CISGraph: A Contribution-Driven Accelerator for Pairwise Streaming Graph Analytics

Songyu Feng[1,2], Mo Zou[1*] , Tian Zhi[1], Zidong Du[1]
[1]*State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
[2]*University of Chinese Academy of Sciences, Beijing, China*
{fengsongyu23s, zoumo, zhitian, duzidong}@ict.ac.cn

***Abstract*—Recent research observed that pairwise query is practical enough in real-world streaming graph analytics. Given a pair of distinct vertices, existing approaches coalesce or prune vertex activations to decrease computations. However, they still suffer from severe invalid computations because they ignore contribution variations in graph updates, hindering performance improvement. In this work, we propose to enhance pairwise analytics by taking updates contributions into account. We first identify that graph updates from one batch have a distinct impact on query results and experience obvious diverse computation overheads. We then introduce CISGraph, a novel *Contribution-driven* pairwise accelerator with valuable updates *Identification* and *Scheduling*. Specifically, inspired by triangle inequality, CISGraph categorizes graph updates into three levels according to contributions, prioritizes valuable updates, delays possible-valuable updates, and drops useless updates to eliminate wasteful computations. As far as we know, CISGraph is the first hardware accelerator that supports efficient pairwise queries on streaming graphs. Experimental results show that CISGraph substantially outperforms state-of-the-art streaming graph processing systems by 25× on average in response time.**

***Index Terms*—pairwise query, streaming graph processing, domain-specific architecture**

I. Introduction

Graphs exhibit powerful representation capacity in neural networks and recommendation systems [18]. Real-world graphs evolve continuously as the topology structure and graph state are modified by updates over time, called streaming graphs. To obtain the latest results, streaming graph processing usually performs incremental computation [17], which broadcasts new states of affected vertices along the topology until convergence instead of re-computing the whole graph from its initial state. In addition, recent advances [3], [13] have observed that answering point-to-point pairwise queries is practical enough rather than heavy and expensive one-to-all computations.

State-of-the-art pairwise query techniques focus on reducing wasteful vertex activations during propagation. For example, PnP [20] estimates an upper bound while SGraph [3] maintains both upper and lower bounds for each vertex. They prune vertices whose state exceeds the bounds to avoid unnecessary activations and state propagations. TDGraph [22] and JetStream [17] optimize one-to-all streaming graph analytics by coalescing multiple vertex activations once they target the same vertex. The key idea behind the design that reducing activations can be extended to pairwise queries as well.

* Corresponding author: Mo Zou, Email: zoumo@ict.ac.cn.

Although existing techniques are well-designed in throughput improvement, they still suffer from long-period response time due to a large amount of invalid computations. In particular, for efficiency, updates are applied to graphs in batches where each batch contains millions of edge additions and deletions. Previous work treats all updates the same and outputs a result only after all updates from one batch are finished. On the contrary, we notice that only a small fraction of updates, called valuable updates, from one batch contribute to the converged result, motivating us to minimize computations by skipping useless updates and fully utilizing on-chip resources.

However, irregular graph structure poses several challenges in updates classification. First, incremental computation propagates the states of affected vertices along the irregular paths, making it challenging to predict whether a graph update will contribute to the final result. Moreover, streaming graphs vary over time, making it impossible to discover a stable classification decision satisfying all snapshots. Finally, reusing previous states directly in edge deletions may produce an incorrect result, especially in monotonic algorithms.

To address the above challenges, we propose CISGraph, a novel high throughput accelerator for pairwise streaming graph analytics. CISGraph augments existing incremental computations to 1) prioritize valuable updates to lower response time, 2) drop useless updates to eliminate wasteful computations, and 3) delay other updates to guarantee correctness in edge deletions. Inspired by triangle inequality, given a pairwise query $Q(s \rightarrow d)$, we simplify the identification problem as whether the updated edge $u \rightarrow v$ impacts the convergent state of v. Taking Point-to-Point Shortest Path (PPSP) as an example, if the sum of u state (i.e., converged shortest path from s to u) and the added edge weight is less than v state, CISGraph identifies it as valuable otherwise as useless. Furthermore, when processing edge deletions, CISGraph assigns valuable updates as delayed when u is not located on the shortest path. Such an update does not have contributions but may impact future convergence. CISGraph schedules valuable updates in a preemptive and answers the query after all valuable updates are finished.

