TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing

Jin Zhao[†], Yun Yang[†], Yu Zhang[†*], Xiaofei Liao[†], Lin Gu[†], Ligang He[§], Bingsheng He[‡], Hai Jin[†], Haikun Liu[†], Xinyu Jiang[†], Hui Yu[†]

[†] 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, China

[§] Department of Computer Science, University of Warwick, United Kingdom

[‡] National University of Singapore, Singapore

{zjin, yunyang, zhyu, xfliao, lingu, hjin, hkliu, xinyujiang, huiy}@hust.edu.cn
ligang.he@warwick.ac.uk hebs@comp.nus.edu.sg

ABSTRACT

Many solutions have been recently proposed to support the processing of streaming graphs. However, for the processing of each graph snapshot of a streaming graph, the new states of the vertices affected by the graph updates are propagated irregularly along the graph topology. Despite the years' research efforts, existing approaches still suffer from the serious problems of *redundant computation overhead* and *irregular memory access*, which severely underutilizes a many-core processor. To address these issues, this paper proposes a topology-driven programmable accelerator *TDGraph*, which is the first accelerator to augment the many-core processors to achieve high performance processing of streaming graphs. Specifically, we propose an efficient topology-driven incremental execution approach into the accelerator design for more regular state propagation and better data locality. TDGraph takes the vertices affected by graph updates as the roots to prefetch other vertices along the graph topology and synchronizes the incremental computations of them on the fly. In this way, most state propagations originated from multiple vertices affected by different graph updates can be conducted together along the graph topology, which help reduce the redundant computations and data access cost. Besides, through the efficient coalescing of the accesses to vertex states, TDGraph further improves the utilization of the cache and memory bandwidth. We have evaluated TDGraph on a simulated 64-core processor. The results show that, the state-of-the-art software system achieves the speedup of 7.1~21.4 times after integrating with TDGraph, while incurring only 0.73% area cost. Compared with four cutting-edge accelerators, i.e., HATS, Minnow, PHI, and DepGraph, TDGraph gains the speedups of 4.6~12.7, 3.2~8.6, 3.8~9.7, and 2.3~6.1 times, respectively.

* Corresponding Author: Yu Zhang (zhyu@hust.edu.cn).

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527409

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; **Special purpose systems**; **Data flow architectures**.

KEYWORDS

streaming graphs, incremental computation, state propagation, accelerator, many-core processor

ACM Reference Format:
Jin Zhao, Yun Yang, Yu Zhang, Xiaofei Liao, Lin Gu, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, Xinyu Jiang, and Hui Yu. 2022. TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527409

1 INTRODUCTION

Real world graphs usually evolve continuously over time, and this type of graph is called streaming graphs [14, 30, 61]. For example, about 6,000 tweets are deleted or added on the twitter graph per second [47]. The continuously arriving updates, e.g., edge deletions or edge additions, are typically applied to the graph in batches [11, 32, 33, 44, 61]. Streaming graph processing is critical in many applications, e.g., anomaly detection [18], targeted advertising [17], financial fraud detection [42], and social network analysis [14]. To obtain timely results for streaming graphs, many software systems [21, 24, 32, 33, 38, 51–53, 59–61] have been developed recently. For efficiency, they use the technique of *incremental computation*, which incrementally refines the results calculated before the graph mutations according to the graph updates. However, for streaming graph processing, these software systems still suffer from significant redundant computations and high data access cost on many-core processors due to the following two reasons.

First, the state propagations originated from the vertices affected by the graph updates are irregularly conducted along the graph topology, which results in the *redundant computation overhead*. Specifically, when handling multiple graph updates (e.g., edge deletions and edge additions) in a batch, the new states of different affected vertices (e.g., the destination vertices of deleted edges or added edges) are propagated individually to their neighbors along different graph paths. Thus, the state propagations of different

affected vertices pass through the same vertices irregularly at different times. The state propagation of each affected vertex may induce the processing of its neighbors iteratively along the graph topology. As the result, the graph data associated with the common neighbors of these affected vertices are loaded repeatedly and processed multiple times.

Second, streaming graph processing suffers from serious *irregular memory access*. In the incremental computation of a streaming graph, usually only a small set of vertices are active and need to be handled, which incurs many random data accesses to their states, because these vertices' states are typically dispersed very sparsely in the main memory. Besides, the vertex state access in the streaming graph processing usually refers to very small data element (e.g., 4 byte per vertex state, which is much smaller than a cache line) each time. As a result, most vertex states fetched into the *Last-Level Cache* (LLC) are actually not needed. This eventually leads to the underutilization of the cache and memory bandwidth.

Although some hardware solutions [9, 19, 36, 37, 64, 67, 73] have been designed to augment the many-core processors for high-performance graph processing, they are designed to accelerate the processing of static graphs. Meanwhile, to support streaming graph processing, some ASIC-based accelerators, e.g., JetStream [44] and DREDGE [35], have been recently proposed. However, they sacrifice the flexibility and programmability of many-core processors. Besides, these accelerators are inefficient for tackling the irregular propagations of the new states of the vertices affected by the graph updates. A new hardware mechanism is much needed to help the existing many-core processors to achieve regular state propagations in streaming graph processing.

We analysed the characteristics of streaming graph processing and made two main observations. First, the state propagations originated from different affected vertices usually pass through a large set of common vertices along the graph topology inherently. It means that most propagations can be conducted together by synchronously processing these common vertices according to the graph topology. Second, most vertex state accesses refer to a small set of vertices due to the power-law property [22, 62]. It provides us the opportunity to efficiently consolidate most accesses to vertex states. Based on these observations, we develop a topology-driven programmable accelerator called *TDGraph*, which augments the many-core processor to achieve high-performance streaming graph processing.

Different from the existing solutions, TDGraph supports an effective *topology-driven incremental execution approach* to achieve more regular state propagation and better data locality in streaming graph processing. Specifically, TDGraph takes some affected vertices as the roots to prefetch other vertices along the graph topology for effective propagations of the affected vertices' states and synchronizes the propagations of these vertices' states on the fly based on the tracked information of the graph topology. In this way, most state propagations can be regularly conducted together, reducing redundant loading and processing of the graph data associated with the common vertices traversed by these propagations. Besides, TDGraph consolidates the vertex states that are accessed frequently into the same cache lines to realize the coalesced accesses of these states, and consequently achieves higher utilization of the memory bandwidth and cache resources.

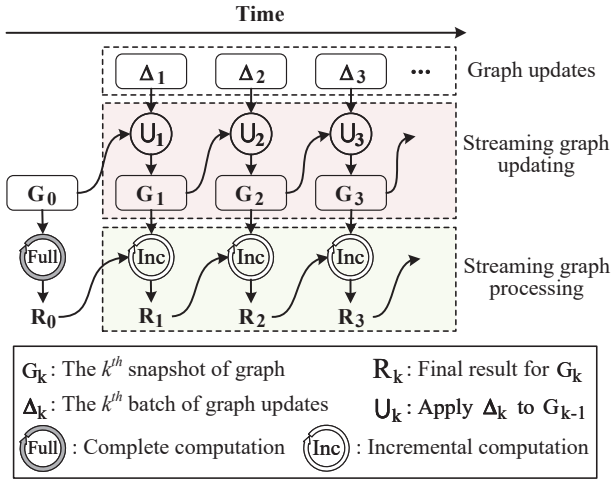


Figure 1: Illustration of streaming graph analytics

To evaluate the effectiveness of TDGraph, we implement it in a simulated 64-core processor and conduct the comprehensive experiments. Compared with the state-of-the-art software streaming graph processing system, TDGraph achieves the speedup of 7.1~21.4 times with only 0.73% area cost. Compared with four cutting-edge hardware accelerators, i.e., HATS [36], Minnow [67], PHI [37], and DepGraph [73], TDGraph improves the performance by 4.6~12.7, 3.2~8.6, 3.8~9.7, and 2.3~6.1 times, respectively.

Our main contributions are summarized as follows:

- We investigate the problems of redundant computation overhead and irregular memory access in streaming graph processing.
- We propose an effective topology-driven incremental execution approach to reduce the redundant computation and improve data locality for streaming graph processing.
- We design an efficient programmable accelerator *TDGraph*, which augments the many-core processor with our proposed topology-driven incremental execution approach for high performance streaming graph processing.
- We prototype TDGraph in a simulated 64-core processor and conduct the extensive experiments to demonstrate its advantages. The results validate the efficiency of TDGraph and show that it can achieve a higher performance improvement than the cutting-edge solutions.

