlashWalker supports biased random walk by adopting Inverse Transform Sampling (*ITS*) technique [12]. To this end, an offsets array of a subgraph stores the sum of weights of out-edges for a vertex and it is denoted as $sumWeight$. To choose an out-edge for a vertex v, the updater first generates a random number rnd in [0, $v.sumWeight$) and finds the

largest index idx of the cumulative distribution list CL, which is the notation of pre-computed function for a vertex, so that rnd is smaller than $CL[idx]$. The biased random walk requires more storage space for CL and more cycles for the binary search.

After updating a walk, the updater moves an updated walk to the guide buffer to determine its destination subgraph. To this end, the walk guider reads a walk from the guide buffer to its register and finds out whether the walk is in a loaded subgraph in this chip-level accelerator by comparing $w.cur$ with two end vertices of each loaded subgraph. If w is in a loaded subgraph, then it is moved to the corresponding walk queue. Otherwise, w is referred to as a roving walk, and is moved to the roving walk buffer. The channel-level accelerator checks and fetches these roving walks in a fixed time interval before stalling the chip-level accelerator's execution.

When a walk is completed, an updater moves it to the completed walk buffer. If the completed walk buffer is full, the completed walks in it are written to flash memory.

C. Channel-level Accelerator

A channel-level accelerator operates similar to a chip-level accelerator. However, there are two significant differences. First, a channel-level accelerator stores K subgraphs whose in-degree are top K among subgraphs stored in flash chips connected to the channel. In this way, the channel-level accelerator can effectively update more walks by keeping a few hot subgraphs. This is because real-world graphs exhibit power-law degree distribution [25]. Second, it accepts and forwards commands/data from the board-level accelerator to a chip-level accelerator, as there is no direct connection between them. These commands are implemented by extending the ONFI commands, which are transferred over a channel bus. Its guider checks each walk in the guide buffer to see if a walk lands in a subgraph in the channel-level accelerator. If it does, the walk is moved to the walk queue for that subgraph. Otherwise, the walk is migrated to the roving walk buffer.

Approximate Walk Search: Although some walks landing in a few hot subgraphs stored in a channel-level accelerator can be updated, there are considerable roving walks that are processed by the board-level accelerator to determine their destination subgraphs, which requires looking up the subgraph mapping table. For a walk query, FlashWalker executes a binary search on the mapping table, which may generate many accesses, forming a performance bottleneck. To address this issue, a channel-level accelerator performs an approximate search for a walk query. This approximate search only returns the range of subgraphs that a given walk lands in. A range of subgraphs is a subgraph set whose vertices' IDs are consecutive and it can be denoted as the vertices range [lowEndID, highEndID]. After this query, these walks are tagged with the ID of the range. To serve these tagged walks, a board-level accelerator searches the corresponding range of subgraph mapping table, reducing the overhead.

To support the approximate walk search, the channel-level accelerator has the subgraph range mapping table where each entry stores the ID of the low-end vertex and the high-end vertex in the corresponding range of subgraphs. If a subgraph range has 256 subgraphs, the subgraph range mapping table can be reduced by 256×, leading to small number of table entries. Searching this small mapping does not introduce a performance overhead. In addition, the subgraph range mapping table can also decide whether a walk is in the current graph partition. If a walk is beyond the current graph partition, it is called a foreigner and can not be processed by the board-level accelerator. Therefore, a guider moves the foreigner to the foreigner buffer. If the foreigner buffer is full, walks in it are flushed to flash memory.