In summary, our main contributions are as follows:

- We identify the problem of invalid computations and serious overheads when incremental computation propagates updates in a contribution-independent way.
- We propose an effective contribution-aware incremental workflow to identify and prioritize valuable updates at

978-3-9826741-0-0/DATE25/© 2025 EDAA

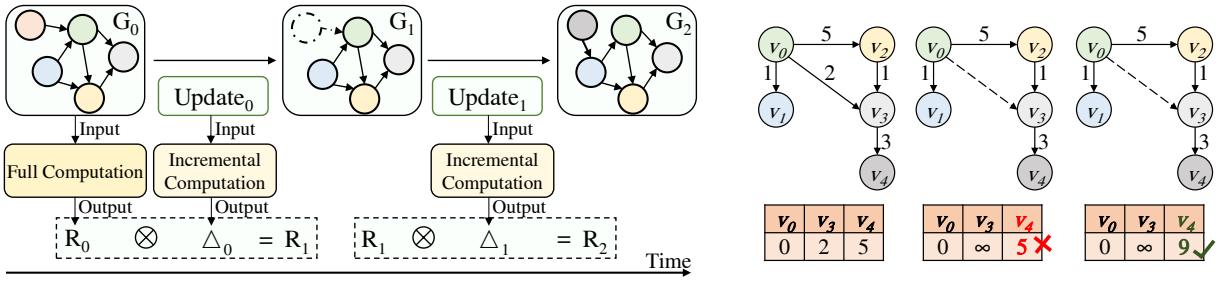


Fig. 1. Illustration of (a) incremental computation in streaming graph analytics and (b) incorrect results caused by directly reusing previous states in PPSP.

runtime with minimal overhead, eliminating wasteful computations and reducing response time.

- We introduce a cycle-accurate accelerator CISGraph to support our contribution-aware workflow. The experimental result shows that CISGraph achieves 25× speedup on average over the state-of-the-art software design by accurate valuable updates identification and significant computation reduction.

II. Background and Motivation

We first introduce the basic notion in pairwise streaming graph analytics and analyze the existing optimizations; then, we discuss and demonstrate the shortcomings inherent in current approaches, motivating the need for enhanced dataflow and architecture.

A. Streaming Graph Analytics

Streaming graph analytics evaluates a timely query result on the latest snapshot. The typical workflow of streaming graph analytics [2], [10], [15] usually applies graph updates in batches and employs an incremental computation model for efficiency. As Figure 1(a) shows, the initial graph G_0 performs full computation just as static graph processing to obtain an initial converged state R_0. Then incremental computation buffers the continuous arriving updates until reaching an assigned threshold (e.g., 100K in our evaluation). After that, instead of re-starting from scratch, incremental computation reads updates in batch $Update_0$ and formulates changed states Δ_0 for affected vertices along topology. Incremental computation performs $R_0 \bigotimes \Delta_0$ to obtain result R_1 for snapshot G_1, where $\bigotimes$ is algorithm-specified. Note that we simulate graph updates as edge additions and deletions since vertex additions and deletions can be transformed into a series of edge updates. In the rest of the paper, we define a vertex in converged state if its state is not modified by upcoming updates. Otherwise, it is not converged and the updates are valuable.

Monotonic algorithms are practical in streaming graph analytics [20].These algorithms update the vertex states by selecting an extreme value (max or min) from the neighboring vertices. Due to monotonic attributes, edge deletions exhibit risks falling into unrecoverable approximation [17], [22]. When evaluating the shortest path from v_0 to v_4, Figure 1(b) demonstrates the incorrect instance of an edge deletion $v_0 \rightarrow v_3$. After reading the edge deletion, incremental computation resets the state of vertex v_3 as ∞ because v_0 cannot reach v_3 anymore. Then incremental computation propagates the new state of v_3 along the path. However, v_4 will not converge from the current state (i.e., 5) to the correct state (i.e., 9) because incremental computation updates v_4 only when it receives a shorter distance. On the contrary, edge additions are always safe because the added edge constricts results or leaves the results unchanged. In this paper, we concentrate on monotonic algorithms.

To ensure the correctness of incremental computation in monotonic algorithms, existing approaches refine the affected vertices to a recoverable state for re-computation. GraphFly [2] traverses graph topology originated from deleted edges and resets all reachable vertices to initial states (e.g., ∞ in PPSP), enabling safe monotonic convergence. GraphBolt [16] identifies and handles the impacted vertices synchronously, increasing processing throughput. KickStarter [19] tags a vertex only when one of its in-neighbors is tagged and that neighbor contributes to its state, trimming the scale of tagged vertices. ACGraph [11] maintains dependency trees to encode graph structure, processing tagged vertices in a normalized top-to-bottom order. However, they usually require an additional tagging traverse before incremental computation, leading to serious time and resource overheads.

