SaGraph: A Similarity-aware Hardware Accelerator for Temporal Graph Processing

Jin Zhao[†§], Yu Zhang[†§], Jian Cheng[†], Yiyang Wu[†], Chuyue Ye[†], Hui Yu[†], Zhiying Huang[†], Hai Jin[†], Xiaofei Liao[†], Lin Gu[†], Haikun Liu[†]

[†] *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan, China*

[§] *Zhejiang-HUST Joint Research Center for Graph Processing, Zhejiang Lab, Hangzhou, China*

{zjin, zhyu, jaycheng, wuyiyang, yechuyue, huiy, hzying, hjin, xfliao, lingu, hkliu}@hust.edu.cn

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247966

***Abstract*—Temporal graph processing is used to handle the snapshots of the temporal graph, which concerns changes in graph over time. Although several software/hardware solutions have been designed for efficient temporal graph processing, they still suffer from serious irregular data access due to the *uncoordinated graph traversal*. To overcome these limitations, this paper proposes *SaGraph*, a domain-specific hardware accelerator to support the efficient processing of temporal graph. Specifically, temporal graph processing shows strong *data access similarity*, i.e., most graph accesses of the processing of different snapshots are the same and usually refer to a small fraction of vertices. SaGraph can dynamically coordinate the graph traversals and adaptively cache the vertex states to fully exploit the data access similarity for smaller data access overhead. We implemented and evaluated SaGraph on a Xilinx Alveo U280 FPGA card. Compared with the cutting-edge software and hardware solutions, SaGraph achieves 8.5×-157.3×, 4.2×-16.1× speedups and 34.7×-423.6×, 5.3×-14.7× energy savings, respectively.**

I. Introduction

Real world graphs are constantly evolving over time, and there is growing interest in mining the time-evolving characteristics of these graphs, e.g., analyzing distance changes among entities [11] and studying diameter variation of social network [12]. These applications usually perform specific graph algorithm on multiple snapshots of the temporal graph to capture how metrics evolve over time, where each snapshot corresponds to a time point of the temporal graph. This scenario is called *temporal graph processing*, where the algorithm executed on a particular snapshot is called an *instance*.

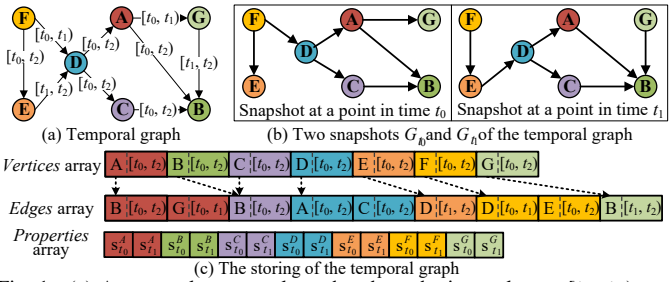
Although some software temporal graph processing systems [3], [9], [17], [21] and hardware solutions [5], [8], [18], [20] have been recently proposed, these solutions suffer from serious irregular memory accesses to the graph data associated with the intersection of these snapshots. Specifically, due to different temporal property (e.g., different lifespans of edges and vertices) of different snapshots, the processing of these snapshots exhibits uncoordinated graph traversal to the same graph structure data of these snapshots, which incurs the following two problems. First, different instances repeatedly load the same graph structure data at different times, incurring redundant off-chip communications. Second, different instances randomly access their states associated with the same vertices at different times, thus the most portion of the fetched vertex states in the on-chip memory are not required.

Challenges come with opportunities. Through analyzing the data access characteristics of the temporal graph processing, we have two findings. First, most graph data accessed by different instances are the same and these graph data accesses are performed along the graph topology, which shows the *temporal similarity*. It indicates that many graph traversals of different instances can be coordinated along the graph topology to amortize the same graph data accesses. Second, most vertex state accesses of different instances refer to a small fraction of vertices, which shows the *spatial similarity*. This motivates us to resident these vertices' states in the on-chip memory for fewer off-chip communications.

Based on the above observations, in this paper, we propose a similarity-aware hardware accelerator *SaGraph*, which can support the efficient processing of temporal graph. Specifically, SaGraph features the specialized hardware pipelines to efficiently coordinate the graph traversals of different instances. It dynamically extracts the topological order of the active vertices for these instances on the fly and fetches the graph data associated with these active vertices based on the extracted order to drive the related instances to synchronously process these fetched graph data together. By such means, the same graph accesses can be shared by more instances, ensuring fewer off-chip communication. In addition, SaGraph specializes the memory subsystem to preferentially cache the frequently accessed vertices' states in the on-chip memory for multiple instances, achieving better data locality.