The remainder is organized as follows: Section 2 describes the challenges of existing streaming graph processing solutions and our motivations. Section 3 presents our approach and the details of TDGraph, followed by an evaluation in Section 4. Section 5 gives a survey of the additional related work. Finally, we conclude this paper in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Streaming Graph Analytics

In streaming graph analytics, as shown in Fig. 1, the graph updates (typically consist of edge deletions and additions) constantly arrive and are buffered in batches [32, 33, 61] (such as Δ_2 in Fig. 1). To generate the latest graph snapshot (e.g., G_2), the newly arrived graph

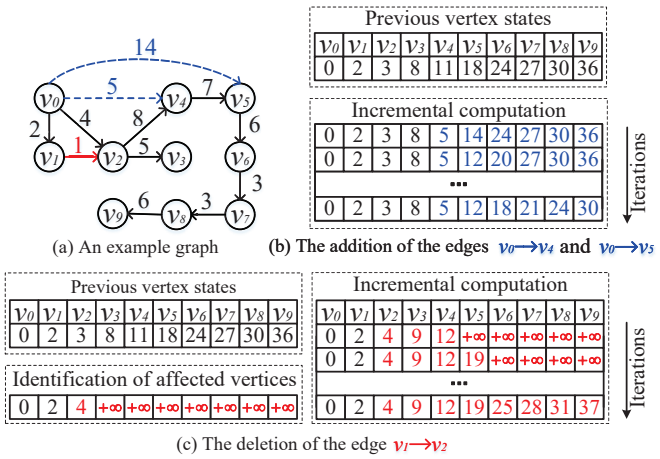


Figure 2: An example to illustrate incremental computation: (a) an exemplar streaming graph, where the blue dashed (or red solid) lines represent edge additions (or deletions); (b) (or (c)) the incremental refinement of the results of the previous graph snapshot in response to the edge addition (or deletion) for the SSSP algorithm

updates (e.g., Δ_2) need to be applied to the previous graph snapshot (e.g., G_1). To obtain new results of the latest graph snapshot (i.e., G_2) in response to the graph updates (i.e., Δ_2), many streaming graph processing systems [32, 33, 61] have been developed recently to process the latest graph snapshot.

Generally, the batch size of graph updates is tiny in comparison with the graph size, and the graph updates only affect a small portion of the vertices [20, 21]. Thus, as shown in Fig. 1, the existing streaming graph processing systems [32, 33, 61] use the *incremental computation* technique to refine the results of the previous graph snapshot (e.g., G_1) to quickly obtain the results for the latest graph snapshot (e.g., G_2). Specifically, the new state of each vertex affected by graph update is propagated to other vertices along the graph topology (i.e., the dependencies between vertices) and used to refine the previously converged states (i.e., the final states in the previous graph snapshot) of these vertices to gain the final states of these vertices in the latest graph snapshot. For different types of graph algorithms[1] (i.e., *accumulative* algorithms and *monotonic* algorithms), the operations performed in incremental computation are different [44].

Accumulative Algorithms. When an edge deletion or addition arrives, the contributions (on other vertices' states) made by the previously converged state of the source vertex on this edge need to be cancelled first, and then the new state (i.e., the initial state in the latest graph snapshot) of this source vertex is propagated to other vertices to generate their final states in the latest graph snapshot. For example, if a new edge $v_0 \rightarrow v_4$ is added in Fig. 2(a), the source vertex (i.e., v_0) of $v_0 \rightarrow v_4$ first sends the inverse value (e.g., the negative value for PageRank) of its previously converged state (e.g., v_0's final ranking score in the previous graph snapshot for PageRank) to all outgoing neighbors (i.e., v_1 and v_2). Then, all outgoing edges (i.e., $v_0 \rightarrow v_1$ and $v_0 \rightarrow v_2$) of v_0 are deleted, and their destination vertices (i.e., v_1 and v_2) are set as the vertices affected by graph updates. After that, the inverse value received by these affected vertices (i.e., v_1 and v_2) is further propagated to their successors (e.g., v_2, v_3, v_4, $\cdots$, and v_9) to remove all contributions of the previously converged state of v_0. Then, all edges of v_0 (i.e., $v_0 \rightarrow v_1$, $v_0 \rightarrow v_2$, and $v_0 \rightarrow v_4$) are added back. After that, v_0 is set as a vertex affected by graph update and propagates its new state to other vertices (e.g., v_1, v_2, v_3, $\cdots$, and v_9) along graph topology to produce these vertices' final states in the latest graph snapshot.

Monotonic Algorithms. For each edge addition (e.g., the edge $v_0 \rightarrow v_4$ in Fig. 2(b)), the incremental computation is conducted as follows. The new state of this edge's destination vertex (i.e., v_4) is first calculated according to the weight of the added edge and the previously converged state of the source vertex (i.e., v_0) on this edge (the step ①). After that, v_4 is set as a vertex affected by graph update (the step ②). Then, v_4's new state is propagated to its neighbors (e.g., v_5, v_6, v_7, v_8, and v_9) along graph topology to refine their states to produce their final states in the latest graph snapshot (the step ③).

For each edge deletion (e.g., the edge $v_0 \rightarrow v_2$ in Fig. 2(c)), the incremental computation is conducted as follows. The vertices to be influenced by this edge (i.e., v_3, v_4, v_5, v_6, v_7, v_8, and v_9) need to be first identified by performing a tag propagation [44, 61] (the step ①). Then, the states of the destination vertex (i.e., v_2) on the deleted edge and these identified vertices (i.e., v_3, v_4, v_5, v_6, v_7, v_8, and v_9) are reset to the initial values (e.g., $+\infty$ for SSSP) (the step ②). After that, each reset vertex (e.g., v_2) gathers the states of its incoming neighbors (e.g., v_0) to obtain its new state (the step ③), and then is set as an affected vertex (i.e., v_2) (the step ④). Finally, each affected vertex (e.g., v_2) propagates its new state to refine its neighbors' states to produce their final states in the latest graph snapshot (the step ⑤).

2.2 Problems of the Existing Solutions

In incremental computation of streaming graphs, the key operation is to propagate the new states of affected vertices along the graph topology to refine other vertices' states. However, the new states of different affected vertices are irregularly propagated to the other vertices at different times. Besides, these propagations cause many random accesses to small vertex state data, which are dispersed sparsely in the main memory. Therefore, the existing streaming graph processing systems [21, 24, 32, 33, 51, 53, 59–61] suffer from redundant computation and irregular memory access. To demonstrate these problems, we evaluate four cutting-edge streaming graph processing systems, i.e., GraphBolt [33], KickStarter [61], DZiG [32], and Ligra-o, by running SSSP, where Ligra-o is the optimized version of Ligra [54] introduced in Section 4.1. Section 4.1 presents the details of the platform and benchmarks used in this evaluation. Fig. 3(a) shows that the state propagation consumes most percentage (e.g., more than 93.7% for Ligra-o) of the total execution time in the existing streaming graph processing systems. It is because of the following two problems caused by irregular state propagation along the graph topology.

[1]For *accumulative* algorithms (e.g., Incremental PageRank [44] and Adsorption [44]), the operation of updating the vertex states is an accumulative operation (e.g., *sum*), whereas it is a selection operation (e.g., *min* or *max*) for *monotonic* algorithms (e.g., SSSP [61] and CC [61]).

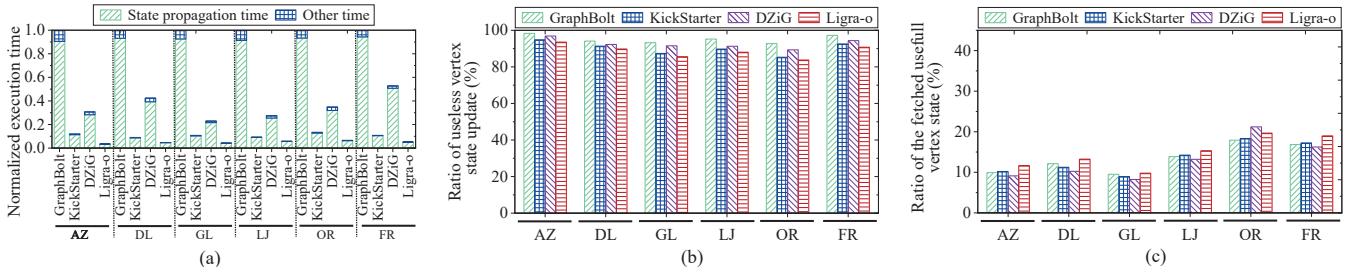


Figure 3: Performance of SSSP by the existing solutions over different streaming graphs: (a) the breakdown of the execution time normalized to GraphBolt; (b) the ratio of the useless vertex state updates to all vertex state updates; (c) the ratio of the fetched useful vertex state data (i.e., removing the fetched state data that are not used) to all fetched vertex state data

Redundant Computation Overhead. We illustrate this problem using the example in Fig. 2, assuming that a batch of graph updates includes the added edges $v_0 \rightarrow v_4$ and $v_0 \rightarrow v_5$. In this example, the states of the affected vertices v_4 and v_5 are first updated to 5 and 14, respectively, and then these two vertices are activated. After that, v_4 propagates its new state along the graph topology, i.e., $v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow \cdots \rightarrow v_9$, while v_5 propagates its new state along $v_5 \rightarrow v_6 \rightarrow \cdots \rightarrow v_9$. In other words, when v_6's state has been updated according to v_5's new state (i.e., s_5=14), s_5 may be simultaneously updated to 12 based on v_4's new state (i.e., s_4=5). Therefore, v_6's state has to be loaded and handled again according to v_5's newest state (i.e., s_5=12). This indicates that it is useless to calculate v_6's state according to v_5's state (i.e., s_5=14) when v_4's new state has not be propagated to v_5, because v_5's state will be updated according to v_4's new state and the dependency between v_4 and v_5. Similarly, the graph data of v_7, v_8, and v_9 also need to be loaded and handled redundantly. In Fig. 3(a) and Fig. 3(b), although Ligra-o outperforms other systems in all cases, more than 83.7% of vertex state updates of Ligra-o are also useless.