To improve performance, TDGraph [22] and JetSream [17] are popular hardware-based designs on streaming graph optimizations. TDGraph propagates a vertex state only when aggregating changed values from all in-neighbors, minimizing redundant accesses and computations. JetStream encodes graph updates into events, coalesces multiple events once they target the same vertex, and applies the merged state value to out-degree neighbors together. However, they focus on one-to-all streaming graph analytics, which is expensive in an exhaustive nature in real-world scenarios.

B. Pairwise Query

Pairwise query answers the result for an assigned pair of vertices $Q(s \rightarrow d)$. For example, a practical navigation system is interested in finding the shortest path from home (i.e., s) to company (i.e., d) instead of from home to arbitrary locations. To improve performance, PnP [20] and SGraph [3] are two software frameworks trying to minimize activated vertices according to bound-based pruning. PnP estimates an upper bound for each vertex and prunes any vertex that exceeds the bound during propagation. SGraph chooses hub vertices as a medium, maintaining an upper bound and predicting a lower bound for each vertex for pruning. However, our results show that

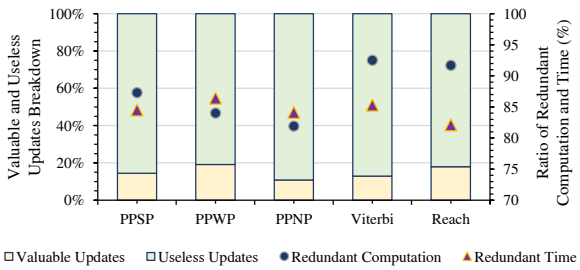


Fig. 2. Breakdown of graph updates, ratio of redundant computations, and wasteful processing time.

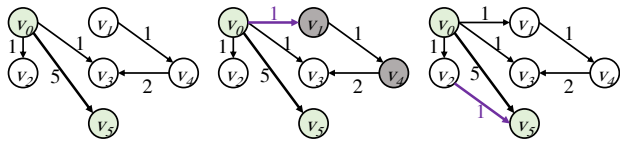


Fig. 3. An example to illustrate useless edge addition $v_0 \rightarrow v_1$ and valuable edge addition $v_2 \rightarrow v_5$.

the speedup of the prediction-based approach exhibits a large degree of randomness, leaving optimization room. Moreover, they give the latest results after finishing all updates in one batch. On the contrary, we observe that only a small fraction of updates contribute to the new result, motivating us to identify them before propagation.

C. Problem: Redundant Computations

In pairwise streaming graph analytics, incremental computation broadcasts new or delta states originating from graph updates along the topology to refine the affected vertices. However, since the pairwise query relies on a single path, updates may not change the converged path and thus do not impact the result. As a result, sequential update processing incurs redundant computational overhead.

To illustrate the problem, we evaluate the percentage of useless graph updates and associated overhead in a batch. Useless updates do not affect the final results, making all computations and time consumed on them are redundant. Section IV-A details the configuration and batch generation. Figure 2 presents the result from orkut dataset. On average, across 10 different pairwise queries, 85% of all updates are useless, leading to 87% redundant computations. The state propagation of these redundant computations consumes over 84% of the total processing time. Furthermore, edge deletions result in more wasteful computations than edge additions, as deletions require additional tagging of affected vertices along the topology to ensure correctness, leading to significant computational overhead.

We clarify the reason using the example in Figure 3. The initial shortest path for pairwise query $Q(v_0 \rightarrow v_5)$ is 5, relaying on path v_0-v_5. When processing edge addition $v_0 \rightarrow v_1$, incremental computation updates v_1 from ∞ to 1 and activates v_1 for propagation. In the next iteration, v_1 broadcasters its new state to v_4 and updates v_4 to 2. However, v_4 cannot reach pairwise query destination v_5, making the computations to v_1 and v_4 invalid. Such an update is useless without any contributions to $Q(v_0 \rightarrow v_5)$. In contrast, edge addition $v_2 \rightarrow v_5$ is valuable, as it shorts $Q(v_0 \rightarrow v_5)$ from previous result of 5 to timely result of 2. However, as calculated before, conventional incremental computation spent most time and hardware resources on useless updates, prompting us to enhance the workflow by identifying valuable updates and eliminating redundant computations.