We have implemented and evaluated SaGraph on a Xilinx Alveo U280 FPGA card. The results show SaGraph outperforms the cutting-edge software solutions running on Intel Xeon CPU and NVIDIA A100 GPU by 18.7×-157.3× and 8.5×-26.4× with 56.2×-423.6× and 34.7×-176.1× energy savings, respectively. Compared to the cutting-edge graph processing accelerators, i.e., ReGraph [5] and ScalaGraph [20], SaGraph achieves 7.8×-16.1× and 4.2×-9.7× speedups with 6.5×-14.7× and 5.3×-13.1× energy savings, respectively.

This paper is supported by National Key Research and Development Program of China under grant No. 2022YFB2404202, NSFC (No. 62072193), Huawei Technologies Co., Ltd (No. YBN2021035018A5), and the Young Top-notch Talent Cultivation Program of Hubei Province. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.



(a) Temporal graph (b) Two snapshots G_{t_0} and G_{t_1} of the temporal graph (c) The storing of the temporal graph

Fig. 1. (a) An example temporal graph, where the interval, e.g., $[t_0, t_2)$, on each edge indicates the time period (i.e., ranging from the point of time t_0 to the point of time t_1) that this edge exists; (b) two snapshots of the temporal graph in the points of time t_0 and t_1; (c) the storing of the temporal graph

II. Background And Motivation

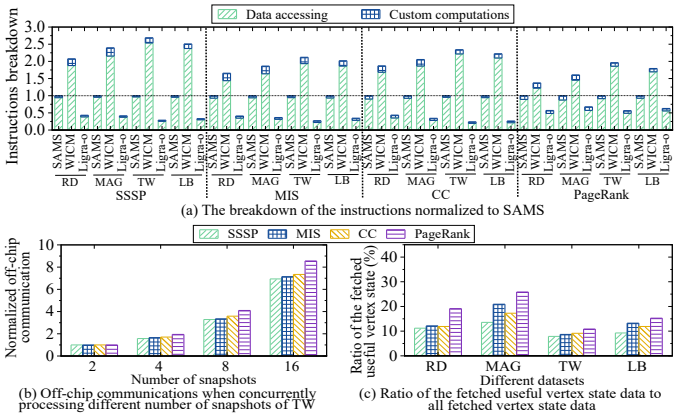
A. Temporal Graph Processing

Temporal Graph. *Temporal graph* is a type of graph evolving over time (e.g., vertex/edge addition/deletion), where each vertex and edge contain their existence information (i.e., their lifespans) [3], [7], [9], [17]. At a point in time t, a temporal graph G_T is expressed as a snapshot G_t=(V_t, E_t), where V_t and E_t represent the sets of vertices and edges existed at time t, respectively. Fig. 1 shows a temporal graph and it has two snapshots, where the interval on each vertex/edge represents the time period (i.e., the lifespan) that this vertex/edge exists.

Temporal Graph Representation and Processing. To store the temporal graph, existing solutions [3], [7], [17] usually employ the most popular *Compressed Sparse Row* (CSR) format [15], [16] and also hold the lifespans for the vertices and edges. As shown in Fig. 1(c), it employs two types of arrays, i.e., *Vertices* and *Edges*, to encode the graph structure data of the temporal graph in Fig. 1(a). The *Edges* array stores each outgoing edge's destination vertex ID and lifespan. The *Vertices* array maintains the beginning and end offsets of the edges in the *Edges* array and the lifespan for each vertex. The *Properties* array holds each vertex' states for all instances, where the states associated with the same vertex are stored successively for the instances. To process a temporal graph G_T=$\{G_{t_0}, ..., G_{t_{n-1}}\}$, a set of instances I=$\{i_{t_0}, ..., i_{t_{n-1}}\}$ will be generated to perform a user-given graph algorithm on this temporal graph's snapshots [3], [7], [9], [17], where i_t represents the algorithm running on a particular snapshot G_t.