Irregular Memory Access. When the existing systems [32, 33, 61] are used for incremental computation of a streaming graph, only a small portion of the vertices need to be processed when the graph has been mutated [20, 21]. That is, only a few vertices' states (which are very small data elements in size [10, 23, 36] and sparsely dispersed in the main memory) need to be loaded for processing. Thus, it incurs many irregular memory accesses. Besides, most vertices' states fetched into the LLC are not needed (the worst case is that only one of the data elements in each cache line is actually needed), which also indicates high redundant data access overhead. From Fig. 3(c), we can find that most vertex states loaded into the LLC are not used before they are swapped out. For example, for Ligra-o, more than 80.4% of the memory accesses to the vertex state data are unnecessary, resulting in the underutilization of memory bandwidth and cache resource.

2.3 Hardware Support for Graph Processing

A number of hardware techniques have been developed to augment the many-core processor for efficient graph processing. GRASP [19] utilizes a domain-specialized cache policy to protect hot vertices against cache thrashing. P-OPT [9] is further designed to use the graph's transpose to ensure Belady's optimal cache replacement for graph processing. To provide efficient vertex state updates, some hardware solutions [48, 68, 69] have also been proposed, and

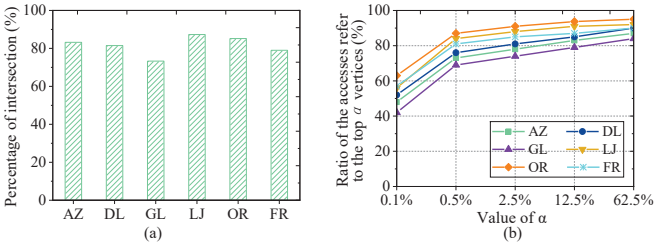


Figure 4: Statistical studies on the characteristics of Ligra-o on SSSP: (a) the intersection of the vertices visited by the state propagations originated from multiple affected vertices; (b) the ratio of the vertex state accesses refer to the top α vertices to that of all vertices

PHI [37] is further proposed to reduce the on-chip traffic through consolidating vertex state updates in the private cache. Meanwhile, some hardware prefetchers [5, 6, 10, 66, 74] are also designed to hide graph data access latency. For achieving faster state propagation and better data locality, several hardware prefetchers, e.g., HATS [36], Minnow [67], and DepGraph [73], have been further proposed. Nevertheless, when using these hardware solutions to support streaming graph processing, they still suffer from significant redundant computation and data access overhead due to irregular state propagations of the vertices affected by the graph updates (see Section 4.3).

2.4 Our Observations

Fig. 4 shows the statistical studies on the characteristics of Ligra-o. We have two main observations.

Observation one: Most propagations of the affected vertices' new states can be conducted together when synchronizing the handling of these vertices along the graph topology. In Fig. 4(a), the intersection of the vertices visited by the propagations of multiple affected vertices' new states accounts for more than 73.3% of all visited vertices, which suggests that most propagations can be conducted together. Taking Fig. 2(b) as an example, the state propagations of the affected vertices (i.e., v_4 and v_5) can be conducted together. Specifically, if the new states of v_4 and v_5 are propagated synchronously along the topological order v_4, v_5, v_6, $\cdots$, and v_9, the new state of v_4 is first propagated to v_5. When v_4's new state has been propagated to v_5, the new state of v_5 is accumulated with the received new state of v_4, and then they are propagated together to v_6, v_7, v_8, and v_9

in sequence. In this way, the common vertices (e.g., v_6, v_7, v_8, and v_9) to be visited by these two propagations are loaded and handled only once, instead of twice required by the existing solutions.

Observation two: In streaming graph processing, most vertex state accesses usually refer to a small set of vertices. This observation is made via the following experiment. When a batch of graph updates have been applied, the vertices are first sorted in the descending order according to the number of times for which their states are accessed. Then, during the incremental computation of the latest graph snapshot, we evaluate the proportion of the vertex state accesses referring to the top α vertices to those of all vertices. Fig. 4(b) shows that more than 69.3% of the state accesses refer to the states of the top 0.5% vertices. This indicates that many irregular memory accesses can be alleviated by consolidating the accesses to the states of the most frequently-accessed vertices.

3 OVERVIEW OF OUR SOLUTION

According to the above observations, we propose an efficient accelerator *TDGraph* to augment many-core processors using our proposed *topology-driven incremental execution approach* for high-performance streaming graph processing. In this section, we first introduce the main idea of our proposed execution approach, and then present the detailed hardware-software codesign of TDGraph.

3.1 Topology-Driven Incremental Execution Approach

This subsection presents the main idea of our proposed topology-driven incremental execution approach, which regularizes the state propagations originated from different affected vertices to reduce redundant computation and also achieves the coalesced accesses to the states of the frequently-accessed vertices. The details of our approach are introduced below.

Topology-Driven Incremental Processing. To regularize the state propagations originated from the vertices affected by graph updates, it first tracks the dependencies between these affected vertices and their successors according to the graph topology. Then, based on this tracked information, some affected vertices are assigned as the roots to retrieve other vertices along the graph topology and synchronously drive the incremental processing of them for regular state propagations.

In detail, for each affected vertex, it first tracks the state propagations (originated from the affected vertices) to pass through this vertex on the fly according to the graph topology. After that, for each affected vertex, only when all state propagations that need to pass through it have been accumulated by it, this vertex is used as the root to retrieve the edges of other vertices for incremental processing along the graph topology in a depth-first fashion [2]. By such means, the new state of each affected vertex is accumulated with the new states of all of its affected predecessors to be synchronously propagated to their common neighbors together along the graph topology. Then, the redundant overhead for loading and processing these common vertices can be reduced. Besides, the new

[2]The depth-first fashion can use a stack with a fixed depth to achieve almost the same performance as that with the deeper stacks, ensuring less hardware cost. Breadth-first fashion requires a large FIFO queue for efficiency.

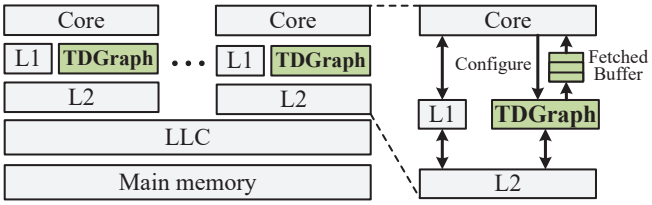


Figure 5: System architecture

states of the affected vertices are allowed to be propagated faster along the graph topology.

We illustrate the above approach using the example in Fig. 2(b). Assume that the edges $v_0 \rightarrow v_4$ and $v_0 \rightarrow v_5$ are added into the graph as a batch. The affected vertices are then v_4 and v_5. In this example, the edge $v_4 \rightarrow v_5$ imposes a dependency between v_4 and v_5, and thus the state propagation originated from v_4 will pass through v_5. Therefore, only when the state propagation originated from v_4 has reached v_5 and has been accumulated by v_5, the affected vertex v_5 is assigned as the root to retrieve v_5's neighbors for processing. Then, the states of both v_4 and v_5 can be propagated together to v_6, which can then work on v_6 immediately. This indicates that the state propagations originated from both v_4 and v_5 can pass through their common successors (e.g., v_6, v_7, v_8, and v_9) together along the graph topology. In this way, for the state propagations of both v_4 and v_5, the graph data associated with v_6, v_7, v_8, and v_9 are loaded and handled only once. In addition, within the same iteration, each vertex's new state (e.g., s_6) can immediately work on its neighbors (e.g., v_7), and then sequentially work on v_8 and v_9 along the graph topology similarly, which offers the fast state propagation.

Vertex States Coalescing. As shown in Fig. 4(b), in streaming graph processing, most vertex state accesses usually refer to a small set of frequently-accessed vertices for the real-world graphs [22, 62]. To achieve better data locality for the accesses to the states of these frequently-accessed vertices, we consolidate the storage of these vertex states into a contiguous memory region (i.e., an in-memory array which we call *Coalesced_States*). A hash table is also used to index each frequently-accessed vertex, which help retrieve its corresponding state efficiently from the right position of the *Coalesced_States*. Then, only the states of the frequently-accessed vertices are swapped into the same cache lines of the LLC, when one of these states is accessed. One off-chip communication can serve multiple propagations (which need to access the main memory more times in existing solutions).

A vertex will be accessed more times when more state propagations (originated from the affected vertices) need to reach it. Thus, we use the number of state propagations (originated from the affected vertices) to pass through each vertex to approximate to the access frequency of this vertex. To identify the frequently-accessed vertices, a parameter α ($0 \leq \alpha \leq 1$, whose value is specified by the users) is used to capture the ratio of the frequently-accessed vertices to all vertices in the latest graph snapshot. α's proper value is usually small because of the power-law property of the graphs [22]. In our solution, the value of α is 0.5% by default.

Runtime Overhead. The software-only implementation of the above approach suffers from high runtime cost (see Section 4.2). First, it needs to fetch the graph data originated from the affected vertices on the fly via the irregular traversals of the graph. Second,

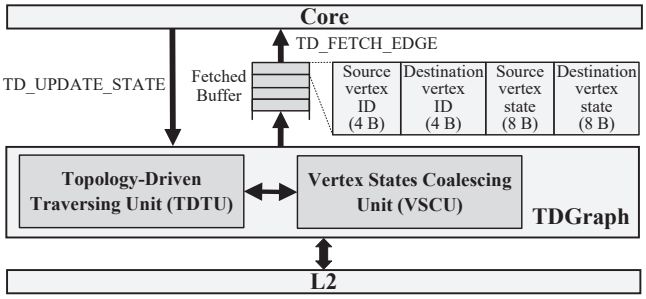


Figure 6: Microarchitecture of TDGraph

it induces additional instructions and also causes low instruction level parallelism due to the data-dependent branches of these instructions. Third, it needs to frequently index the states of the frequently-accessed vertices.