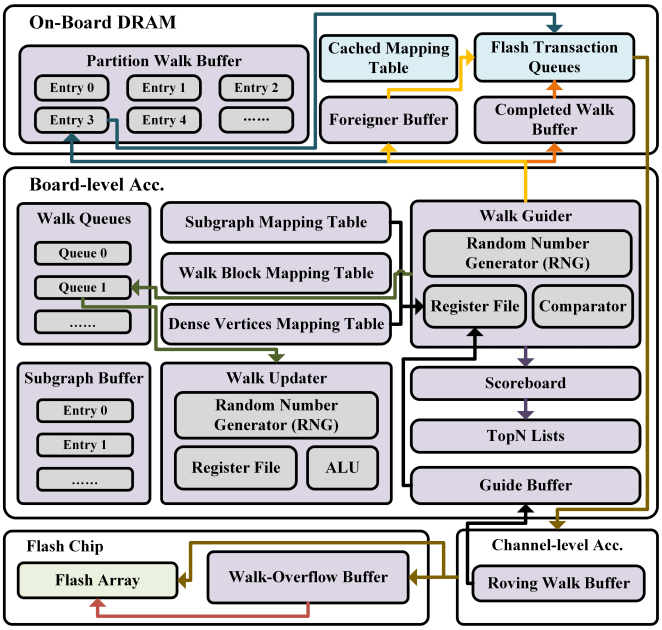


Fig. 4: Board-level design of FlashWalker

D. Board-level Accelerator

Figure 4 shows the architecture of the board-level accelerator. It performs the four major tasks: directing roving walks to the corresponding walk buffer, updating walks landing in subgraphs stored in the subgraph buffer, scheduling subgraphs for chip-level accelerators, and writing walks to flash memory. Similar to channel-level accelerators, board-level accelerator also keeps several hot subgraphs in the subgraph buffer. After fetching roving walks from the guide buffer, the walk guider directs them to walk queues, partition walk buffer, or foreigner buffer. This directing decision needs the subgraph mapping table and the dense vertices mapping table, which map a *vertex ID* (vID) to the corresponding *subgraph ID* (sgID) or IDs. The walk updater grabs a walk from the walk queue and updates it with the help of subgraphs stored in the subgraph buffer. The board-level accelerator schedules a subgraph to a chip-level accelerator whenever there is an empty space in its subgraph buffer. The board-level accelerator also writes completed walks, overflow walks, and foreigner walks to flash memory.

Subgraph Mapping Table: Before updating a walk w, the accelerator needs to determine its subgraph containing the vertex $w.cur$, as the choosing the next stop for w has to access the edges of $w.cur$ stored in the subgraph. A subgraph stores its vertices and their out-edges in a flash memory block with the fixed size and the flash memory block is referred to as a graph block. Therefore, a subgraph contains varied number of vertices since it has different number of out-edges. To determine a subgraph for a vertex, we set up the subgraph mapping table whose entry has: two end vertices in the subgraph, a flash memory address for the subgraph, and the sum of out-degree of the subgraph. We can determine the subgraph's ID for a vertex, by comparing the vertex's ID with two end vertices in the subgraph mapping entries. To reduce the search latency, we perform the binary search for the subgraph mapping table whose entries are sorted with the ID of the low-end vertex in a subgraph.

Walk Query Cache: An updater performs the binary search on the subgraph mapping table. Although the binary search can significantly reduce the number of searches, a walk query still requires many searches to complete. In addition, the mapping table access contentions, caused by multiple walk guiders, further worsen the access latency. To reduce the latency, we propose the walk query cache that stores a very small frequently accessed subgraph mapping entries. There are two reasons the walk query cache works. First, a binary search always touches common nodes in the upper level of the binary search tree, and therefore these nodes exhibit strong temporal locality. Second, vertices in graphs follow the power-law distribution, and hence there are significant number of walks that walk through a very few hot subgraphs. Accordingly, a guider accesses the subgraph mapping table upon query cache miss, significantly reducing access traffic to the subgraph mapping table.

Partition Walk Buffer: A updated walk could jump to a subgraph which is unavailable in an accelerator of FlashWalker since the number of loaded subgraphs is small for a large-scale graph. The partition walk buffer is designed to store walks whose subgraph is absent in the accelerator, and these walks are updated after their subgraphs are loaded by the accelerators. Walks landing in the same subgraph are stored in the same partition walk buffer entry. . Since the walk buffer storage overhead is proportional to the number of subgraphs, the large number of subgraphs makes it impossible to store walk buffer entries for all subgraphs in the on-board DRAM. Accordingly, only the fixed number of subgraphs' walk buffer entries are in the accelerator. To this end, we divide a graph into graph partitions, each of which consists of the same number of subgraphs, except for the last partition. We associate one entry of the partition walk buffer with one entry in the subgraph mapping table so that only the required subgraph mapping entries are stored in the accelerator, reducing the space overhead of the table. Because some walks could go beyond a graph partition, the subgraph mapping table can not determine subgraphs for these walks, referred to as foreigner walks, so we set up the foreigner buffer to store these walks.

Pre-walking for a Dense Vertex: Since a vertex degree follows the power-law distribution in many graphs, especially for social network graphs, a small number of vertices have a very large number of out-edges so that these edges can not fit in a graph block. In this case, such a vertex, referred to as a dense vertex, is accommodated by multiple graph blocks. For example, a dense vertex has 1,213,787 out-edges and requires 19 graph blocks in the Twitter graph. We call a walk w a dense walk if $w.cur$ is a dense vertex. As discussed earlier, updating a walk w needs the subgraph containing the vertex $w.cur$, which means that if w is a dense walk, all the graph blocks where $w.cur$ lands are required to be loaded. However this is impossible due to the resource constraints. To address this issue, we propose the pre-walking technique for the dense walks. Its main idea is that we choose the graph block gb_{next} in which w's next stop dst_{next} lands before determining dst_{next}. Deciding dst_{next} is similar to choosing the next stop for a walk, and this decision requires the metadata about all graph blocks of $w.cur$. In the case of the unbiased random walk, we first get a random number rnd in [0, $w.cur.outDegree-1$] and the gb_{next} is the $\lceil rnd/size(gb) \rceil th$ graph block of $w.cur$, where $size(gb)$ is the number of edges in each graph block. After gb_{next} is available, the guider moves w to the walk buffer entry associated with gb_{next}.