III. Workflow and Architecture

A. Contribution-Aware Workflow

Triangle inequality [3], [9] is a geometry concept, defining that in a given triangle, the length of any side is less than or equal to the sum of the other two sides. This principle can be applied to pairwise queries in a streaming graph. Taking PPSP in Figure 3 as an example, after adding edge $v_2 \rightarrow v_5$ in the right snapshot, v_0, v_2, and v_5 make up a triangle. Then we can get the following equation:

$$Dist(v_0, v_2) + Dist(v_2, v_5) \geq Dist(v_0, v_5), \quad (1)$$

where *Dist(a, b)* records the shortest path from *a* to *b*. In this example, after applying $v_2 \rightarrow v_5$, *Dist*(v_0, v_5) is 2, which equals the sum of *Dist*(v_0, v_2) and *Dist*(v_2, v_5), meaning that v_0-v_2-v_5 is the shortest path for *Q*(v_0, v_5), named global key path. Otherwise, if *Dist*(v_0, v_2) plus *Dist*(v_2, v_5) is larger than *Dist*(v_0, v_5), there must exists another shortest key path from v_0 to v_5. If the sum of *Dist*(v_0, v_2) and *Dist*(v_2, v_5) is smaller than *Dist*(v_0, v_5), PPSP algorithm will choose v_0-v_2-v_5 as the shortest path, which states the fact that $Dist(v_0, v_2) + Dist(v_2, v_5) \geq Dist(v_0, v_5)$.

Based on triangle inequality, we classify an edge update $u \rightarrow v$ as valuable if the converged state of v is changed due to the update. On the one hand, if an edge update violates triangle inequality, it will not influence *Q(s, d)* because a shorter path from *s* to *d* exists. On the other hand, if an edge update changes v state (i.e., *Dist(s, v)* is updated), we propagate v state along the topology. Unlike SGraph and PnP prune vertex activations during propagation, we classify graph updates and only process

Algorithm 1 Updates Classification for PPSP.

```
1:  Input: Converged State Array state and Update Batch batch
2:  Output: Valuable Set VS and Delayed Set DS
3:  for each addition u --weight--> v ∈ batch do
4:      if state[u] + weight < state[v] then
5:          VS.append(u→v)
6:      else
7:          drop u→v
8:      end if
9:  end for
10: for each deletion u --weight--> v ∈ batch do
11:     if state[u] + weight == state[v] then
12:         if u on global key path then
13:             VS.prepend(u→v)
14:         else
15:             VS.append(u→v)
16:         end if
17:     else
18:         drop u→v
19:     end if
20: end for
```

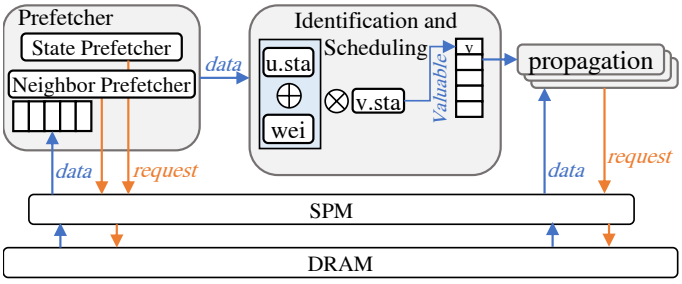


Fig. 4. Architecture of CISGraph

valuable updates, which minimizes wasteful computations as much as possible.

Moreover, valuable edge deletions can be further identified as delayed and non-delayed updates. A valuable edge deletion $u \rightarrow v$ changes v state but both u is not located on the global key path, meaning that although $u \rightarrow v$ is valuable but $Q(s, d)$ relies on another existing path. In this case, $u \rightarrow v$ is classified as delayed. If $u \rightarrow v$ is valuable and u is located on the global key path, $u \rightarrow v$ is identified as non-delayed (valuable) and processed preemptively. Algorithm 1 summarizes the workflow. We drop all useless updates to eliminate redundant computations. All delayed updates are processed in their coming sequential. When detecting a valuable update, we assign it the highest priority and propagate it first. Currently, we focus on single-query scenarios and leave the study of multi-query cases in future work.

Although the above workflow can be implemented on CPUs and receives performance improvement over existing solutions (more details in Section IV-B), we find that it still suffers from inefficient pipelines. CPU requires additional instructions to finish identification, propagation, and workload scheduling. Moreover, instruction level parallelism is low because of topology dependency. To this end, we implement an accelerator CISGraph to exploit the full benefits of the workflow.