3.2 Overview of TDGraph

The high runtime cost of the software-only implementation may outweigh the performance improvement brought by our approach, which motivates the design of our TDGraph. Different from existing solutions, the key idea of the TDGraph is to efficiently support our topology-driven incremental execution approach, which embraces a novel hardware-software codesign with topology-driven architecture on standard processor cores. Fig. 5 shows that each core of the many-core processor is coupled with a TDGraph engine, which resides between this core and its private L2 cache. Like prior works [36, 67, 73], TDGraph operates on virtual addresses and accesses the graph data through the L2 TLB and L2 cache of its paired core. The data will be retrieved from the LLC, or from the main memory, when the data are not in the L2 cache. Each TDGraph engine contains two lightweight hardware units, i.e., *Topology-Driven Traversing Unit* (TDTU) and *Vertex States Coalescing Unit* (VSCU), as shown in Fig. 6.

At the execution time, the existing software streaming graph processing system that is running on the cores invokes TDGraph. Then, TDTU prefetches and synchronizes the incremental processing of the graph data originated from the vertices affected by graph updates on the fly. The graph data fetched by TDTU are passed to the existing software streaming graph processing system, which in turns handles these graph data. The accesses to the states of the frequently-accessed vertices are consolidated by VSCU. The details of our hardware-software codesign are presented next.

3.2.1 Software-level Operations. An existing software streaming graph processing system (which runs on the cores) is used to apply the graph updates, divide the graph into chunks for parallel processing over the cores, initialize the data structures (e.g., the *Coalesced_States* array) needed by TDGraph, call the APIs provided by TDGraph to configure TDGraph, identify the frequently-accessed vertices, process the graph data prefetched by TDGraph, ensure load balancing (e.g., utilizing the work-stealing strategy [12]), etc. Note that the vertices affected by the graph updates are also set by the existing software system as the initial active vertices, which drive the incremental computation of the streaming graph, and the information of these affected vertices is used to configure TDGraph.

3.2.2 Hardware-level Operations. In this subsection, we present the operations of TDTU and VSCU.

Operations of TDTU. For each chunk allocated to the paired core, TDTU tracks the dependencies between the affected vertices and their successors in this chunk. Then, according to the tracked dependencies, TDTU assigns the traversing order of these vertices for synchronizing the state propagations of the affected vertices. After that, TDTU takes the assigned vertices as the roots to traverse the edges in this chunk in a depth-first fashion, aiming to prefetch the graph data associated with these visited edges into a FIFO queue called *Fetched Buffer*. These prefetched graph data will be processed by the software streaming graph system running on the paired core. TDGraph offers a low-level API, called *TD_fetch_edge*(), to enable the software system to fetch the graph data from the *Fetched Buffer*, where this API translates to an ISA instruction *TD_FETCH_EDGE* provided by TDGraph.

Operations of VSCU. For each chunk allocated to the paired core, VSCU consolidates the states of the frequently-accessed vertices in this chunk into the in-memory array *Coalesced_States* (its each entry stores the state of a vertex). For each vertex state access (e.g., the TDTU of the TDGraph engine paired with this core needs to prefetch the state of a vertex), VSCU first identifies whether this vertex is a frequently-accessed vertex (detailed in Section 3.3.3). If so, VSCU produces the address of this vertex's state in *Coalesced_States* for the access of this state data. Otherwise, VSCU provides the address of this vertex state in the in-memory array *Vertex_States_Array* (introduced in Section 3.3.1). Note that a low-level API, i.e., *TD_update_state*(), is provided for the software streaming graph processing system to update a vertex state. This API is implemented using the ISA instruction *TD_UPDATE_STATE* offered by TDGraph. For the frequently-accessed vertex, this API updates its state data in the corresponding entry of *Coalesced_States*. For the other vertex, this API updates its state data in *Vertex_States_Array*. That is, at the processing time, the state of the frequently-accessed vertex is only written to the corresponding entry in *Coalesced_States*. When the processing ends, the vertex states in *Coalesced_States* are written back to *Vertex_States_Array*.

3.2.3 Cooperation of Hardware and Software. In this subsection, we use the example in Fig. 2 to illustrate the cooperation of software and hardware operations in TDGraph. When a batch of graph updates (i.e., the additions of $v_0 \rightarrow v_4$ and $v_0 \rightarrow v_5$, and the deletion of $v_0 \rightarrow v_2$) have been applied to the graph by the software streaming graph processing system, this software system identifies the set of vertices (i.e., v_2, v_4, and v_5) affected by the graph updates, and then sets these vertices as the initial active vertices, where an in-memory bitvector *Active_Vertices* is used to indicate whether a vertex is active.

When a thread running on a core needs to process a chunk (each chunk contains the graph data associated with a continuous range of vertices, e.g., v_2, v_3, v_4, and v_5 in Fig. 2) assigned by the software system, this thread needs to configure the TDGraph engine of this core by calling an API *TD_configure*(). After that, TDGraph tracks the state propagations (originated from the affected vertices, i.e., v_2, v_4, and v_5) that will pass through each vertex in this chunk. According to the tracked information, TDGraph repeatedly selects a suitable active vertex (e.g., v_2) in this chunk as the root

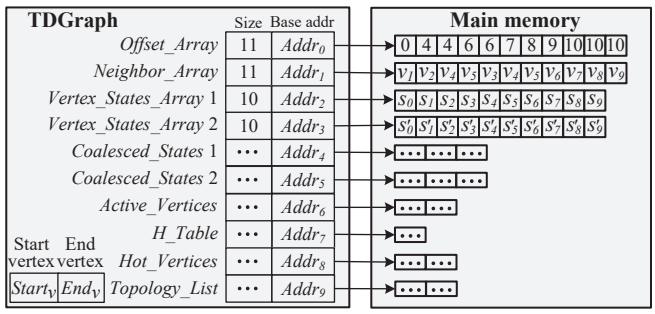


Figure 7: The configuration information of TDGraph

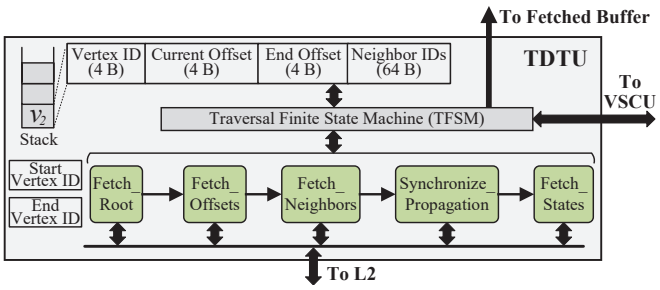


Figure 8: Microarchitecture of TDTU

to prefetch the graph data in this chunk (e.g., the edge $v_2 \rightarrow v_4$, and the states of v_2 and v_4) along the graph topology until there is no active vertex. These prefetched graph data are stored in the *Fetched Buffer*. By calling the API, i.e., *TD_fetch_edge*(), the thread running on this core can efficiently fetch the graph data from the *Fetched Buffer* for incremental computation. The thread can also call the API *TD_update_state*() to update the states of the frequently-accessed vertices in the right position of *Coalesced_States*.

3.3 Hardware Design

3.3.1 Initialization and Configuration. This subsection introduces the key data structures needed by TDGraph, and then explains their initialization and the configuration of TDGraph.

Key Data Structure. Because *Compressed Sparse Row* (CSR) is the most popular format, like JetStream [44], each graph snapshot is stored in the CSR format by default. Specifically, three in-memory arrays, i.e., *Offset_Array*, *Neighbor_Array*, and *Vertex_States_Array*, are used. *Offset_Array* records the begin offset and end offset of each vertex's neighbors in *Neighbor_Array*, while *Neighbor_Array* stores the outgoing neighbors of each vertex. *Vertex_States_Array* maintains the algorithm-specific state data for each vertex.

Meanwhile, two in-memory bitvectors, i.e., *Active_Vertices* and *Hot_Vertices*, are used to record the active vertices and the frequently-accessed vertices, respectively. An in-memory array *Topology_List* is created to store the number of propagations (originated from the affected vertices) that will pass through each vertex. The states of the frequently-accessed vertices are consolidated in the in-memory array *Coalesced_States*. To efficiently retrieve the state of each frequently-accessed vertex, an in-memory hash table *H_Table* is also employed to index the state of this vertex in *Coalesced_States*. Note that there are $\frac{\alpha \times |V|}{\sigma}$ entries in *H_Table*, where σ is set as 0.75 by default [45] and an entry in *H_Table* takes the form of <*vertex ID*, *vertex_offset*>. Note that the storage space of these data structures can be expanded dynamically when it is not enough (e.g., the graph becomes larger).

Initialization of Data Structure. When receiving a batch of graph updates, the software streaming graph processing system applies these graph updates to the graph to generate the latest graph snapshot, and also identifies the initial active vertices to initialize *Active_Vertices* according to these graph updates.