We design the dense vertices mapping table to determine the graph block for a dense walk. The table consists of a bloom filter and a hash table. The bloom filter checks the membership of dense vertices, while the hash table returns the dense vertex metadata for a dense vertex query. The metadata of a dense vertex dv describes graph blocks of dv's subgraph dsg, and it contains: the amount of graph blocks for dsg, the ID of the first graph block of dsg, and the out-degree of its last graph block of dsg. The bloom filter could output a false positive response for a non-dense vertex. Such a false positive response makes the hash table fail to find the graph block list for this vertex. Hence, the proposed dense vertices mapping can work correctly. The walk guider looks up the dense vertices mapping table before the subgraph mapping table. This serial looking up two tables does not introduce performance overhead due to the bloom filter and a smaller number of dense vertices.

Writing Walks to Flash Memory: The board-level accelerator manages different types of buffers, including the partition-walk buffer, the completed walk buffer, and the foreigner buffer. The accelerator flushes walks in the completed walk buffer and the foreigner buffer if they overflow. When a partition-walk buffer entry is used up, the accelerator move this entry to the walk-overflow buffer in the chip-level accelerator, which is then flushed to the flash memory.

Subgraph Scheduling: Unlike GraphWalker, FlashWalker aims to minimize the flash memory writes caused by walk buffer overflow. The overflow of a walk buffer entry makes GraphWalker immediately flush buffered walks to flash memory, leading to large number of flash memory write operations. To reduce flash memory write operations, we propose a novel subgraph scheduling algorithm that reduces occurrences of walk buffer overflow, by prioritizing more critical subgraphs

over less critical subgraphs.

The critical degree of a subgraph indicates how possible these walks in the subgraph are flushed to flash memory and the priority of subgraph scheduling to achieve high throughput, by favoring subgraphs with more number walks [17]. Base on the above analysis, the critical degree for subgraph i is defined as $score_i$ in Eq. 1, where $sg_i.pwb$ and $sg_i.fls$ are the number of walks in the partition-walk buffer and the number walks in the flash memory for the subgraph i, respectively. The parameter α indicate that walks in the partition walk buffer are more critical than walks in the flash memory. This is because the former is written to flash memory if the entry overflows, leading to low walk update rate, while the latter does not have slow flash memory operations.

The β is introduced to consider difference between dense walks/subgraphs and non-dense walks/subgraphs. A dense/non-dense subgraph includes/excludes a dense vertex. We can store more dense walks in a buffer entry or a flash memory page, without storing cur for these walks. The storage of dense walks has two conflicting implications: 1) A non-dense walk buffer entry is more susceptible to overflow than a dense walk buffer entry if they have the same number of walks, and 2) Reading walks stored in flash for a non-dense subgraph could take longer time than for a dense subgraph if they have the same number of walks. The appropriate β can balance between the overhead of buffer overflow and the benefit of high walk updating rate. Each subgraph in the current graph partition has a score entry in the scoreboard backed by eDRAM.

$$score_i = \begin{cases} (sg_i.pwb \times \alpha + sg_i.fls) \times \beta & nondense \\ sg_i.pwb \times \alpha + sg_i.fls & dense \end{cases} \quad (1)$$

When a walk queue for a loaded subgraph becomes empty in a chip-level accelerator, the subgraph scheduler in the board-level accelerator is informed to decide a subgraph and issue a subgraph loading command to the chip-level accelerator. To reduce subgraph transmission overhead caused by narrow channel bus, FlashWalker restricts that subgraphs fetched by a chip-level accelerator must be in the same chip's flash planes. Since there are a large number of subgraphs, retrieving information and sorting $score$ for them need many cycles, leading to a low performance. To avoid this sorting overhead, we maintain $score$ for top N subgraphs for each chip, referred to as the $topN$ list, and the changed $score$ of a subgraph could update the $topN$ list. N is a design parameter. Frequent inserting walks to the partition walk buffer triggers the modification of $score$ of a subgraph, and causes accesses to the $topN$ list, greatly degrading the performance. However, we find that the inserting one walk to the partition walk buffer does not change the value of $score$ much. Therefore, we access the $topN$ list every M walk-insertions for a subgraph, to avoid the performance overhead.

IV. Evaluation

In this section, to validate the performance benefits brought by FlashWalker, we choose GraphWalker [17], the state-of-the-art random walk specific graph system, as the baseline for comparison, and use a range of different graph datasets and random walk hyper-parameters for experiments.