B. Hardware Design

Figure 4 presents the overall architecture of CISGraph, containing three main phases: *Prefetching* phase fetches data from memory to scratchpad memory (SPM) for reuse opportunity, *Identification and Scheduling* phase allocates resources for valuable updates, *Propagation* phase broadcasts new states of affected vertices for convergence. The workflow of CISGraph can be summarized as follows. When obtaining a batch of graph updates, CISGraph modified graph topology according to edge additions and deletions to generate a snapshot. Then CISGraph distributes update $u \rightarrow v$ to the ith pipeline where $i = v \mod pipelines$. *Prefetching* phase requests necessary data from memory, like u and v states, for updates identification. *Identifying and Scheduling* phase checks triangle inequality for each update and identifies valuable updates. Only valuable and delayed updates propagate new states along the topology. During this phase, CISGraph schedules valuable updates before delayed updates to reduce response time. To broadcast the new state and reduce hardware cost, CISGraph reuses hardware modules in *Prefetcher* phase to fetch out-neighbor states and edge weights. *Propagation* follows algorithm principles, i.e., sum and min in PPSP, to obtain a converged result.

Prefetcher modules include state and neighbor prefetcher modules to decouple data dependency in graph analytics. Because Compressed Sparse Row (CSR) stores neighbor IDs and weights continuously in memory, CISGraph requires one memory access, specifying the start address and request length, to fetch the whole edge list of one vertex from memory to SPM. After that, neighbor prefetcher module accesses SPM for neighbor IDs and edge weights. In contrast, fine-grained and random state requests are driven by neighbor IDs coming from neighbor prefetcher. So CISGraph implements state and neighbor prefetcher modules separately to fetch state value and neighbor data for one vertex in parallel.

Identification and scheduling modules maintain a buffer to store the indices and new states of the vertices affected by edge updates. The identification module reads weight for updated edge $u \rightarrow v$ from the batch and receives previous converged states for u and v from prefetcher modules. For edge additions, the identifying module performs algorithm-specific operations taking weight, u.state, and v.state as a triangle to find valuable updates. CISGraph appends all valuable edge additions to the end of the buffer, waiting for propagation. For edge deletions, besides the above comparisons, CISGraph classifies valuable updates as delayed and non-delayed according to whether u is located on the global key path or not. In PPSP, if u.state plus weight equal to v.state, this is a non-delayed update because u is on the global key path, otherwise a delayed update. For simplification, CISGraph inserts valuable updates at the beginning of the output buffer and appends delayed updates at the end of the output buffer, ensuring valuable updates have a high priority. CISGraph overlaps the processing of delayed updates with updates gathering to reduce response time further. When all updates are categorized and added to the buffer, CISGraph can give a converged answer once no valuable update exists in the output buffer.

Propagation modules broadcast new states of valuable updates to downstream vertices following algorithm-specific principles. For efficiency, CISGraph propagates vertices in two steps. In each cycle, propagation modules fetch a valuable update $u \rightarrow v$ from the output buffer. In the first step, the propagation module traverses out-neighbors of v and computes candidate states for v out-neighbors. This computation depends on the convergence rule for graph algorithms, e.g., PPSP adopts sum operation and PPWP performs min operation. Then in the second step, the propagation module selects a new state between the candidate and the previous converged state for v out-neighbors, which depends on the convergence rule as well. For example, PPSP selects the minimal state between the two options and PPWP updates the vertex by the max one. If a vertex's new state differs from the previous converged state, the vertex is activated and its new value is written to SPM. SPM is organized as cache to enable evictions. To offset the speed gap between identification and propagation, CISGraph adds multiple propagation modules to finish the above operations. To alleviate workload imbalance, all propagation modules adds activated vertices into a global buffer. In the next iteration, CISGraph distributes activated vertices based on their IDs to propagation modules.

TABLE I
EXPERIMENTAL CONFIGURATIONS.

	Software Framework	CISGraph
Compute	4× Intel(R) Xeon(R) Gold 6254 CPU	4× CISGraph Pipelines
Unit	@3.10GHz	@1GHz
On-chip	2MB L1, 32MB L2	32MB eDRAM scratchpad
Memory	99MB LLC	@ 2GHz, 0.8ns latency
Off-chip	8× DDR4-3200	8× DDR4-3200
Memory	12GB/s channel	12GB/s channel

TABLE II
FIVE MONOTONIC GRAPH ALGORITHMS IN OUR EVALUATION. $\oplus$ AND $\otimes$ FOR $u \xrightarrow{w} v$.

Algorithm	$\oplus$	$\otimes$
PPSP	T = u.state + w	MIN(T, v.state)
PPWP	T = min(u.state, w)	MAX(T, v.state)
PPNP	T = max(u.state, w)	MIN(T, v.state)
Viterbi	T = u.state / w	MAX(T, v.state)
Reach	T = u.state	MAX(T, v.state)

TABLE III
REAL WORLD GRAPH DATASETS.