Configuration of TDGraph. To process a chunk on a core, as shown in Fig. 7, the default configuration information of TDGraph paired with this core includes: (a) the base addresses and sizes of *Offset_Array*, *Neighbor_Array*, *Vertex_States_Array*, *Active_Vertices*, *Hot_Vertices*, *Topology_List*, *Coalesced_States*, and *H_Table*; (b) ID of the start vertex (i.e, $Start_v$) and ID of the end vertex (i.e, End_v) of the chunk assigned to this core. Similar to the configuration method of a DMA engine, the above configuration information is conveyed into the memory-mapped registers of TDGraph. Note that, if a thread is descheduled by the OS, it quiesces TDGraph and only saves $Start_v$ (which records the offset of the next edge to be handled in the assigned chunk), since the basic processing unit of the existing software streaming graph systems [32, 33, 61] is an edge. Other information (e.g., the data structures' base addresses and sizes) is not saved because it remains unchanged during execution. When the quiesced thread is rescheduled, the information (e.g., $Start_v$) is resumed.

3.3.2 Topology-driven Graph Data Prefetching. When a chunk is dispensed on a core for processing, the TDTU of the TDGraph engine paired with this core first tracks the graph topology information for efficient synchronization of the affected vertices' propagations, and then prefetches the graph data along the graph topology according to the tracked information. Both the operations *graph topology tracking* and *graph data prefetching* are carried out in a depth-first fashion and share the same set of hardware resources. To distinguish these two operations, a *flag* is used with the initial value being 0. When the *flag* is 0, TDTU conducts *graph topology tracking*, while TDTU performs *graph data prefetching* when the *flag* is 1. Note that a hardware stack with the fixed depth is adopted by TDTU to store the intermediate information for the depth-first graph traversal, where the traversal depth is bounded. As shown in Fig. 8, a level of the stack stores the following information of a visited vertex: (a) this vertex's ID; (b) the current/end offsets of this vertex's unvisited edges; (c) a cache line of IDs of this vertex's neighbors. The details of these two operations are presented below.

Graph Topology Tracking. To efficiently track the graph topology information for synchronization, four stages, i.e., *Fetch_Root*, *Fetch_Offsets*, *Fetch_Neighbors*, and *Synchronize_Propagation*, are used as shown in Fig. 8, which are implemented as a pipeline and managed by *Traversal Finite State Machine* (TFSM) of TDTU. Specifically, when the hardware stack is empty, the *Fetch_Root* stage takes an initial active vertex (e.g., v_2) as the root vertex by scanning *Active_Vertices*, and pushes this vertex into the stack. In the *Fetch_Offsets* stage, the topmost vertex (i.e., v_2) in the stack is taken to retrieve the begin/end offsets of its neighbors from the *Offset_Array*. In the *Fetch_Neighbors* stage, it retrieves a cache line of IDs of this vertex's unvisited neighbors from *Neighbor_Array*.

After that, in the *Synchronize_Propagation* stage, it pushes one of these fetched neighbors (e.g., v_4) into the stack, and the edge (i.e., $v_2 \rightarrow v_4$) between the previous topmost vertex (i.e., v_2) and the current topmost vertex (i.e., v_4) is set as visited. Meanwhile, the value for v_4 in the *Topology_List* (the initial value is zero for each vertex) is also increased by one, which indicates that a state propagation originated from the root vertex v_2 passes through v_4.

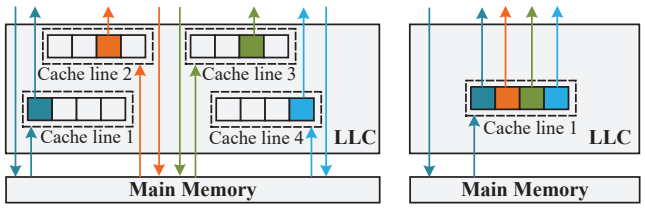
At the end of the final stage of the pipeline, if the topmost vertex is an initial active vertex or no other vertices need to be visited, it pops out the topmost vertex from the stack and then the next neighbor of the new topmost vertex in the stack is used as the root to start a new traversal, from which the above process repeats. When all edges of the initial active vertices and their successors have been visited, the number of state propagations (originated from these initial active vertices) that will pass through each vertex in this chunk is produced.

Graph Data Prefetching. To achieve efficient graph data prefetching according to the above tracked information, TDTU uses TFSM to manage a pipeline with five stages, i.e., *Fetch_Root*, *Fetch_Offsets*, *Fetch_Neighbors*, *Synchronize_Propagation*, and *Fetch_States*.

Specifically, as shown in Fig. 8, in the *Fetch_Root* stage, when the stack is empty, a suitable active vertex (i.e., its value in the *Topology_List* is zero[3]) is selected as the root vertex (e.g., v_2). Then, this vertex is pushed into the stack and set as inactive in the *Active_Vertices*. Note that the operations of the *Fetch_Offsets* and *Fetch_Neighbors* stages are the same as those of the *graph topology tracking*. In the *Synchronize_Propagation* stage, one of its neighbors (e.g., v_4) fetched by the *Fetch_Neighbors* stage is pushed into the stack, and an edge (i.e., $v_2 \rightarrow v_4$) is also produced and enqueued into the *Fetched Buffer*. This produced edge is also set as visited. Meanwhile, the value for this neighbor (i.e., v_4) in the *Topology_List* decreases by one, which indicates that the propagations that pass through this edge (i.e., $v_2 \rightarrow v_4$) have reached this neighbor v_4. After that, the *Fetch_States* stage retrieves the states (i.e., s_2 and s_4) of the source and destination vertices on the produced edge from the *Coalesced_States* or the *Vertex_States_Array* according to the addresses provided by VSCU (detailed in Section 3.3.3) and put them into the *Fetched Buffer*.

At the end of the final stage of the pipeline, if the value of the topmost vertex in the *Topology_List* is greater than zero[4] or no other vertices need to be visited, the topmost vertex is popped out of the stack and the next neighbor of the new topmost vertex in the stack is then used as the root to start a new traversal and further prefetch the graph data. Note that the last visited vertex of each traversal is set as an active vertex, which can be taken as a new root vertex. The above procedure repeats until all related edges of the active vertices in this chunk have been visited.

3.3.3 Vertex States Coalescing. When a vertex v_i's state (i.e., s_i) needs to be accessed, VSCU first uses *Hot_Vertices* to identify whether this vertex is a frequently-accessed vertex. Note that the frequently-accessed vertices are identified by the software streaming graph processing system according to the information in the *Topology_List* tracked by TDTU, and these vertices are recorded in the in-memory bitvector *Hot_Vertices*.



(a) State accesses of existing solutions (b) Coalesced state accesses

Figure 9: Illustration of the coalesced vertex state accesses in VSCU

If v_i is not a frequently-accessed vertex, VSCU directly returns the address of s_i in the *Vertex_States_Array*. Otherwise, VSCU uses the ID of v_i to obtain the offset (i.e., $vertex\_offset_i$) of v_i's state in the *Coalesced_States* according to the *H_Table*. Note that, if the corresponding offset of v_i cannot be obtained from *H_Table* (e.g., this is the first time to access v_i's state), VSCU will fetch v_i's state from the *Vertex_States_Array* and then sequentially consolidate v_i's state into an empty entry in the *Coalesced_States*. A corresponding entry (i.e., $<v_i, vertex\_offset_i>$) is also created in *H_Table*. After that, the address of s_i in the *Coalesced_States* can be produced by performing $base\_address + vertex\_offset_i$, where the *base_address* is the base address of the *Coalesced_States*. Then, according to the produced address, the access to the state of the frequently-accessed vertex can be directed to the right position in the *Coalesced_States*, where multiple accesses can be directed to the data elements in the same cache line. By such means, the frequently-accessed vertex states in the same cache line can serve multiple state propagations, which reduces data access overhead as shown in Fig. 9.

Note that, when the data of the *Coalesced_States* are replicated in the cache, the cache coherence is ensured by the existing caching system using MESI [15, 29] by default.

4 EVALUATION

4.1 Experimental Methodology

Simulation Infrastructure. We have simulated a 64-core processor using ZSim [46], whose parameters are listed in Table 1. In detail, the simulated processor employs the out-of-order cores that are modeled after and validated against the Intel Skylake cores. Each core owns the private L1 and L2 caches, while all simulated cores share the last-level cache. Note that each simulated core is also extended to support AVX512 as MacSim [2]. To communicate with each other, the simulated cores adopt the mesh network with the parameters similar as in Intel Knights Landing [55]. Then, we implement TDGraph into the simulated 64-core processor. We compiled all programs using GCC 9.2 with the -O3 flag and the vectorization being enabled.

Benchmarks. In our experiments, we use four graph algorithms (i.e., *Incremental PageRank* (PageRank) [44], Adsorption [44], *Single Source Shortest Path* (SSSP) [61], and *Connected Components* (CC) [61]) in two categories: PageRank and Adsorption are the representative graph algorithms with the accumulative update

[3]It means that all state propagations that pass through this vertex have been accumulated by this vertex. Note that, when the paired core is idle, the active vertex with the lowest value in the *Topology_List* is selected as a root vertex.

[4]It indicates that the state update of this topmost vertex needs to wait for the arrival of the state propagations of some affected vertices, because some propagations originated from the affected vertices have not reached this vertex.