TABLE II: FlashWalker accelerators configurations

Module	Chip-level	Channel-level	Board-level
Frequency	500MHz	500MHz	1GHz
# Updaters	1	1	4
Updater Cycle	16ns	8ns	4ns
# Guiders	1	4	128
Guider Cycle	16ns	8ns	4ns
Subgraph Buffer Cap	1MB	2MB	16MB
Walk Queues Cap	64KB	128KB	1MB
Guide Buffer Cap	-	16KB	128KB
Roving Walk Buf Cap	32KB	8KB	-
Area (mm^2)	1.30	1.84	14.31

TABLE III: FlashWalker SSD & DRAM configurations

SSD Configurations		DRAM Configurations	
PCIe Bandwidth	1GB/s×4	Protocol	DDR4
Host Interface Type	NVMe	Frequency	1600MHz
# Chans, Chips, Dies, Planes	32, 4, 2, 4	Capacity	4GB
# Blocks, Pages	2048, 64	# Channels	1
Page Capacity	4KB	Chip Width	16bit
Flash Comm Protocol	NV-DDR2	Bus Width	64bit
Channel Transfer Rate	333MT/s	BL	8
Flash Technology	MLC	tCL	22
Flash Read Latency	35μs	tRCD	22
Flash Program Latency	350μs	tRP	22
Flash Erase Latency	2ms	tRAS	52

A. Experimental Setup

We implement FlashWalker PEs in RTL with Chisel [26] and synthesize it using Yosys [27] on the 45ns FreePDK45 standard cell library [28], with a target of 1 GHz frequency. The scratchpad memory and SRAM overhead in our design are estimated using Destiny2 [29] and Cacti [30]. We evaluate FlashWalker using a cycle-level microarchitectural simulator, which includes MQSim [20] and DRAMSim3 [31] to model SSD and DRAM respectively.

The configurations of each level accelerators, SSD, and DRAM for FlashWalker in our experiments are summarized in Table II and Table III. The updater cycle and guider cycle indicate the time interval between each two operations performed by walk updater and guider respectively. The walk updater performs 5 operations to process a walk if not stalled, while the guider performs a different number of operations for each walk according to the number of queries issued.

For walk query, we set board-level subgraph mapping table capacity to 2MB, walk blocks mapping table capacity to 128KB, and dense vertices mapping table capacity to 128KB. We also provide 32 walk query caches in total for board-level walk guiders, and every 4 walk guiders share a single walk query cache. Each walk query cache has a capacity of 4KB. In the absence of additional explanations, α and β in Equ. 1 are set to 1.2 and 1.5, respectively. We run the baseline GraphWalker [17] on the computer that has a AMD Ryzen 7 3700X 8-Core @ 3.60GHz processor, 32GB DDR4 memory, and a 2TB Samsung 970 EVO Plus SSD with

TABLE IV: Statistics of datasets

Dataset	$\|V\|$	$\|E\|$	CSR Size	Text Size
Twitter (TT)	41.6M	1.46B	5.8GB	23GB
Friendster (FS)	65.6M	3.61B	14GB	59GB
ClueWeb (CW)	4.78B	7.94B	95GB	138GB
RMAT2B (R2B)	62.5M	2B	8GB	32GB
RMAT8B (R8B)	250M	8B	32GB	137GB

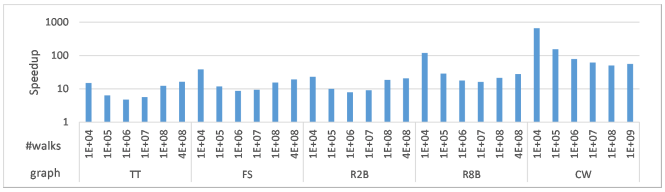


Fig. 5: FlashWalker speedup with different number of walks

PCIe3.0×4 interface. Since simulating large-scale graphs takes much longer time, our evaluations use medium-scale graphs to project FlashWalker's performance for large-scale graphs, by artificially setting the memory capacity used by GraphWalker to be 8GB by default. The statistics of GraphWalker execution is averaged over 10 repetitions, and the operating system cache is cleared before starting the next repetition, to exclude caching effects of the same input graph.

Table IV lists the graph datasets [32] used in our experiments, where TT [33], FS [34], and CW [35] are real-world graphs, while R2B and R8B are synthesized using PaRMAT [36]. The vertex IDs of TT, FS, R2B, and R8B are represented using 4 bytes, while the vertex IDs of CW are represented using 8 bytes as the total number of its vertices exceeds the 4-byte representation range. Similarly, for TT, FS, R2B, and R8B, we set the subgraph size to 256KB, while for CW, we set the subgraph size to 512KB. The walk length is fixed as 6 in all experiments.

The graph partitioning time overhead in the pre-processing phase is O(#edges), which is similar to GraphWalker. Our measured execution time excludes the time required by graph partitioning, as partitioned graphs are standard inputs to many different graph processing tasks, and it is also excluded by the evaluation of many typical graph processing systems [12], [14], [17], [37].