Graphs	Abbreviation	#Vertices	#Edges	Average Degree
Orkut	OR	2599558	41631643	16
Livejournal	LJ	4846610	68475391	14
UK-2002	UK	18483187	261787258	14

IV. EVALUATION

A. Experimental Setup

Methodology. To measure execution latency precisely, we develop a customized cycle-accurate simulator CISGraph. The simulator models the detailed behaviors and timing constraints of each module and follows the inter-module workflow described in Section III. Given the memory-intensive nature of graph processing, we incorporate a cycle-accurate scratchpad memory model for on-chip memory accesses, using CACTI 6.5 [12] to configure access latency. Additionally, CISGraph is integrated with DRAMSim3 [14] to model off-chip memory accesses accurately. Table I shows the hardware parameters of the evaluated CISGraph implementation.

Baselines. We compare CISGraph against SGraph [3] and a Cold-Start baseline. SGraph is a state-of-the-art pairwise query processing system. It prunes vertices whose new state falls outside the upper and lower bounds. To determine the two bounds, SGraph maintains the distance of each vertex to a set of hub vertices (i.e., 16 vertices with the highest degree) and updates distances during execution. The Cold-Start baseline, on the other hand, performs a full computation from the initial state for each snapshot to obtain timely results. For fairness, we compare both SGraph and Cold-Start with the software-only CISGraph-O, as well as hardware-software co-designed CISGraph. CISGraph-O runs on the same platform as SGraph and Cold-Start baseline. CISGraph supports CISGraph-O. Table I outlines the platform configuration for software frameworks, i.e., SGraph, Cold-Start, and CISGraph-O.

Algorithms. We evaluate CISGraph on five monotonic graph algorithms: Point-to-Point Shortest Path (PPSP), Point-to-Point Widest Path (PPWP), Point-to-Point Narrowest Path (PPNP), Reachability (Reach), and Viterbi algorithm (Viterbi). All algorithms are monotonic and accept a pair of distinct vertices as source and destination. PPSP, PPWP, and PPNP answer the shortest, widest, and narrowest distance between assigned source and destination, respectively. Reach outputs whether the assigned source and destination are reachable following a breadth-first-search scheme. Viterbi searches the most likely path in a graph with probabilistic transitions. Table II describes $\oplus$ and $\otimes$ principles during identification for the five algorithms.

Datasets and Batches. Table III lists the scale of three graph datasets used in our evaluation. To eliminate the impact of topological differences, we randomly select 10 pairs of vertices for pairwise query and measure the average performance. To generate batches and model streaming graphs, referring to existing works [3], [17], [22], we load 50% of the edges in the graph dataset as an initial snapshot. Then we randomly select the remaining edges to model edge additions and sample the loaded edges to model edge deletions. We generate batches containing 50K edge additions and 50K edge deletions. Note that we classify edge additions into valuable and useless while edge deletions into valuable, delayed, and useless to response in advance. Only after finishing all valuable edge additions, CISGraph starts edge deletions for fairness.

B. Performance Comparison

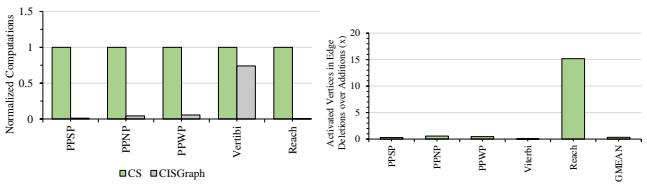
Overall Performance. Table IV demonstrates the speedup of SGraph, CISGraph-O, and CISGraph over CS baseline. The execution time in CISGraph includes the propagation phase and identification phase. The propagation phase broadcasts delta states along the graph topology oriented from affected vertices and the identification phase filters valuable updates. We can make the following key observations. First, CISGraph-O achieves 1.6× to 91× speedup and 16.6× on average over CS. This is because CS cannot reuse previous results, leading to a large amount of useless propagation. On the contrary, CISGraph-O filters updates first, guaranteeing correctness with minimal computation overhead. Second, CISGraph-O outperforms SGraph by 0.6× to 45.7× and 6.7× on average. Although SGraph prunes vertices, it selects a group of hub vertices as agents to determine the upper and lower bounds. Such a boundary may not be accurate. In the best case, SGraph converges quickly and performs better than CISGraph-O. However, in the worst case, due to inaccurate agent selection, SGraph cannot prune any vertices because all vertices on the path fall within the boundary. Finally, CISGraph outperforms SGraph and CISGraph-O by 0.93× to 366× and 0.96× to 13.4×, respectively. The performance improvement comes not only from the accurate updates identification, which avoids redundant and wasteful computations but also from efficient hardware pipelines, which provide the opportunity to give an accurate answer before all updates are finished.