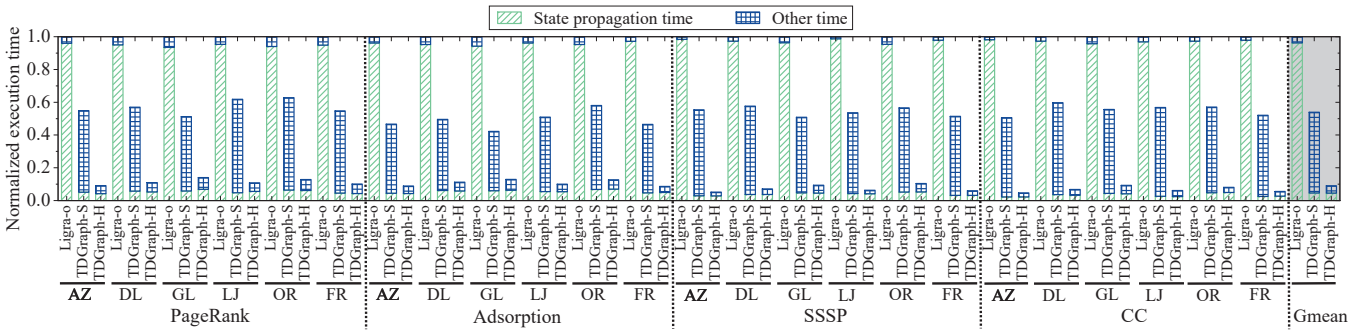


Figure 10: Execution time of various schemes normalized to that of Ligra-o

Table 1: Configuration of the simulated system

Cores	64 cores, x86-64 ISA, 2.5 GHz, Skylake-like OOO [46]
L1 Instruction Cache	32 KB per-core, 4-way set-associative, 3-cycle latency
L1 Data Cache	32 KB per-core, 8-way set-associative, 4-cycle latency
L2 cache	256 KB, private per-core, 8-way set-associative, 7-cycle latency
L3 cache	64 MB, shared, 16-way hashed set-associative, 27-cycle bank latency, DRRIP replacement [25]
Global NoC	8×8 mesh, 512-bits/cycle/link, X-Y routing, 3 cycles/hop
Coherence	MESI, 64 B lines, in-cache directory, no silent drops
Memory	12-channel DDR4 3200 CL17

Table 2: Characteristic statistics of datasets (d is the graph diameter and $\overline{D}$ is the average vertex degree)

Datasets	*#Vertices*	*#Edges*	d	$\overline{D}$
com-Amazon (AZ) [3]	334,863	925,872	44	6
com-DBLP (DL) [3]	317,080	1,049,866	21	7
ego-Gplus (GL) [3]	2,394,385	5,021,410	9	2
LiveJournal (LJ) [3]	4,847,571	68,993,773	17	17
Orkut (OR) [3]	3,072,441	117,185,083	9	76
Friendster (FR) [3]	65,608,366	1,806,067,135	32	29

operation, while SSSP and CC are used to evaluate the performance of monotonic graph algorithms. Table 2 lists the six real-world graph datasets used in the evaluation. Like the existing solutions [32, 33, 53, 59, 61], we first load 50% of the graph edges to obtain an initial fixed point. Then, the remaining edges are randomly streamed in to model the edge additions, while the deleted edges are randomly sampled from the loaded graph. The edge additions and deletions are mixed in each batch of graph updates, where each batch contains 100K edge updates by default.

Evaluation Methodology. We first incorporate several optimizations (e.g., a state-of-the-art incremental computation technique [44], software prefetching [7], careful loop unrolling [36, 37, 64], the optimization to utilize SIMD [75], and other optimizations [70, 72]) into the best-performing shared-memory graph processing system Ligra [54], which enables Ligra to efficiently support streaming graph processing. We call the optimized version of Ligra as *Ligra-o*. Ligra-o outperforms the cutting-edge streaming graph processing systems, i.e., GraphBolt [33], KickStarter [61], and DZiG [32], by up to 28.4, 3.5, and 10.2 times, respectively (see Fig. 3(a)).

Then, Ligra-o is employed as the baseline software system to evaluate the performance of TDGraph. Specifically, we integrate Ligra-o with the software-only TDGraph and the hardware implemented TDGraph to obtain TDGraph-S and TDGraph-H, receptively, where the depth of the stack is 10 and α=0.5% by default. Note that, for different graph datasets, the memory overhead required by TDGraph for maintaining the extra data structures (e.g., *Coalesced_States*, *H_Table*, and *Topology_List*) accounts for 0.3%∼2.4% of the total memory consumption. Meanwhile, we also integrate Ligra-o with four cutting-edge graph processing accelerators, i.e., HATS [36], Minnow [67], PHI [37], and DepGraph [73], respectively. After that, TDGraph-H is compared with these accelerators. Note that we employ McPAT [31] to evaluate the energy consumption of the chip components and utilize DDR4 power calculator [1, 58] to calculate the energy of the main memory.

4.2 Comparison with Software Approaches

Fig. 10 shows the execution time on the simulated platform. The breakdown of the execution time is also presented, including the state propagation time (which includes the time for fetching the graph data associated with each vertex and the time to update vertex state, etc.) and the other time. The state propagation time spent by TDGraph-S is only 2.3%∼7.1% of that by Ligra-o. This is because Ligra-o suffers from significant redundant computation and data access cost due to the irregular state propagations along the graph topology.

Nevertheless, TDGraph-S only outperforms Ligra-o slightly, because TDGraph-S suffers from high runtime cost. In Fig. 10, the other time spent by TDGraph-S (e.g., the time for dependency tacking, the time for fetching graph data on the fly along the graph topology, and the time for indexing the states of frequently-accessed vertices) occupies 85.2%∼94.7% of the total execution time. It impedes the overall efficiency of our approach. Compared with TDGraph-S, TDGraph-H not only offers fewer redundant vertex state updates and better data locality than Ligra-o, but also reduces the runtime

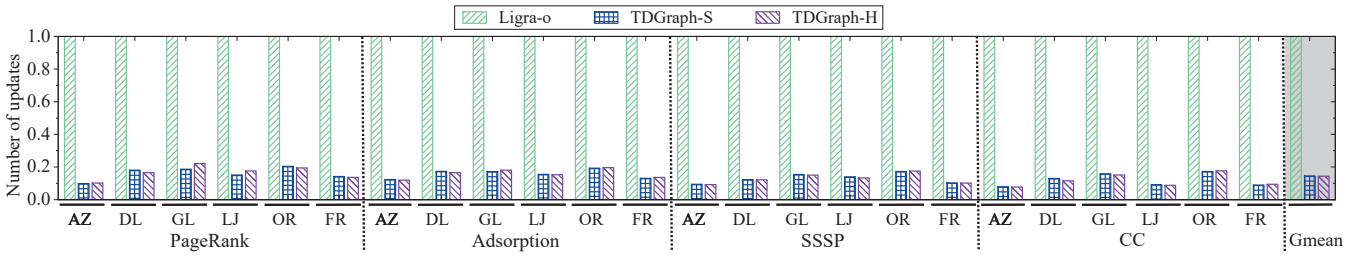


Figure 11: The number of vertex state updates of various schemes normalized to that of Ligra-o

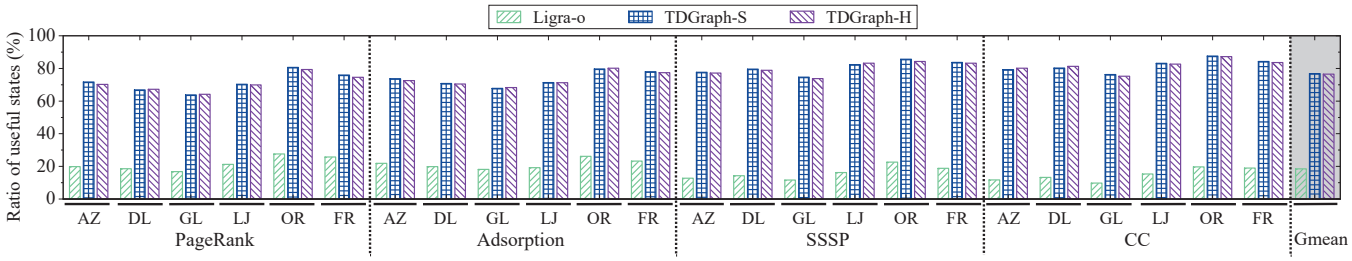


Figure 12: The ratio of the fetched useful vertex state data to all fetched vertex state data

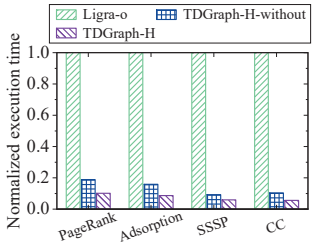


Figure 13: Normalized execution time of different schemes over *FR*

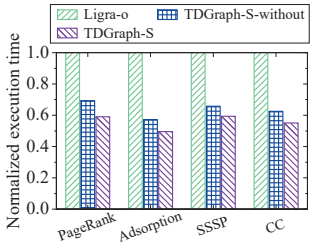


Figure 14: Execution time of different schemes over *FR* on a real platform

cost of TDGraph-S. TDGraph-H outperforms Ligra-o and TDGraph-S by 7.1~21.4 and 3.6~10.8 times, respectively.

In Fig. 11, the number of vertex state updates performed by TDGraph-H is only 7.8%~22.1% of that by Ligra-o. The reason is that the state propagations originated from different affected vertices are efficiently regularized by TDGraph-H, which reduces many redundant vertex state updates. It also indicates that TDGraph-H reduces much data access cost associated with these redundant updates. In Fig. 12, the ratio of the useful vertex states fetched by TDGraph-H is higher than that by Ligra-o. These results indicate that most vertex states loaded by TDGraph-H can serve state propagations multiple times, ensuring a higher utilization of the memory bandwidth and cache resource. The reason for this is that the states of the frequently-accessed vertices are consolidated efficiently into the same cache lines in TDGraph-H.