B. Challenges of Temporal Graph Processing

Many solutions [3], [7], [9], [17] have been proposed to handle the temporal graph. However, temporal graph processing exhibits serious irregular access to the same graph data of multiple snapshots. This unique challenge of temporal graph processing incurs significant data access overhead in existing solutions. To demonstrate it, we measured the performance of three cutting-edge software temporal graph processing systems (i.e., SAMS [17], WICM [3], and Ligra-o) running on Intel Xeon CPU, where Ligra-o (detailed in Section IV-A) is the version of Ligra [16] optimized to support temporal graph processing. Section IV also presents the details of the platform and benchmarks. Fig. 2(a) shows that, although Ligra-o outperforms others under all circumstances, most executed instructions of Ligra-o (more than 80.5%) are responsible for



(a) The breakdown of the instructions normalized to SAMS

(b) Off-chip communications when concurrently processing different number of snapshots of TW (c) Ratio of the fetched useful vertex state data to all fetched vertex state data

Fig. 2. Performance of existing solutions over different graph algorithms

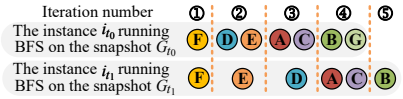


Fig. 3. Execution of multiple instances on a temporal graph

data accessing (e.g., traversing the graph and loading the vertex states), whereas only a small ratio of instructions are executed for custom computations which process the loaded data. We illustrate the reasons for the above inefficiency as follows.

Specifically, using existing solutions [3], [7], [9], [17], the same vertices may be accessed by different instances at different times. Fig. 3 gives an example, which shows the execution of *Breadth-First Search* (BFS) algorithm [4], [17] rooting from the vertex Ⓕ on the snapshots G_{t_0} and G_{t_1} of Fig. 1(b). We can observe that, when the instance i_{t_0} has individually accessed the graph structure data associated with Ⓓ at its second iteration, the instance i_{t_1} needs to access the same data associated with Ⓓ at its third iteration. Thus, such *uncoordinated graph traversal* in temporal graph processing may incur significant redundant off-chip communications (e.g., the accesses of the graph structure data associated with Ⓓ). Fig. 2(b) shows that the amount of off-chip communications significantly increases when Ligra-o concurrently processes more snapshots, although most graph structure data are the same for different snapshots [9], [17]. It is because that, when serving more instances, more copies of the same graph structure data have to be loaded repeatedly at different times.

Besides, using existing solutions, temporal graph processing suffers from poor data locality. First, due to the uncoordinated graph traversal, different instances individually access their states associated with the same vertices at different times, incurring poor spatial locality. Second, the vertices with longer lifespan usually need to be accessed by more instances. However, the states of the vertices with long lifespan suffer from cache thrashing. It results in poor temporal locality. Fig. 2(c) shows, in Ligra-o, more than 74.1% of vertex states loaded into the cache are not used before being swapped out.

C. Limitation of State-of-the-Art Solutions

1) Inefficiencies of General-Purpose Processors: The serious irregular data access of temporal graph processing incur poor data locality, which makes it ill-suited to current data prefetching techniques and cache hierarchy design on CPUs. GPUs also suffer from low efficiency when serving temporal

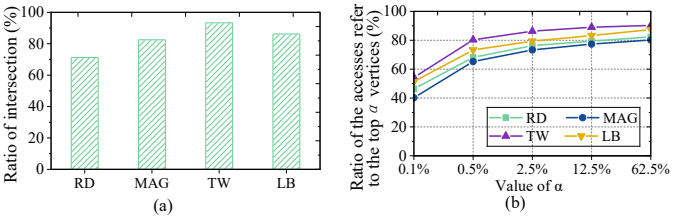


Fig. 4. Studies of the data access characteristics of temporal graph processing on Ligra-o: (a) the average ratio of the intersection of different snapshots to each snapshot on average; (b) the ratio of multiple instances' vertex state accesses refer to the top α vertices to those refer to all vertices

graph processing which exhibits irregular memory accesses, because the GPU is designed for compute-intensive workloads with regular execution pattern [13].

2) Inefficiencies of Conventional Accelerators: Compared with general-purpose processors, specialized hardware accelerators [5], [8], [18]–[20] gain significant energy saving and performance speedup for graph processing. However, they are inefficient for temporal graph processing because they can only alleviate the irregularity in the processing of a single snapshot and suffer from significant redundant off-chip communications to the same graph data of multiple snapshots (see Fig. 7&8). It calls for costuming a new accelerator for the unique challenge of irregular data access in temporal graph processing.

D. Motivation

1) Opportunities for Customization: Because most portion of the graph structure data is the same for different snapshots of temporal graph [9], [17], there is strong spatial and temporal similarities among different instances. It provides the opportunities to overcome the limitations of existing architectures and improve the performance of temporal graph processing.