Computation Reduction. To explain the performance improvements, we count computations in CISGraph and CS using OR dataset and show the result in Figure 5(a). Since CS starts from the initial states without any reuse, it produces most computations. On average, CISGraph reduces computations by

TABLE IV
EXECUTION SPEEDUP OF SGRAPH, CISGRAPH-O, AND CISGRAPH. NORMALIZED TO CS BASELINE.

		OR	UK	LJ	GMean
PPSP	CS	1×	1×	1×	1×
	SGraph	7.7×	13.7×	3×	6.7×
	CISGraph-O	9.7×	26.3×	20.4×	17.4×
	CISGraph	18.7×	95.6×	241.6×	75.6×
PPWP	CS	1×	1×	1×	1×
	SGraph	81.2×	20.8×	1.4×	13.2×
	CISGraph-O	207.6×	69.5×	62.8×	96.7×
	CISGraph	1073×	331.9×	153.4×	379.5×
PPNP	CS	1×	1×	1×	1×
	SGraph	9.3×	0.24×	0.9×	1.3×
	CISGraph-O	10.2×	18.3×	16.2×	14.5×
	CISGraph	9.8×	87.9×	218×	57.3×
Viterbi	CS	1×	1×	1×	1×
	SGraph	2.7×	2×	1.3×	1.9×
	CISGraph-O	1.7×	91×	1.6×	6.2×
	CISGraph	2.5×	602.9×	8.6×	23.4×
Reach	CS	1×	1×	1×	1×
	SGraph	0.4×	0.6×	0.4×	0.4×
	CISGraph-O	5.9×	9.4×	10.7×	8.4×
	CISGraph	6.1×	44.2×	63.7×	25.8×

67% over CS. Moreover, we have an interesting discovery that reduction computations in SGraph vary between different algorithms and point-to-point pairs. In PPNP and Reach, SGraph reduces computations significantly, however, it spends much time on boundary maintaining, offsetting the performance gains. Considering a specific algorithm, SGraph varies significantly between different PnP pairs. In the best case, SGraph stops propagating in less than three hops from the source vertex while in the worst case, SGraph has to activate all vertices because of inaccurate boundaries. In that case, SGraph performs worse than CS due to additional time spent on bounds maintenance. Unlike SGraph, CISGraph identifies useless updates before applying, receiving consistent performance improvement.



(a) (b)

Fig. 5. Illustration of (a) computations in CISGraph and CS, normalized to CS and (b) activated vertex number of edge deletions over edge additions.

Sensitivity Study to Edge Additions and Edge Deletions. When processing edge deletions, previous works usually require one more traversal to tag affected vertices for correct results. Unlike them, CISGraph uses triangle inequality to identify valuable and delayed updates for edge deletions, avoiding the additional traversal. Moreover, CISGraph classifies valuable edge deletions as non-delayed and delayed, answering the pairwise query after all non-delayed updates are finished. Figure 5(b) compares the activation vertices to process 50K edge additions and 50K edge deletions. Although CISGraph activates more vertices in deletions than additions on Vertibi, on average, CISGraph activates 2.92× vertices for edge additions over edge deletions, proving that CISGraph activates fewer vertices for edge deletions than edge additions before the response.

V. RELATED WORK

A number of software and hardware techniques are proposed to accelerate streaming graphs. KineoGraph [4] includes an incremental computation engine to analyze streaming data but only supports growing graphs. GraphBolt [16], GraphFly [2], KickStarter [19], and ACGraph [11] track dependencies to capture vertices affected by edge deletions. Layph [21] constraints state propagation in a skeleton by creating a multi-layer graph. ABR [1] reorders updates in each batch to reduce dependency. STINGER [6] and Aspen [5] are streaming graph representations considering high-throughput graph updates. TDGraph [22] takes vertices influenced by updates as roots and preloads neighbors along graph topology. Meanwhile, TDgraph synchronizes incremental updates during computation. JetStream [17] treats updates as events and handles monotonic graph algorithms through an event-driven computation model. Both TDGraph and JetStream coalesce updates to reduce memory accesses and redundant computations. However, they focus on on-to-all queries, leaving a significant amount of wasteful computations on pairwise queries.