B. Speedup over GraphWalker

Figure 5 shows the FlashWalker speedup over GraphWalker with varied number of walks. FlashWalker achieves 4.79× to 660.50× (51.56× on average) speedup over GraphWalker shown in Figure 5. Notice that the average speedup ratio for larger graphs is higher than smaller graphs. This is mainly because GraphWalker's memory can accommodate almost the entire small graphs like TT and R2B, and it only needs to load the graph data through the PCIe lanes once at the beginning. In such cases, the advantages brought by FlashWalker mainly lie in the much higher bandwidth utilization than PCIe lanes and the parallelism provided by 128 flash chips.

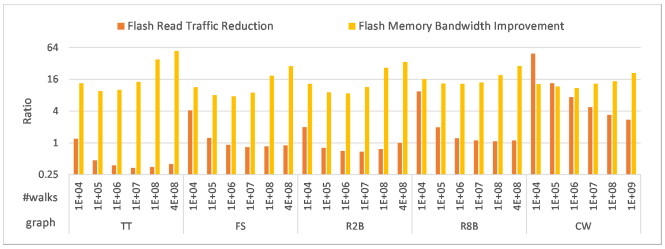


Fig. 6: Flash memory read traffic reduction and bandwidth improvement

To confirm the reasons for the speedup of FlashWalker, Figure 6 shows its improvement of achieved flash memory bandwidth and its flash memory read traffic reduction. Due to exploiting large I/O parallelism of flash planes and the finer granularity of subgraphs, FlashWalker achieves 17.21× flash memory bandwidth improvement and 3.82× read traffic reduction over GraphWalker on average, for all tasks. In addition, for tasks where the number of walks is set as 10^9 for CW and 4×10^8 for the remaining graphs, FlashWalker achieves 33.44× flash memory bandwidth improvement and 1.23× read traffic reduction on average.

In particular, when processing TT, FlashWalker reads a higher total amount of data than GraphWalker. This is due to the parallelism overload caused by the small size of TT. Chip-level accelerators always manage to stay busy to avoid idleness, and thus frequently load new subgraph from flash, even if the new subgraph only has a few walks in it. In contrast, GraphWalker keeps most of the graph data in memory, and only one subgraph is chosen to process at a time, thus reducing the amount of data read. However, since the average bandwidth utilization of FlashWalker is always much higher, it offsets the loss caused by more reads and allows the overall performance to remain higher than GraphWalker.

When processing CW, the total amount of data read by FlashWalker is much less than GraphWalker. There are two main reasons for this. First, the graph data significantly exceeds the memory capacity of GraphWalker, resulting in frequent subgraph swapping in and out. Second, FlashWalker has smaller subgraph granularity, making it unnecessary to read an entire big subgraph (1GB in GraphWalker) for only a few walks, thus improving the utilization of graph data, which is also noted as I/O efficiency.

C. FlashWalker Performance Projection

In this subsection, we study the FlashWalker performance under different scales of graphs with the same set of workloads, by configuring the memory capacity used by GraphWalker to be 4GB, 8GB, and 16GB and fixing the memory capacity of FlashWalker. For the same graph, GraphWalker execution with smaller memory leads to more block I/O operations and emulates RW behavior for a larger-scale graph. Therefore, FlashWalker speedup over the GraphWalker with 4GB and 16GB memory can project performance of graphs larger and smaller than 8GB, respectively, shown in Figure 7.

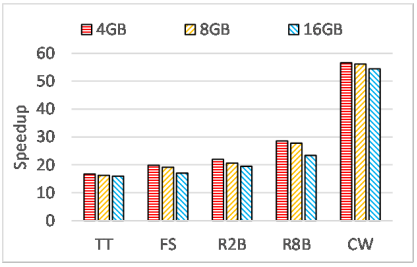


Fig. 7: FlashWalker speedup over GraphWalker with varied DRAM capacities

The number of walks are set as 10^9 for CW and 4×10^8 for the remaining graphs. As shown in Figure 7, FlashWalker's speedup does not drop significantly when the memory capacity is increased to 16GB. For TT, as GraphWalker is already able to accommodate the entire graph when using 8GB memory, the larger buffer does not bring much additional benefit. For CW, as the graph size still far exceeds the memory capacity, large amount of SSD data reads are still inevitable. For remaining graphs, the performance advantages of FlashWalker overshadows GraphWalker's performance gains from the reduced SSD data reads due to its high bandwidth utilization, parallelism, and I/O efficiency.

D. Resource Consumption Behavior of FlashWalker

Figure 8 shows the FlashWalker resource consumption behavior in the execution process. These resources include flash memory read bandwidth, flash memory write bandwidth, the SSD channel bandwidth. In addition, we also show the achieved overall bandwidth and the progression of RW execution, which is the percentage of the finished walks.