Opportunity one: Most graph data accessed by different instances are the same and these graph data accesses are conducted along the graph topology, which shows the temporal similarity. When handling the active vertices of different instances along their topological order, the graph traversals of these instances can be coordinated to fully exploit this temporal similarity. Fig. 4(a) shows that the ratio of the intersection of different snapshots to a snapshot is more than 71.3% on average, which suggests that different instances can share most accesses to graph structure data. Taken Fig. 3 as an example, at the second iteration, we can handle the active vertices of the instances i_{t_0} and i_{t_1} along their topological order, i.e., Ⓔ and Ⓓ. In detail, when Ⓔ is first processed by both i_{t_0} and i_{t_1}, Ⓓ will be activated for i_{t_1}. Then, both i_{t_0} and i_{t_1} can share the accesses of the graph structure data associated with Ⓓ. Besides, for i_{t_0} and i_{t_1}, their states (which are stored successively) associated with Ⓓ (i.e., $s^D_{t_0}$ and $s^D_{t_1}$) will also be accessed by them simultaneously, ensuring better data locality. It motivates us to dynamically coordinate different instances' graph traversals along the graph topology, aiming to exploit the temporal similarity for fewer off-chip communications.

Opportunity two: Most vertex state accesses performed by different instances refer to a small fraction of vertices, which exhibits the spatial similarity. Specifically, when a vertex owns longer lifespan and higher degree, more instances usually need to access it and more vertices of an instance have to propagate their states through it. Thus, this vertex' states are usually accessed more frequently under such circumstances. To demonstrate it, we give each vertex v a priority Pri_v=LS_v×D_v, where LS_v and D_v are v's lifespan and degree, respectively. Then, we sort the vertices in descending order based on their priorities. Fig. 4(b) evaluates the ratio of multiple instances' accesses referring to the states of the top α vertices to those of all vertices in the temporal graph processing. The results show that more than 65.2% of the state accesses come from accessing the top 0.5% vertices. Inspired by this, we can preferentially store these vertices' states in the on-chip memory for multiple instances to exploit the spatial similarity of temporal graph processing for better data locality.

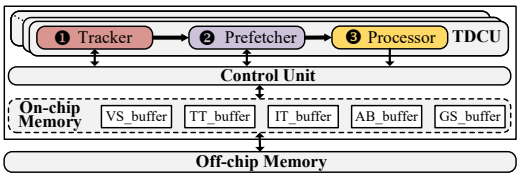


Fig. 5. Architecture of SaGraph

2) Challenges: Exploiting the similarities among different instances of temporal graph processing remains challenging. This is because it needs to dynamically track the topological order of the active vertices for all instances through irregularly traversing the temporal graph. Besides, it produces additional instructions and also induces low instruction-level parallelism because these instructions involve data-dependent branches, e.g., depending on irregular graph structures. To address these challenges, we propose an efficient accelerator *SaGraph* with specialized hardware designs for temporal graph processing.

III. SaGraph Architecture

A. SaGraph Overview

Fig. 5 shows the architecture of SaGraph. The *topology-driven coordinating unit* (TDCU) is used to dynamically coordinate the graph traversals of different instances and then prefetch/process the graph data according to the coordinated graph traversals. The on-chip memory is employed to cache different types of data to reduce off-chip communications. Their main functionalities are as follows.

TDCU contains three key parts. During the execution, the ***Tracker*** starts from active vertices of the instances to explore the graph so as to track the topological order of all instances' active vertices on the fly (step ❶). Along the tracked topological order, the ***Prefetcher*** takes each active vertex of the instances to prefetch the edges of this vertex, these edges' lifespans, and the source/destination vertices' states of these edges (step ❷). For each prefetched edge, the ***Processor*** triggers the instances which need to process this edge to concurrently handle this edge using user-defined primitives (step ❸). In this way, multiple instances can fully share the common graph accesses by efficiently exploiting the temporal similarity among them.

The on-chip memory is composed of several buffers, which are employed to isolate the accesses of different types of data (e.g., vertex state data and graph structure data) and eliminate

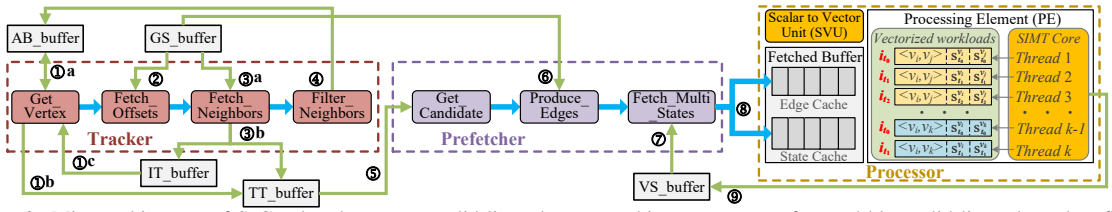


Fig. 6. Microarchitecture of SaGraph, where green solid lines denote on-chip memory transfers and blue solid lines show data flow

the data thrashing among these data. To fully exploit the spatial similarity of the temporal graph processing, it also dynamically caches the frequently accessed vertices' states for multiple instances, avoiding the data thrashing of them.

