JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator

Shafiur Rahman
mrahm008@ucr.edu
University of California, Riverside
Riverside, California, USA

Mahbod Afarin
mafar001@ucr.edu
University of California, Riverside
Riverside, California, USA

Nael Abu-Ghazaleh
nael@cs.ucr.edu
University of California, Riverside
Riverside, California, USA

Rajiv Gupta
gupta@cs.ucr.edu
University of California, Riverside
Riverside, California, USA

ABSTRACT

Graph Processing is at the core of many critical emerging workloads operating on unstructured data, including social network analysis, bioinformatics, and many others. Many applications operate on graphs that are constantly changing, i.e., new nodes and edges are added or removed over time. In this paper, we present JetStream, a hardware accelerator for evaluating queries over streaming graphs and capable of handling additions, deletions, and updates of edges. JetStream extends a recently proposed event-based accelerator for graph workloads to support streaming updates. It handles both accumulative and monotonic graph algorithms via an event-driven computation model that limits accesses to a smaller subset of the graph vertices, efficiently reuses the prior query results to eliminate redundancy, and optimizes the memory access pattern for enhanced memory bandwidth utilization. To the best of our knowledge, JetStream is the first graph accelerator that supports streaming graphs, reducing the computation time by 90% compared with cold-start computation using an existing accelerator. In addition, JetStream achieves about 18× speedup over KickStarter and GraphBolt software frameworks at the large baseline batch sizes that these systems use with significantly higher speedup at smaller batch sizes.

CCS CONCEPTS

• **Computer systems organization → Data flow architectures**.

KEYWORDS

streaming graphs, incremental algorithms, accelerators

ACM Reference Format:
Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. 2021. JetStream: Graph Analytics on Streaming Data with Event-Driven Hardware Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480126

This work is licensed under a Creative Commons Attribution International 4.0 License.

MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480126

1 INTRODUCTION

Graphs are used to represent data in many application domains because of their ability to represent entities (vertices) and relationships (edges). Real-world graphs such as social networks and web graphs are often massive and irregular, making it challenging to achieve good performance on graph analytics workloads. There has been substantial research to develop graph analytics frameworks that deliver high performance on shared-memory [31, 39] and distributed platforms [7, 23]. Recent research has also used GPUs [47] and custom accelerators [1, 13, 32, 33].

Most graph frameworks optimize the performance of a given query against a fixed graph. However, in many real-world applications, we are faced with the *streaming graph* scenario where the graph is constantly changing as new entities are created, old entities are removed, and new interactions take place over time. A stream of updates in the form of edge/vertex additions/deletions is typically applied to the graph in batches for efficiency. As the graph evolves, a straightforward approach is to restart the query from scratch after applying a batch of graph updates. However, the number of vertices or edges modified in a batch is typically exceedingly small relative to the size of the graph. Thus, as the changes only modify a small subset of the graph for many queries, much of the computation performed during reevaluation is redundant.

To address this inefficiency, streaming graph systems support incremental update of query results following changes to the graph, resulting in order of magnitudes speedups over restarting the query. Examples of such software systems include Kineograph [8], Tornado [38], and Naiad [29] that can handle only growing graphs (i.e., no deletions are allowed). By far, the problem of incrementally supporting deletions is more challenging, and only KickStarter [45], Graphbolt [26], and DZig [25] support it.

JetStream builds on a recent accelerator (GraphPulse [33]) which uses an event-driven asynchronous processing model, with reported speedups of up to 6× relative to BSP-based accelerator (Graphicionado [13]). The event-driven model naturally supports asynchronous graph processing with faster convergence via greater parallelism, reduced work, and elimination of synchronization at iteration boundaries. In addition to its state-of-the-art performance, we chose GraphPulse because it maps incremental update operations to a series of events naturally within the existing architecture. JetStream supports all algorithms compatible with delta-accumulative computation [50], as is the case in GraphPulse.