We make the following observation for FlashWalker execution. First, the channel bandwidth is saturated in the most time for TT, FS, and R8B as the figure shows it close to 10.4GB/s, which is the theoretically maximal aggregated channel BW. Secondly, flash memory read bandwidth are not fully used for TT, FS and R8B during a long period of time, since their flash memory read BW is below 55.8GB/s, which is the theoretically maximal aggregated chip read throughput. Thirdly, there are very small flash memory write bandwidth consumption, which is consistent with the RW algorithm. Lastly, CW execution is bound by the straggler processing because it finishes about 90% walks in the first 2 seconds and spends 5.5 seconds to handle 10% stragglers.

The low flash memory read BW is due to the high density of walks in the early stages, which prevents chip-level accelerators from fetching new subgraphs frequently, while a large amount of uploaded roving walks cause the channel bus to become a bottleneck. As the number of remaining walks drops, the bottleneck goes back to flash, thus leading to the rapid increase of the flash memory read BW. The low flash memory write bandwidth, on the other hand, represents a low write-back frequency, which proves the effectiveness of FlashWalker's subgraph scheduling. The observation about CW is discussed in section IV-E.

E. Speedup of Proposed Optimizations

Figure 9 evaluates speedup for three proposed optimizations over the baseline FlashWalker, which has no any optimizations. These three optimizations are: 1) the *walk query* (WQ) that applies approximate walk search to channel-level accelerators and adopts the walk query cache to the board-level accelerator; 2) the *hot subgraphs* (HS) that stores hot subgraphs in channel-level accelerators and the board-level accelerator; and 3) the *subgraph scheduling* (SS) that schedules subgraphs according to their scores calculated with Equ. 1, where α is 0.4 to reduce the burden on the channel bus according to the observation in Section IV-D and β is 1.5. These optimizations are incrementally enabled and are applied to prior enabled optimization(s).

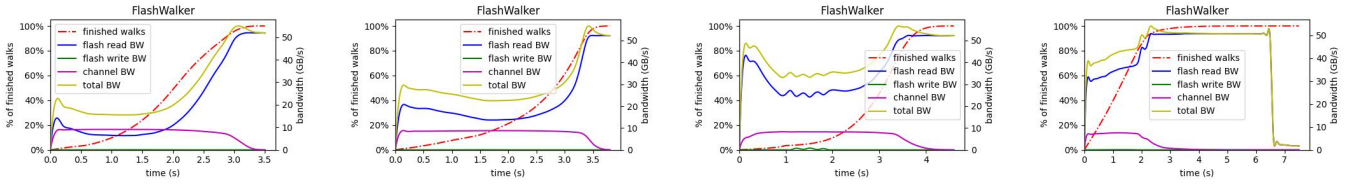
WQ successfully reduces the walk query latency and hence greatly improves the performance of FS, R2B, and R8B by 18.39%, 16.68%, and 13.80% respectively. However, TT does not benefit much from it (only 5.02%), due to the skewness of TT's edge distribution which allows some of the subgraphs to gather a large number of walks. A single chip-level accelerator does not have a particularly strong capability to quickly update so many walks in a few subgraphs, and thus cannot provide enough roving walks for the channel-level and board-level accelerators. Therefore, the main bottleneck of the system does not lie in the walk query, but in the walk update. Although HS has little effect on FS, R2B, and R8B, on the other hand, it significantly improves the performance of TT (20.76%, together with WQ) as it allows the board-level and channel-level accelerators to update the walks landing in hot subgraphs with greater efficiency, thus sharing the burden of the chip-level accelerators. Lastly, SC further improves the performance of TT, FS, R2B, and R8B by 21.53%, 21.26%, 18.83%, and 18.31% over the baseline.

However, these optimizations improve the performance of CW marginally. This is because the number of walks is relative small compared with the graph size. As shown in Figure 8(d), FlashWalker takes most of the time to process the stragglers, and the bottleneck to process these stragglers is caused by the slow flash read latency, which can not be mitigated by these optimizations. If CW has more walks, the processing of stragglers becomes a small portion of its execution time, and these optimizations are expected to be effective to process those non-stragglers, achieving noticeable performance improvement. However, we do not evaluate CW with more walks, since it requires longer simulation time.

V. Related Work

A number of software graph processing frameworks have been proposed in recent years [18], [38], [39], to simplify the programming for graph analytics. These graph processing frameworks are based on many different programming models, including vertex-centric paradigms, sparse matrix operations, task-based models, etc., with the most popular being the vertex-centric paradigm.