B. Graph Topology Tracking

To track the topological order of all instances' active vertices, the *Tracker* repeatedly takes an unvisited active vertex of the instances as the root to explore the temporal graph in a breadth-first order on the fly, until all active vertices have been visited. To efficiently perform the topology tracking, as shown in Fig. 6, the *Tracker* has four stages, which are conducted as a pipeline. Note that two tables, i.e., *Intermediate_Table* and *Topology_Table*, are created in the off-chip memory to store the intermediate information for topology tracking and the visited vertices, respectively. Meanwhile, a bitvector, i.e., *Active_Bit*, is created in the off-chip memory to record all instances' active vertices, avoiding redundant graph exploration.

During the execution, at the *Get_Vertex* stage, if the *Intermediate_Table* is empty, the *Tracker* scans the *Active_Bit* to obtain an active vertex, then labels this vertex as inactive in *Active_Bit* (step ①a), and inserts this found vertex into the *Topology_Table* (step ①b). Otherwise, the *Tracker* pops a vertex from the *Intermediate_Table* (step ①c). Next, at the *Fetch_Offsets* stage, the beginning/end offsets of the vertex gotten at the *Get_Vertex* stage are fetched from the *Vertices* array (step ②). Then, at the *Fetch_Neighbors* stage, the unvisited neighbors of this vertex are obtained from the *Edges* array (step ③a), and these neighbors are pushed into the *Topology_Table* and *Intermediate_Table* (step ③b). At the *Filter_Neighbors* stage, these obtained neighbors are marked as inactive in *Active_Bit* if they are active in *Active_Bit* (step ④). The above procedures repeat until all active vertices of the instances have been visited. The active vertices' visiting order, which is recorded in the *Topology_Table*, is taken as the topological order of these active vertices approximately.

C. Topology-aware Graph Data Prefetching

To allow the common graph data accesses to be fully amortized by different instances, the *Prefetcher* sequentially retrieves vertex from the *Topology_Table* and prefetches this vertex's graph data to drive the execution of multiple instances.

Specifically, as shown in Fig. 6, the *Prefetcher* performs three stages as a pipeline to efficiently prefetch the graph data. At the *Get_Candidate* stage, the *Prefetcher* gets a vertex from the *Topology_Table* sequentially (step ⑤). Then, at the *Produce_Edges* stage, this vertex's neighbors and their lifespans are obtained from the *Edges* array to produce the edges (step ⑥). At the *Fetch_Multi_States* stage, the source and destination vertices' states of these produced edges are fetched from the *Properties* array accordingly for the instances which need to process these edges (step ⑦). The prefetched edges and the vertex states associated with these edges are pushed into the *Fetched Buffer*, which contains an *Edge Cache* (storing the prefetched edges) and a *State Cache* (storing the prefetched vertex states) (step ⑧). Note that each entry in the *Edge Cache* stores <*Source vertex*, *Destination vertex*, *Lifespan*> and each entry in the *State Cache* stores <*Instance*, *Source vertex state*, *Destination vertex state*>.

D. Data-driven Concurrent Processing

To effectively share the common graph data accesses, for each edge in the *Fetched Buffer*, the *Processor* triggers the instances which need to process this edge to handle this edge together. In this way, each edge needs to be loaded only once to serve multiple instances. Besides, these instances' vertex states associated with this edge are accessed simultaneously, ensuring better data locality.

To enhance the throughput of temporal graph processing, we implement the *Processing Element* (PE) of the *Processor* with the SIMT execution model. Specifically, as shown in Fig. 6, the *Processor* employs a *Scalar to Vector Unit* (SVU) to perform loop unrolling on the processing of multiple instances. The SVU obtains an edge from the *Edge Cache* and the vertex states associated with this edge from the *State Cache*. Then, the SVU dispatches these data to the SIMT threads of the same PE to be handled by the instances which need to handle this edge accordingly. For example, as shown in Fig. 6, the edge $<v_i, v_j>$ and the vertex states of v_i and v_j associated with the instances which need to handle this edge (e.g., i_{t_0}, i_{t_1}, and i_{t_2}) are dispatched to the SIMT threads of the same PE (e.g., the *Thread 1*, *Thread 2*, and *Thread 3*), respectively. Then, the SIMT threads on each PE simultaneously process the workloads dispatched to this PE and update the vertex states of the related instances (step ⑨). Note that, like GraphDynS [18], the number of SIMT threads on each PE is set as eight by default. To improve the efficiency of SIMT, when the number of workloads associated with an edge is smaller than that of the SIMT threads on the same PE, the processing of different edges is conducted on the same PE together as shown in Fig. 6.