The addition of edges is straightforward in the event-driven model; the added edge simply creates a new event. In contrast, edge deletion is substantially more difficult for most algorithms because it is often impossible to determine whether an update should propagate. We support deletions in two phases: (1) incrementally transforming query results for the previous version of the graph into a *recoverable state* for the updated graph, and (2) bringing the results to convergence again. Although GraphBolt and KickStarter also proceed in two phases, they rely on the Bulk Synchronous Processing (BSP), model which cannot work in JetStream's asynchronous model. Therefore, we develop new event-based algorithms where both phases execute in a fully asynchronous fashion. JetStream serves both the class of accumulative algorithms supported by GraphBolt and monotonic algorithms supported by KickStarter.

The JetStream design leverages the coalescing queue (a vital component that enables combining events destined to the same vertex) from GraphPulse to accelerate streaming by eliminating key inefficiencies of software streaming frameworks such as KickStarter. When concurrently processing a batch of deletes, KickStarter performs many random reads and relies on atomic operations to reset the vertex values to a *recoverable* state. JetStream eliminates the above sources of inefficiency by having events carry the update contributions and using coalescing to achieve faster convergence without requiring atomic operations (as delete events to the same vertex are coalesced). We also leverage asynchronous processing to overlap different operations such as edge insertions, re-approximation of states after delete, and initial query for better efficiency. We introduce additional optimizations that limit the propagation of delete events when they are determined to be unnecessary, further improving performance. JetStream achieves on average 18× improvement over state-of-the-art streaming graph software. Furthermore, JetStream outperforms GraphPulse using cold-restart by a factor of 13× on streaming queries, an advantage that grows for smaller batch sizes. Lowering the overhead to this level brings us closer to achieving real-time streaming operation where graphs are updated on the fly since we do not need to aggregate updates into large batches to amortize query evaluation costs.

The key contributions of this paper are as follows:

- First Streaming Graph Accelerator: JetStream is the first accelerator to support operations on streaming graphs (or dynamic graphs). This is a burgeoning area of graph analytics for which JetStream explores architecture support and optimizations.
- New Asynchronous Streaming Algorithms: JetStream supports the union of GraphBolt and KickStarter (software streaming graph frameworks that also support edge deletion).
- Large Performance Improvements that improve with smaller batch sizes: JetStream substantially outperforms both software frameworks. In addition, its advantage grows as the batch size is reduced, making it conceivable to work on small batch sizes and allow near real-time updates.
- Requires only small modifications to GraphPulse: JetStream extends the event-driven execution approach of GraphPulse. Since graph mutations can be encapsulated as events, we were able to design JetStream with only a few extensions to the existing architecture, and support edge deletion and coalescing as well as two optimizations for significant performance boost.

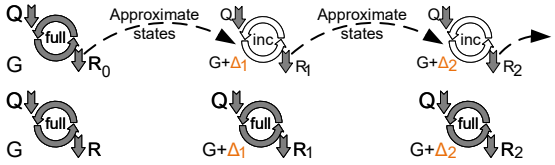

Figure 1: Query evaluation on a streaming graph using an incremental algorithm (top) and static algorithm (bottom).

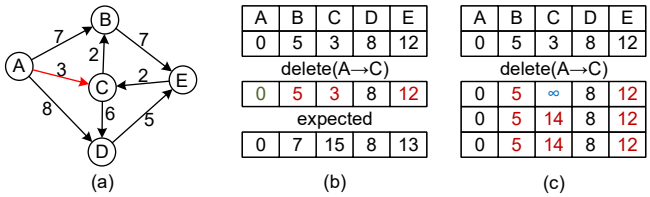

Figure 2: Using intermediate and initial values leads to incorrect results for SSSP: (a) an example graph; (b) uses previous state to recompute; (c) resets impacted vertex.

2 BACKGROUND AND MOTIVATION

2.1 Streaming Graph Analytics

A query evaluation over a streaming graph, as shown in Fig. 1, has two distinct characteristics. First, it supports streaming updates: new graph updates also arrive as the query is being evaluated. These updates are collected in a batch (e.g., Δ_1 or Δ_2 in Fig. 1) and are applied only after the query evaluation is complete and its results reported. *Graph updates consist of edge additions and deletions.* A vertex addition can be modeled by addition of the first edge to/from the vertex while modification of an edge weight is modeled by its deletion followed by an addition of an edge with the same weight. Second, query reevaluation leverages the existing state computed before the updates: after a batch of updates has been applied, the query evaluation is resumed incrementally to obtain the query results for the updated graph. In an algorithm (or accelerator) that supports streaming operation, the reevaluation is performed as an incremental update of the previous query result computed on the original graph, shown as *approximate states* in Fig. 1, to avoid wasteful redundant computations. As updates continue to arrive, the incremental computation is performed repeatedly. JetStream improves upon most prior software streaming algorithms, which only support streaming edge additions, by allowing edge deletions. It also improves on most software frameworks by supporting concurrent processing of multiple updates, gaining efficiencies from combining some of their overheads.