Fig. 13 shows that TDGraph-H-without (i.e., only TDTU is enabled while VSCU is disabled) also outperforms Ligra-o by 5.3~10.8 times. On top of TDTU, VSCU further improves the performance by 1.5-1.9 times. Besides, as shown in Fig. 14, TDGraph-S-without also outperforms Ligra-o on a real platform with a 64-core Intel Xeon Phi 7210 processor and 64 GB main memory, where TDGraph-S-without is the version of TDGraph-S without using the vertex states coalescing strategy.

Table 3: Power and area cost of different accelerators

Hardware Accelerators	Power		Area	
	(mW)	%TDP	(mm^2)	%core
HATS	425	0.22%	0.007	0.38%
Minnow	849	0.43%	0.017	0.92%
PHI	493	0.25%	0.008	0.43%
DepGraph	562	0.29%	0.011	0.61%
TDGraph	**647**	**0.34%**	**0.013**	**0.73%**

4.3 Comparison with Hardware Accelerators

Fig. 15 shows that TDGraph-H outperforms HATS, Minnow, PHI, and DepGraph by 4.6~12.7, 3.2~8.6, 3.8~9.7, and 2.3~6.1 times, respectively, because TDGraph-H can significantly reduce the updates of vertex states and achieve better data locality. Note that the average LLC miss rates are 68.5%, 75.7%, 63.2%, 72.1%, and 24.3% for HATS, Minnow, PHI, DepGraph, and TDGraph-H, respectively. TDGraph-H is also compared with GraphPulse [43] and JetStream [44]. Figure 16 shows that more useless data are prefetched by JetStream than TDGraph-H. Note that GraphPulse requires much more memory accesses, although most prefetched data are useful. Thus, TDGraph-H outperforms JetStream and JetStream-with as shown in Fig. 17, where JetStream-with is the version of JetStream with state coalescing as VSCU. Figure 18 shows that TDGraph-H also outperforms GRASP, where TDGraph-H-GRASP uses both GRASP (without using VSCU) and TDTU.

4.4 Area Cost and Energy Evaluation

Table 3 lists the area and power consumptions of different accelerators under typical operating conditions. These accelerators are implemented using Verilog RTL and are synthesized using a commercial 14 nm process with the same target frequency. The area cost incurred by TDGraph-H only accounts for 0.73% of that by a general-purpose core of the simulated chip. This is because most

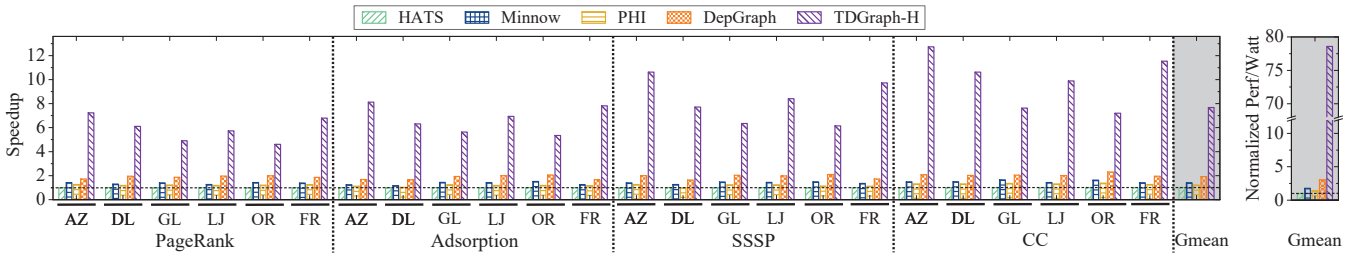


Figure 15: Speedups of different schemes against HATS and Perf/Watt of different schemes normalized to that of HATS

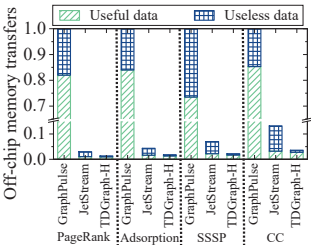


Figure 16: Normalized volume of off-chip memory transfers over *FR*

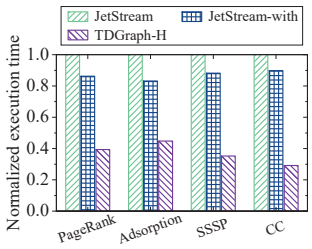


Figure 17: Execution time of JetStream and TDGraph-H over *FR*

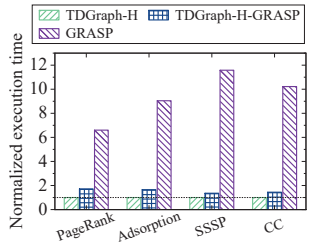


Figure 18: Execution time over *FR*

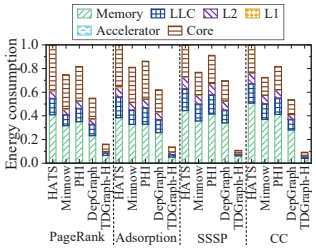


Figure 19: Energy breakdown over *FR*

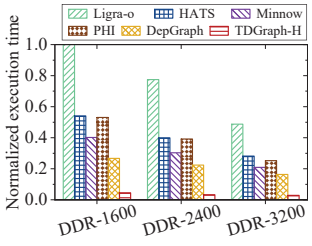


Figure 20: Sensitivity to the memory bandwidth on SSSP

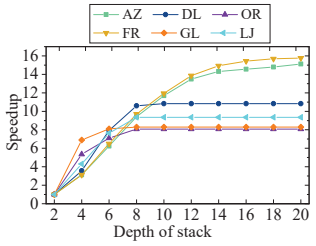


Figure 21: Sensitivity to the depth of stack on SSSP

information required by TDGraph-H is stored in the existing memory subsystem. The area cost of TDGraph-H is only a few buffer (i.e., the storage of 4.8 Kbits for the FIFO *Fetched Buffer* and 6.1 Kbits for the stack) and its hardware logic (i.e., TDTU and VSCU). Fig. 19 shows the energy breakdown normalized to that of HAST over *FR*. It shows that much less energy consumption is required by TDGraph-H due to fewer updates of vertex states and less memory traffic. Note that both the ratio of the area/power of different hardware accelerators to that of the simulated core and the energy breakdown are evaluated approximately (with some errors) using McPAT [31] at 22 nm.

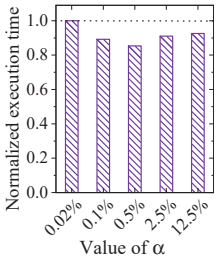


Figure 22: Impact of α

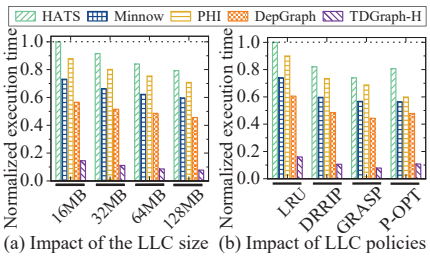


Figure 23: Impact of LLC on SSSP over *FR*

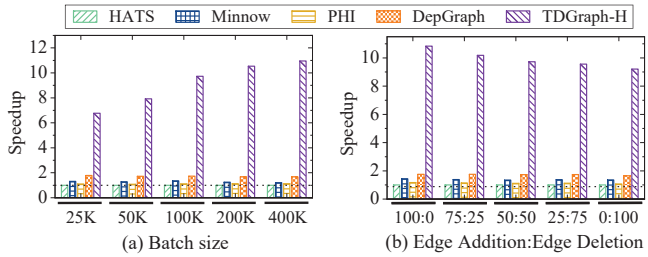


Figure 24: Impact of (a) batch size and (b) batch composition

4.5 Sensitivity Studies

Fig. 20 shows the impact of memory bandwidth over *FR*. We can observe that TDGraph-H always outperforms other schemes due to higher utilization of memory bandwidth. Fig. 21 evaluates the sensitivity to the depth of the stack for DepGraph-H. When the stack depth is more than ten, the performance of TDGraph-H stays almost unchanged. It means that TDGraph-H is insensitive to the stack depth when the depth is larger than ten, and therefore a fixed depth can simply be used. Fig. 22 shows the performance of TDGraph-H on SSSP over *FR* under different values of α. The setting of α is a trade-off. Selecting too few frequently-accessed vertices may incur many accesses, whereas selecting too many such vertices may impede the overall efficiency due to higher cost.

Fig. 23 evaluates the performance under various LLC configurations. It shows that TDGraph-H consistently outperforms the other schemes as the LLC size increases. Moreover, compared with the other LLC management methods, i.e., LRU [27], DRRIP [25], and P-OPT [9], TDGraph-H achieves better performance when GRASP [19] is employed. This is because GRASP can efficiently prevent the coalesced states of the frequently-accessed vertices from cache thrashing.

Finally, Fig. 24(a) shows the performance of SSSP over *FR* with different bath sizes. TDGraph-H achieves higher performance as the batch size increases, because the propagations originated from

more vertices can be regularized by TDGraph-H. Fig. 24(b) evaluates the performance of SSSP over *FR* when the ratio of the number of edge additions to that of edge deletions varies. It shows that TDGraph-H always outperforms the other schemes under different batch compositions.