E. Similarity-aware Memory Hierarchy

For efficient access of the *Properties* array, *Topology_Table*, *Intermediate_Table*, *Active_Bitvector*, and the graph structure data, as shown in Fig. 5, five on-chip buffers, i.e., *VS_Buffer*, *TT_Buffer*, *IT_Buffer*, *AB_Buffer*, and *GS_Buffer*, are used to cache these data, respectively. However, the states of the vertices with high priority (introduced in Section II-D1) may be evicted from the *VS_Buffer* because existing cache management strategies [2], [6], [10] are unaware of temporal property

TABLE I

GRAPH CHARACTERISTICS ($\overline{LS_V}$ AND $\overline{LS_E}$ ARE THE AVERAGE LIFESPAN OF VERTICES AND EDGES, AND $|S|$ IS THE NUMBER OF SNAPSHOTS)

Datasets	*#Vertices*	*#Edges*	$\overline{LS_V}$	$\overline{LS_E}$	$\|S\|$	Categories
Reddit (RD) [1]	9.1 M	523 M	6.6	1.22	121	Real-world graph
MAG [2]	116 M	1 B	20.9	15.8	219	
Twitter (TW) [3]	43.9 M	2.1 B	29.5	28.4	30	
LDBC-9_0-FB (LB) [3]	12.8 M	1.1 B	69.8	44.7	104	Synthetic graph

[1] http://cs.cornell.edu/jhessel/projectPages/redditHRC.html
[2] www.openacademic.ai/oag [3] twitter.mpi-sws.org

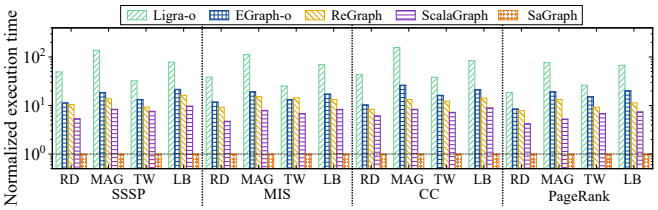


Fig. 7. Normalized execution time of different solutions

(e.g., the vertices' lifespans). The access of the *Properties* array occupies the most proportion of all data accesses. Thus, we propose a *Similarity-Aware Data Caching* (SADC) scheme to efficiently manage the *VS_Buffer*. Note that the other buffers are managed using LRU [10] by default. Specifically, for the management of the *VS_Buffer*, SADC employs the priority of the vertices to determine the victims. When the states of a vertex v are requested, if the *VS_Buffer* is not full, v's states will be directly cached in the *VS_Buffer*. Otherwise, SADC first obtains v's priority, i.e., Pri_v=$LS_v \times D_v$, where v's lifespan LS_v and v's degree D_v (i.e., subtracting v's beginning offset from its end offset) are obtained according to the *Vertices* array. If Pri_v is larger than the lowest priority (i.e., Pri_L) of all vertices whose states are cached in the *VS_Buffer*, SADC evicts the states of the vertex whose priority equals to Pri_L from the *VS_Buffer*, and v's states are then cached in the *VS_Buffer*. This way, SADC efficiently prevents the high priority vertices' states (i.e., the frequently accessed vertices' states) from data thrashing, ensuring better data locality.

IV. EVALUATION

A. Experimental Setup

SaGraph Settings. We have implemented SaGraph on a Xilinx Alveo U280 FPGA card, which is equipped with a XCU280 FPGA chip. The FPGA provides 9 MB BRAM resources, 1.3 M LUTs, 2.6 M Registers, and two 4 GB HBM2 stacks. SaGraph contains eight TDCUs, and we use the BRAM resources to implemente the on-chip memory of SaGraph. We employ Xilinx Vivado 2019.1 to obtain the clock rate of SaGraph and conservatively use 250 MHz in our experiments.

Datasets and Benchmarks. Our experiments consider four popular algorithms, i.e., *Single Source Shortest Path* (SSSP), *Maximal Independent Sets* (MIS), *Connected Components* (CC), and PageRank, performed on three real-world temporal graphs and one synthetic temporal graph (listed in Table I).