By integrating high-bandwidth memory, GPU-based graph processing solutions can achieve many orders of magnitude



(a) TT (4×10^8 walks) (b) FS (4×10^8 walks) (c) R8B (4×10^8 walks) (d) CW (10^9 walks)

Fig. 8: Resource consumption behavior of FlashWalker

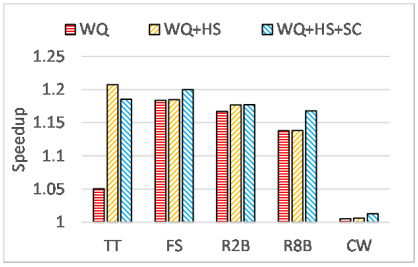


Fig. 9: FlashWalker speedup under different optimizations

performance improvements over CPU solutions [40]–[42]. These systems are built by integrating a number of graph analysis software frameworks and libraries optimized specifically for GPUs. However, most of these frameworks require a costly offline or online preprocessing to transform irregular data layout into regular one, and such preprocessing often dominates the entire execution process, e.g., the preprocessing time of Gunrock [40] reaches twice the processing time.

To address the slow data movement issue, *processing-in-memory* (PIM) has been applied to the graph computing. There are categories of PIM for graph processing: 1) *processing near memory* (PnM) with the logical layer of 3D stacked memory [43]–[46], and 2) *processing using memory* (PuM) by adopting a scheme based on ComputeDRAM [47], [48], and 3) in-situ computing by exploiting the properties of ReRAM [49]–[52]. However, these PIM proposals suffer from the limited storage capacity of the memory. For example, the current 3D stacked memory is smaller than 64GB, and the commodity DRAM capacity is smaller than 100 GB in a commodity server. The capacity-limited PIM fails to handle more larger data sets.

As for domain-specific accelerators, Graphicionado [53] adopts a pipelined architecture, and introduces a large capacity on-chip eDRAM to improve the locality of vertex property updates. However, its update process still suffers from irregularities. To address this issue, GraphPulse [37] adopts an event-driven model and proposes the coalesce operation, making serialization unnecessary.

There are some prior works on random walk specific graph systems. DrunkardMob [14] adopts a lightweight scheduling strategy and follows GraphChi's [18] iteration-based model to implement a random walk system on a single PC. GraphWalker [17] optimizes the I/O management mechanism and proposes the asynchronous updating scheme, improving the performance significantly. KnightKing [12] designs a distributed graph random walk system, and optimizes the walk updates for complex random walk algorithms.

VI. Conclusion

Graph random walk algorithms are important and basic to graph analytic. However, the state-of-the-art random walk processing systems are bounded by slow disk I/O bandwidth. To address this issue, we propose FlashWalker, an in-storage accelerator for random walk that moves walk updating close to graph data stored in flash memory, by exploiting significant parallelisms inside SSD. Featuring a heterogeneous and parallel processing system, FlashWalker includes a board-level accelerator, channel-level accelerators, chip-level accelerators, with small circuit overhead. To address challenges posed by the tight resource constraints for processing large scale graph, we propose novel designs: storing a few popular subgraphs in accelerators, the pre-walking for dense walks, two optimizations for search for the subgraph mapping table, and a subgraph scheduling algorithm. Our evaluation shows FlashWalker reduces the execution time of random walk algorithms by up to 660.50×, compared with the state-of-the-art system for random walk algorithms.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported jointly by National Natural Science Foundation of China (NSFC) under grants No. 61825202, 62072198, 61732010, and USA NSF 1745748.

References

[1] Z. Bar-Yossef, A. C. Berg, S. Chien, J. Fakcharoenphol, and D. Weitz, "Approximating aggregate queries about web pages via random walks," in *Proceedings of VLDB*, 2000, pp. 535–544.

[2] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," in *Proceedings of SOSP*, 2021, pp. 311–326.

[3] G. Jeh and J. Widom, "SimRank: a measure of structural-context similarity," in *Proceedings of SIGKDD*, 2002, pp. 538–543.

[4] R. Li, J. X. Yu, X. Huang, and H. Cheng, "Random-walk domination in large graphs," in *Proceedings of ICDE*, 2014, pp. 736–747.

[5] P. Yi, H. Xie, Y. Li, and J. C. S. Lui, "A bootstrapping approach to optimize random walk based statistical estimation over graphs," in *Proceedings of ICDE*, 2021, pp. 900–911.

[6] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Math.*, vol. 2, no. 3, pp. 333–358, 2005.

[7] N. Przulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 26, no. 6, pp. 853–854, 2010.

[8] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *CoRR*, vol. abs/1608.00163, 2016.

[9] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: online learning of social representations," in *Proceedings of SIGKDD*, 2014, pp. 701–710.

[10] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of SIGKDD*, 2016, pp. 855–864.

[11] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *IEEE Data Engineering Bulletin*, vol. 40, no. 3, pp. 52–74, 2017.

[12] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "KnightKing: a fast distributed graph random walk engine," in *Proceedings of SOSP*, 2019, pp. 524–537.

[13] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using GPUs," in *Proceedings of EuroSys*, 2021, pp. 311–326.

[14] A. Kyrola, "DrunkardMob: billions of random walks on just a PC," in *Proceedings of RecSys*, 2013, pp. 257–264.