2.2 Incremental Query Evaluation

Incremental reevaluation uses the result of the prior query to find an intermediate approximation, which becomes the initial state for computing the query result on the updated graph. Using the previous result for an approximation can lead to faster convergence than using a random initial state for the updated graph. Intuitively, for many query types, only a small fraction of vertices are affected by graph changes since batch sizes are typically tiny compared

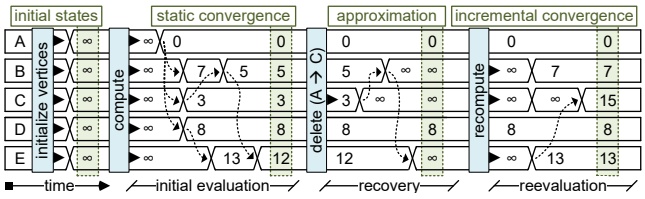

Figure 3: Conceptual timeline showing vertex values over time through initial evaluation, recovery, and reevaluation phases for SSSP on the example graph in Figure. 2.

Algorithm 1 Event-Driven Execution Model for SSSP

```
    V[:] ← fill(∞)                                   ▷InitializeVertex()
    Q ← insert({⟨root, 0⟩})                          ▷ InitialEvents()
 1: procedure Compute(G(V, E), Q)
 2:     while Q is not empty do
 3:         ⟨i, δ_i⟩ ← pop(Q)
 4:         temp ← V[i]
 5:         V[i] ← min(V[i], δ_i)                    ▷ Reduce(a, b)
 6:         if V[i] ≠ temp then                      ▷ Needs to propagate
 7:             for each ⟨u → v, w⟩ ∈ E | u = i do
 8:                 δ_v ← V[u] + w                   ▷ Propagate(u, v, w)
 9:                 Q ← insert(⟨v, δ_v⟩)
10:             end for
11:         end if
12:     end while
13: end procedure                                    ▷ Converged graph state in V
```

to the size of the graph (thousands of edges in graphs with billions of edges). Thus, a complete restart of the graph computation ends up doing substantial redundant work. Of course, we need to have an effective algorithm for identifying which vertices require recomputation for doing incremental updates.

Motivation and Basic Operation: Monotonic algorithms often produce incorrect results in the presence of deleted edges. We consider the example of an edge deletion(A→C) in the graph of Fig. 2(a) for Shortest Path algorithm. Since the vertices only update when they receive a shorter path value than their current state, the graph never reaches the expected result using the previous result as shown in Fig. 2(b). We call this approximation *unrecoverable* because the computation cannot recover to the correct result after being set into an incorrect state by the edge deletion. If we reset the target of deletion to its initial value as shown in Fig. 2(c), it still never reaches the correct result because other vertices (B, D, E) previously influenced by it are also in incorrect states.

Fig. 3 shows the progress of a query evaluation through different phases. First, a graph is *initialized* to an *initial state*. As computation progresses, the graph moves through several *intermediate states* to reach a *final state* when the algorithm terminates. Here, the final state is the correct *converged state* (static), and all intermediate states (including the initial state) are *recoverable states* because the graph can reach the correct state from there. A *recoverable approximation* is equivalent to one of these *recoverable states* from which the graph is guaranteed to converge correctly. After applying the graph mutations, the challenge in incremental graph computation is to find a recoverable approximation based on the previous converged states. For this example, all the vertices possibly influenced through the deleted edge in the initial evaluation is identified and reset in the *recovery phase* to arrive at a recoverable approximation for the reevaluation. Incremental recomputation on this approximation in the *reevalauation phase* leads to the correct result.