Baselines. We first optimize Ligra [16] (which is the best-performing CPU-based shared-memory graph processing system) by incorporating several optimizations (e.g., the temporal graph processing method used in SAMS [17], careful loop unrolling [15], and software prefetching [1]). The optimized version of Ligra is called *Ligra-o*, which outperforms the

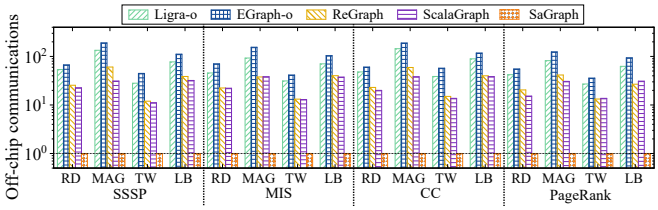


Fig. 8. Normalized off-chip communications of different solutions

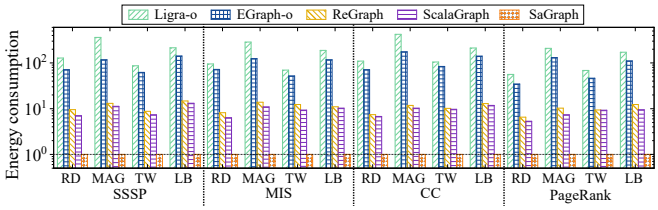


Fig. 9. Normalized energy consumption of existing solutions

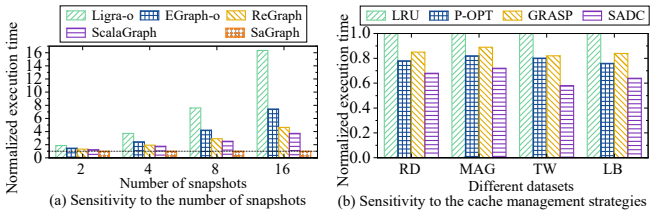
cutting-edge CPU-based temporal graph processing systems, i.e., SAMS [17] and WICM [3], by up to 4.3 and 9.7 times as shown in Fig. 2(a), respectively. Next, Ligra-o is used as the CPU baseline and runs on a server with two 32-core 2.3 GHz Intel Xeon Gold 6314U processors and 128 GB DRAM. We also optimize EGraph [21] (which is the best-performing GPU-based temporal graph processing system) by integrating several optimizations (e.g., efficiently using zero-copy [14] and path-based execution model [22]) to get EGraph-o, which outperforms the original version of EGraph by up to 3.6 times. EGraph-o is used as the GPU baseline and runs on an NVIDIA A100 GPU (80 GB HBM2e with 1.94 TB/s memory bandwidth). Besides, SaGraph is compared with the cutting-edge graph processing accelerators, i.e., ReGraph [5] and ScalaGraph [20], which run on Xilinx Alveo U280 FPGA.

B. Performance of SaGraph

1) Overall Performance: Fig. 7 shows that SaGraph outperforms the other solutions in all cases. Compared with Ligra-o, EGraph-o, ReGraph, and ScalaGraph, SaGraph improves the performance by 18.7×-157.3×, 8.5×-26.4×, 7.8×-16.1×, and 4.2×-9.7×, respectively. The better performance achieved by SaGraph comes from fewer off-chip communications when serving temporal graph processing.

2) Off-chip Communications: Fig. 8 shows that the amount of off-chip communications in SaGraph is only 1.92%, 1.36%, 4.22%, and 4.69% of Ligra-o, EGraph-o, ReGraph, and ScalaGraph on average, respectively. There are two reasons. First, different instances' graph traversals are effectively coordinated. Thus, the common graph data accesses are fully shared by different instances, sparing much redundant off-chip communications. Second, the frequently accessed vertices' states are cached in on-chip memory, ensuring better data locality.

3) Energy Savings: Fig. 9 evaluates the energy efficiency of different solutions. Note that the energy consumptions of SaGraph, ReGraph, and ScalaGraph are measured using xbutil [5]. For Ligra-o, we use CPU Energy Meter [5] to measure its energy. The energy consumption of EGraph-o is estimated using nvidia-smi [5]. Fig. 9 shows that SaGraph reduces the energy consumption of Ligra-o, EGraph-o, ReGraph, and ScalaGraph by 56.2×-423.6×, 34.7×-176.1×, 6.5×-14.7×, and 5.3×-13.1×, respectively.



(a) Sensitivity to the number of snapshots (b) Sensitivity to the cache management strategies

Fig. 10. Sensitivity analysis of SaGraph

C. Sensitivity Analysis

Fig. 10(a) evaluates the performance of different solutions when concurrently running SSSP on different number of snapshots of TW. It shows that SaGraph obtains better performance improvement when there are more concurrently processed snapshots. It is because that SaGraph can effectively coordinate the graph traversals to reduce more redundant off-chip communications when more snapshots are concurrently processed. Fig. 10(b) depicts the performance of SSSP on SaGraph with different strategies (i.e., LRU [10], P-OPT [2], and GRASP [6]) for the management of the *VS_Buffer*. The results show that our SADC strategy outperforms others because the frequently accessed vertices' states are effectively cached.