[15] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-SAW: a framework for graph sampling and random walk on GPUs," in *Proceedings of SC*, 2020, pp. 56:1–56:15.

[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] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. S. Lui, "GraphWalker: an I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks," in *Proceedings of ATC*, 2020, pp. 559–571.

[18] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: large-scale graph computation on just a PC," in *Proceedings of OSDI*, 2012, pp. 31–46.

[19] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proceedings of ATC*, 2008, pp. 57–70.

[20] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices," in *Proceedings of FAST*, 2018, pp. 49–66.

[21] M. Jung, J. Zhang, A. H. M. O. Abulila, M. Kwon, N. Shahidi, J. Shalf, N. S. Kim, and M. T. Kandemir, "SimpleSSD: modeling solid state drives for holistic system simulation," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 37–41, 2018.

[22] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. D. Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W. Hwu, "DeepStore: in-storage acceleration for intelligent queries," in *Proceedings of MICRO*, 2019, pp. 224–238.

[23] K. K. Matam, G. Koo, H. Zha, H. Tseng, and M. Annavaram, "GraphSSD: graph semantics aware SSD," in *Proceedings of ISCA*, 2019, pp. 116–128.

[24] S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. S. Rosing, "THRIFTY: training with hyperdimensional computing across flash hierarchy," in *Proceedings of ICCAD*, 2020, pp. 27:1–27:9.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: distributed graph-parallel computation on natural graphs," in *Proceedings of OSDI*, 2012, pp. 17–30.

[26] "Chisel/FIRRTL hardware compiler framework." [Online]. Available: https://www.chisel-lang.org

[27] C.Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[28] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: an open-source variation-aware design kit," in *Proceedings of MSE*, 2007, pp. 173–174.

[29] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "DESTINY: a tool for modeling emerging 3D NVM and eDRAM caches," in *Proceedings of DATE*, 2015, pp. 1543–1546.

[30] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[31] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. L. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[32] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of AAAI*, 2015, pp. 4292–4293.

[33] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," in *Proceedings of ICWSM*, 2010, pp. 10–17.

[34] Friendster social network, "Friendster: the online gaming social network," https://archive.org/details/friendster-dataset-201107.

[35] C. L. A. Clarke, N. Craswell, and I. Soboroff, "Overview of the TREC 2009 web track," in *Proceedings of TREC*, 2009.

[36] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proceedings of PACT*, 2015, pp. 39–50.

[37] S. Rahman, N. B. Abu-Ghazaleh, and R. Gupta, "GraphPulse: an event-driven hardware accelerator for asynchronous graph processing," in *Proceedings of MICRO*, 2020, pp. 908–921.

[38] N. Sundaram, N. Satish, M. M. A. Patwary, S. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: high performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[39] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: edge-centric graph processing using streaming partitions," in *Proceedings of SOSP*, 2013, pp. 472–488.

[40] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of PPoPP*, 2016, pp. 11:1–11:12.

[41] Z. Fu, B. B. Thompson, and M. Personick, "MapGraph: A high level API for fast development of high performance graph analytics on GPUs," in *Proceedings of GRADES*, 2014, pp. 2:1–2:6.

[42] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on GPUs," in *Proceedings of SC*, 2015, pp. 68:1–68:12.

[43] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of ISCA*, 2015, pp. 105–117.

[44] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "GraphH: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, 2019.

[45] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: reducing communication for PIM-based graph processing with efficient data partition," in *Proceedings of HPCA*, 2018, pp. 544–557.

[46] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: enabling instruction-level PIM offloading in graph computing frameworks," in *Proceedings of HPCA*, 2017, pp. 457–468.

[47] F. Gao, G. Tziantzioulis, and D. Wentzlaff, "ComputeDRAM: in-memory compute using off-the-shelf DRAMs," in *Proceedings of MICRO*, 2019, pp. 100–113.

[48] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. Gómez-Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. D. Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: set-centric instruction set architecture for graph mining on processing-in-memory systems," in *Proceedings of MICRO*, 2021, pp. 282–297.

[49] G. Dai, T. Huang, Y. Wang, H. Yang, and J. Wawrzynek, "GraphSAR: a sparsity-aware processing-in-memory architecture for large-scale graph processing on ReRAMs," in *Proceedings of ASPDAC*, 2019, pp. 120–126.

[50] L. Song, Y. Zhuo, X. Qian, H. H. Li, and Y. Chen, "GraphR: accelerating graph processing using ReRAM," in *Proceedings of HPCA*, 2018, pp. 531–543.

[51] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, "GRAM: graph processing in a ReRAM-based computational memory," in *Proceedings of ASPDAC*, 2019, pp. 591–596.

[52] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan, "GaaS-X: graph analytics accelerator supporting sparse data representation using crossbar architectures," in *Proceedings of ISCA*, 2020, pp. 433–445.

[53] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of MICRO*, 2016, pp. 56:1–56:13.