Unlike streaming graph processing, evolving graph processing captures historical information from a series of previous snapshots. TEGRA [8] proposes an iterative incremental computation model and in-memory intermediate state to enable efficient evolving graph operations. MEGA [7] applies a batch-oriented execution that schedules updates to enable concurrent snapshot computation.

VI. CONCLUSION

This paper proposes CISGraph, the first pairwise streaming graph accelerator. CISGraph classifies updates into different levels and processes updates that will contribute to the final result in priority, greatly narrowing response time. CISGraph delays updates that may contribute to future results to obtain correct convergent results. CISGraph prunes update that will not contribute to the final result to avoid wasteful computations. Experimental results show that CISGraph speeds up the execution by 25× over state-of-the-art software frameworks.

VII. ACKNOWLEDGMENT

This work is partially supported by the National Key R&D Program of China (under Grant 2022YFB4501600), the NSF of China (under Grants U20A20227, 62222214, 62341411, U22A2028, 61925208, 62102398, 62102399, 62302478, 62302482, 62302483, 62302480, 62302481), Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB0660200, XDB0660201, XDB0660202), CAS Project for Young Scientists in Basic Research (YSBR-029), Youth Innovation Promotion Association CAS, Xplore Prize, and the 2022 Fundamental Disciplines Top Quality Student Training Program 2.0 Project.

REFERENCES

[1] A. Basak, Z. Qu, J. Lin, A. R. Alameldeen, Z. Chishti, Y. Ding, and Y. Xie, "Improving Streaming Graph Processing Performance using Input Knowledge," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1036–1050.

[2] D. Chen, C. Gui, Y. Zhang, H. Jin, L. Zheng, Y. Huang, and X. Liao, "GraphFly: Efficient Asynchronous Streaming Graphs Processing via Dependency-Flow," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–14.

[3] H. Chen, M. Zhang, K. Yang, K. Chen, A. Zomaya, Y. Wu, and X. Qian, "Achieving Sub-second Pairwise Query over Evolving Graphs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 1–15.

[4] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the Pulse of a Fast-Changing and Connected World," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 85–98.

[5] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-Latency Graph Streaming using Compressed Purely-Functional Trees," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, 2019, pp. 918–934.

[6] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High Performance Data Structure for Streaming Graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[7] C. Gao, M. Afarin, S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Mega Evolving Graph Accelerator," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 310–323.

[8] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 337–355.

[9] X. Jiang, M. Afarin, Z. Zhao, N. Abu-Ghazaleh, and R. Gupta, "Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 18–32.

[10] X. Jiang, C. Xu, X. Yin, Z. Zhao, and R. Gupta, "Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 17–32.

[11] Z. Jiang, F. Mao, Y. Guo, X. Liu, H. Liu, X. Liao, H. Jin, and W. Zhang, "ACGraph: Accelerating Streaming Graph Processing via Dependence Hierarchy," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[12] H. Labs. CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. [Online]. Available: https://arch.cs.utah.edu/cacti/

[13] D. Li, H. Qu, and J. Wang, "A Survey on Knowledge Graph-Based Recommender Systems," in *2023 China Automation Congress (CAC)*. IEEE, 2023, pp. 2925–2930.

[14] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[15] M. Mariappan, J. Che, and K. Vora, "DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs," in *Proceedings of the sixteenth European conference on computer systems*, 2021, pp. 83–98.

[16] M. Mariappan and K. Vora, "GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[17] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1091–1105.

[18] K. Sharma, Y.-C. Lee, S. Nambi, A. Salian, S. Shah, S.-W. Kim, and S. Kumar, "A Survey of Graph Neural Networks for Social Recommender Systems," *ACM Computing Surveys*, vol. 56, no. 10, pp. 1–34, 2024.

[19] K. Vora, R. Gupta, and G. Xu, "Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations," in *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, 2017, pp. 237–251.

[20] C. Xu, K. Vora, and R. Gupta, "PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 587–600.

[21] S. Yu, S. Gong, Y. Zhang, W. Yu, Q. Yin, C. Tian, Q. Tao, Y. Yan, G. Yu, and J. Zhou, "Layph: Making Change Propagation Constraint in Incremental Graph Processing by Layering Graph," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2766–2779.

[22] J. Zhao, Y. Yang, Y. Zhang, X. Liao, L. Gu, L. He, B. He, H. Jin, H. Liu, X. Jiang *et al.*, "TDGraph: A Topology-Driven Accelerator for High-Performance Streaming Graph Processing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 116–129.