V. Related Work

Temporal Graph Processing Systems. To efficiently support temporal graph processing, many software systems have been proposed. Chronos [9] employs a locality-aware scheduling method to exploit better data locality. SAMS [17] and WICM [3] try to reduce redundant computations and communications for temporal graph processing. To handle the temporal graph on GPU, EGraph [21] further proposes the *Loading-Processing-Switching* execution model. However, they still suffer from significant redundant data access cost due to the irregular data accesses in the processing of multiple snapshots.

Hardware Graph Processing Accelerators. For energy-efficient graph processing, many accelerators have been designed. Graphicionado [8] is the first graph processing accelerator and can efficiently reduce random data accesses. To alleviate the irregularity in graph processing, GraphDynS [18] uses a hardware/software co-design. For higher convergence rate, GraphABCD [19] is designed to support asynchronous graph processing. ScalaGraph [20] is further proposed to improve the scalability of PEs, while ReGraph [5] is recently designed for higher performance graph processing on FPGA using heterogeneous pipelines. However, these hardware solutions also suffer from serious irregular data access when serving temporal graph processing. Compared with them, SaGraph can effectively coordinate the graph traversals in temporal graph processing for more regular data access and better data locality.

VI. Conclusion

SaGraph is the first temporal graph processing accelerator, which can effectively alleviate the challenge of irregular data access for temporal graph processing. By dynamically coordinating the graph traversals and adaptively caching the frequently accessed vertices' states for the execution of multiple snapshots, SaGraph minimizes the off-chip communications and achieves high energy efficiency. The results show that SaGraph achieves 8.5×-157.3×, 4.2×-16.1× speedups and 34.7×-423.6×, 5.3×-14.7× energy savings compared with the state-of-the-art software and hardware solutions, respectively.

References

[1] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses: A microarchitectural perspective," *ACM Transactions on Computer Systems*, vol. 36, no. 3, pp. 8:1–8:34, 2019.

[2] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical optimal cache replacement for graph analytics," in *Proceedings of HPCA*, 2021, pp. 668–681.

[3] A. Baranawal and Y. Simmhan, "Optimizing the interval-centric distributed computing model for temporal graph algorithms," in *Proceedings of EuroSys*, 2022, pp. 541–558.

[4] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of SC*, 2011, pp. 1–12.

[5] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W. Wong, "ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines," in *Proceedings of MICRO*, 2022, pp. 1–13.

[6] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proceedings of HPCA*, 2020, pp. 234–248.

[7] S. Gandhi and Y. Simmhan, "An interval-centric model for distributed computing over temporal graphs," in *Proceedings of ICDE*, 2020, pp. 1129–1140.

[8] 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. 1–13.

[9] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of EuroSys*, 2014, pp. 1–14.

[10] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of MICRO*, 2013, pp. 284–296.

[11] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin, "Version traveler: Fast and memory-efficient version switching in graph processing systems," in *Proceedings of USENIX ATC*, 2016, pp. 523–536.

[12] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of KDD*, 2005, pp. 177–187.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[14] S. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, and W. Hwu, "EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 114–127, 2020.

[15] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of MICRO*, 2018, pp. 1–14.

[16] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of PPoPP*, 2013, pp. 135–146.

[17] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann, "Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs," *Proceedings of the VLDB Endowment*, vol. 10, no. 8, pp. 877–888, 2017.

[18] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of MICRO*, 2019, pp. 615–628.

[19] Y. Yang, Z. Li, Y. Deng, Z. Liu, S. Yin, S. Wei, and L. Liu, "GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent," in *Proceedings of ISCA*, 2020, pp. 419–432.

[20] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue, "ScalaGraph: A scalable accelerator for massively parallel graph processing," in *Proceedings of HPCA*, 2022, pp. 199–212.

[21] Y. Zhang, Y. Liang, J. Zhao, F. Mao, L. Gu, X. Liao, H. Jin, H. Liu, S. Guo, Y. Zeng, H. Hang, L. Chen, Z. Ji, and W. Biao, "EGraph: Efficient concurrent GPU-based dynamic graph processing," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–12, 2022.

[22] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu, "DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs," in *Proceedings of ASPLOS*, 2019, pp. 601–614.