5 ADDITIONAL RELATED WORK

Streaming Graph Processing Systems. Recently, many streaming graph processing systems [21, 26, 38, 52, 53, 59, 60] have been proposed. Kineograph [14] and GraphIn [51] use incremental computations, however, may produce incorrect results for edge deletions. For edge deletions, KickStarter [61] and GraphBolt [33] are further designed to trace the dependencies across intermediate results. For better performance, DZiG [32] proposes a DelZero-aware incremental refinement strategy. Nevertheless, these software systems still suffer from significant redundant computations and data access overhead because of the irregular state propagation along the graph topology. In comparison, TDGraph is designed to augment the many-core processors to efficiently reduce the redundant computation overhead and data access cost for the streaming graph algorithms by regularizing the state propagations and achieving better data locality. Note that SLFE [57] also uses topology information to reduce redundant computations, however, suffers from high runtime cost to extract the topological information for streaming graph processing and also generates extra instructions, which have data-dependent branches that limit the parallelism. Wonderland [70] and FBSGraph [72] can also reduce redundant computations, however, still suffer from many redundant computations for streaming graph processing due to irregular state propagation.

Other Related Hardware Accelerators: Many hardware accelerators [4, 8, 16, 28, 34, 40, 41, 65] have also been designed for graph processing recently. Graphicionado [23] is the first ASIC-based one and can reduce random memory accesses. ISCU [49, 50] designs a compacting and filtering technique to prepare data for SMs of GPU for higher GPU utilization. However, when serving the streaming graph processing on CPU, ISCU not only suffers from serious redundant computation overhead, but also needs to issue multiple accesses for the compacting/filtering of multiple data elements that reside in multiple cache lines. To alleviate the irregularities in graph processing, GraphDynS [63] employs a hardware/software co-design approach. For efficient asynchronous graph processing, GraphPulse [43] designs a novel event-driven accelerator. Meanwhile, to reduce data movements, GraphPIM [39], GraphP [71], and GraphQ [76] are designed to use the processing-in-memory technique, while GraphR [56] and GaaS-X [13] leverage the massive parallelism of ReRAM. Nevertheless, these hardware techniques are mainly designed for the processing of static graphs.

For streaming graph analytics, Basak *et al.* [11] propose an input-aware software and hardware codesign to accelerate graph updating. Toward high-performance steaming graph processing, some ASIC-based accelerators have been designed. DREDGE [35] provides a hardware accelerator to accelerate the repartitioning of streaming graphs. JetStream [44] is designed to use an event-driven computation model to efficiently support the incremental computation of streaming graphs. However, they are still inefficient for resolving the irregular state propagations in streaming graph processing.

6 CONCLUSION

This paper proposes the first programmable accelerator *TDGraph* to augment the many-core processor for high-performance streaming graph processing. TDGraph regularizes the state propagations for the vertices affected by graph updates along the graph topology and fully consolidates most accesses to the vertex states. By doing so, TDGraph reduces the useless updates of vertex states significantly and also gains much better data locality, thereby achieving higher utilization of the cores. The experimental results show that TDGraph improves the performance by up to 21.4 times over the cutting-edge software system with only 0.73% area cost.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their constructive comments. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper. This paper is supported by National Natural Science Foundation of China under grant No. 61832006, 61825202, 62072193, and 61929103. This work is also sponsored by Huawei Technologies Co., Ltd (No. YBN2021035018A5).

REFERENCES

[1] 2022. DDR4 SDRAM System Power Calculator. https://media-www.micron.com/-/media/client/global/documents/products/power-calculator/ddr4_power_calc.xlsm?rev=a8a5e30d8a7e41c4adcaad2df73934b4.
[2] 2022. macsim. https://github.com/gthparch/macsim.
[3] 2022. SNAP. http://snap.stanford.edu/data/index.html.
[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.
[5] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*. 39:1–39:11 pages.
[6] Sam Ainsworth and Timothy M. Jones. 2018. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 578–592.
[7] Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Transactions on Computer Systems* 36, 3 (2019), 8:1–8:34.
[8] Mikhail Asiatici and Paolo Ienne. 2021. Large-Scale Graph Processing on FPGAs with Caches for Thousands of Simultaneous Misses. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*. 609–622.
[9] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture*. 668–681.
[10] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture*. 373–386.
[11] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R. Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving Streaming Graph Processing Performance using Input Knowledge. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1036–1050.
[12] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM* 46, 5 (1999), 720–748.
[13] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*. 433–445.
[14] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th European Conference on Computer Systems*. 85–98.
[15] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing.

[16] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.

[17] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. 2018. Pixie: A System for Recommending 3+ Billion Items to 200+ Million Users in Real-Time. In *Proceedings of the 2018 World Wide Web Conference*. 1775–1784.

[18] Dhivya Eswaran, Christos Faloutsos, Sudipto Guha, and Nina Mishra. 2018. SpotLight: Detecting Anomalies in Streaming Graphs. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1378–1386.

[19] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*. 234–248.

[20] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental Graph Computations: Doable and Undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[21] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. 2021. Automating Incremental Graph Processing with Flexible Memoization. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1613–1625.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[23] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. 56:1–56:13.

[24] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the 9th European Conference on Computer Systems*. 1:1–1:14.

[25] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction. In *Proceedings of the 37th International Symposium on Computer Architecture*. 60–71.

[26] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the 16th European Conference on Computer Systems*. 17–32.

[27] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 284–296.

[28] Sang Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*. 411–424.

[29] Kevin M. Lepak and Mikko H. Lipasti. 2002. Temporally silent stores. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 30–41.

[30] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 177–187.

[31] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.

[32] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the 16th European Conference on Computer Systems*. 83–98.

[33] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the 14th EuroSys Conference 2019*. 25:1–25:16.

[34] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture9*. 116–128.

[35] Andrew McCrabb, Eric Winsor, and Valeria Bertacco. 2019. DREDGE: Dynamic Repartitioning during Dynamic Graph Execution. In *Proceedings of the 56th Annual Design Automation Conference*. 28.

[36] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sánchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 1–14.

[37] Anurag Mukkara, Nathan Beckmann, and Daniel Sánchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.

[38] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the ACM SIGOPS 24th Symposium on Operating Systems Principles*. 439–455.

[39] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*. 457–468.

[40] Quan M. Nguyen and Daniel Sánchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077.

[41] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven M. Burns, and Özcan Özturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*. 166–177.

[42] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[43] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture*. 908–921.

[44] Shafiur Rahman, Mahbod Afarin, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2021. JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1091–1105.

[45] Kenneth A. Ross. 2007. Efficient Hash Probes on Modern Processors. In *Proceedings of the 23rd International Conference on Data Engineering*. 1297–1301.

[46] Daniel Sánchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 475–486.

[47] David Sayce. 2020. The Number of tweets per day in 2020. https://www.dsayce.com/social-media/tweets-day/.

[48] Steven L. Scott. 1996. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 26–36.

[49] Albert Segura, Jose-Maria Arnau, and Antonio González. 2019. SCU: a GPU stream compaction unit for graph processing. In *Proceedings of the 46th International Symposium on Computer Architecture*. 424–435.

[50] Albert Segura, Jose-Maria Arnau, and Antonio Gonzalez. 2021. Energy-Efficient Stream Compaction Through Filtering and Coalescing Accesses in GPGPU Memory Partitions. *IEEE Trans. Comput.* (2021), 1–12. https://doi.org/10.1109/TC.2021.3104749

[51] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proceedings of the 22nd International Conference on Parallel and Distributed Computing*. 319–333.

[52] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate Streaming Graph Analysis through Preprocessing Buffered Updates. In *Proceedings of the 2018 ACM Symposium on Cloud Computing*. 301–312.

[53] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data*. 417–430.

[54] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.

[55] Avinash Sodani, Roger Gramunt, Jesús Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (2016), 34–46.

[56] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Helen Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture*. 531–543.

[57] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *Proceedings of the VLDB Endowment* 12, 2 (2018), 154–168.

[58] Yanwei Song and Engin Ipek. 2015. More is less: improving the energy efficiency of data movement via opportunistic use of sparse codes. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 242–254.

[59] Pourya Vaziri and Keval Vora. 2021. Controlling Memory Footprint of Stateful Streaming Graph Processing. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 269–283.

[60] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic Analysis of Evolving Graphs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 32:1–32:27.

[61] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings*

of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems. 237–251.

[62] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 194–204.

[63] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 615–628.

[64] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression In Irregular Applications. In *Proceedings of the 48th ACM/IEEE Annual International Symposium on Computer Architecture*. 1070–1082.

[65] Yifan Yang, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. GraphABCD: Scaling Out Graph Analytics with Asynchronous Block Coordinate Descent. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*. 419–432.

[66] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.

[67] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 593–607.

[68] Guowei Zhang, Virginia Chiu, and Daniel Sanchez. 2016. Exploiting Semantic Commutativity in Hardware Speculation. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. Article 34:1–34:12.

[69] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–25.

[70] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 608–621.

[71] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture*. 544–557.

[72] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[73] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 371–384.

[74] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, and Haikun Liu. 2021. LCCG: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. 45:1–45:14.

[75] Ruohuang Zheng and Sreepathi Pai. 2021. Efficient Execution of Graph Algorithms on CPU with SIMD Extensions. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. 262–276.

[76] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 712–725.