Recovery Algorithms: A simple way to find the set of vertices affected by a deleted edge is to iteratively propagate a *tag* downstream from the target vertex of the deleted edge as in GraphIn [37]. Note that if a vertex is not affected by an update, the propagation is not forwarded again. The set of vertices tagged this way definitively contains all possibly impacted vertices. The tagged vertices can then be reset to the initial value to acquire a recoverable approximation for a monotonic convergence. When the query is reevaluated, the reset vertices converge to correct states based on the mutated graph. An example for obtaining a recoverable approximation using tag propagation in the recovery phase is shown in Fig. 3.

JetStream develops event-driven adaptations of vertex tagging and dependence tracking so that they can be used to extend the GraphPulse architecture to support incremental computation over a streaming graph. Monotonically converging algorithms where vertex value computation is a *selection* task – such as ShortestPath, ConnectedComponents, WidestPath, and BFS – benefit from this approach. Graphs with accumulative update functions – such as PageRank and Adsorption – uses a simpler recovery technique in the event-driven approach. Here, the impact of a deleted edge is negated by sending the total contribution through that edge with negative polarity. This makes the event-driven approach highly suited for the incremental computation of these algorithms.

3 JETSTREAM DESIGN OVERVIEW

We present the design of our event-based streaming accelerator and its underlying algorithms in this section. First, we describe the event-driven execution model that GraphPulse [33] is based on. Then, we formalize the problems of building a streaming accelerator over a static one and describe the JetStream model that solves these problems.

3.1 Event-based Processing in GraphPulse

JetStream extends GraphPulse to support streaming graphs [33]. GraphPulse employs event-driven execution to eliminate overheads of shared-memory frameworks (e.g., poor temporal and spatial locality, atomic memory accesses, and synchronization). The event-driven execution is based on delta-accumulative incremental computation (DAIC) [50] model. In this model, contributions coming over different edges (termed *delta*) can be applied independently and without any fixed order to compute the vertex state. The model has two primary components – *i)* a Reduce task used to compute vertex state from incoming *deltas* and previous vertex state; and *ii)* a Propagate task used to compute the *delta* over each outgoing edge. In the event-driven model, lightweight messages called *events* carry the *deltas* to their respective destination vertices. A vertex recomputes its state only if it receives an event (*delta*) and generates a new event only when its state changes from the incoming event.

GraphPulse presents a complete execution model to run an iterative graph algorithm using the event-based approach. Algorithm

1 shows the event-driven execution model and how the SSSP application is mapped to the model. The user defines a Reduce() method (line 5) expressing the *reduction* of incoming contribution and vertex state. A Propagate() function (line 8) is defined for finding the delta over an outgoing edge and creating new events. InitialVertex() and InitialEvents() methods are defined to initialize the vertex states, V, and the initial set of events (Q) before the execution starts. The initial vertex values are set to an Identity value for the Reduce() function, so that a vertex's first *reduction* operation with an events is bound to change its state and propagate. With the processing of the initial events, vertex states get updated towards convergence, and new events are generated and inserted to Q. For each event in Q, the vertex update task is triggered. When a vertex reaches convergence, its state does not change from incoming events, preventing new event propagation (line 6). Eventually, Q becomes empty when all vertices reach convergence terminating the application.

Proper execution and termination of the event-driven model depend on two properties of the graph algorithms. First, the Reordering Property requires that incoming contributions over edges can be applied to a vertex in any order and independently. Second, the Simplification Property requires that vertex that does not change state should not impact other vertices, i.e., it should not propagate, and other vertices should not require its contribution for computing their states. Many important graph algorithms such as SSSP, SSWP, BFS, Connected Components, Incremental PageRank, and many Linear Equation Solvers satisfy these properties and are supported in GraphPulse. JetStream supports all the algorithms supported in GraphPulse without any change to the application.

Limitations. We assume that Reordering and Simplification preserve correctness; however, some graph algorithms do not satisfy this condition and thus cannot be expressed using our model. For example, Graph Coloring, K-Core, and MIS algorithms require vertex contribution across all incoming edges to update a vertex. This violates the Simplification Property since contributions from some neighboring vertices are needed even if their states were unchanged. If the algorithm requires contributions from neighbors that are multi-hop away (e.g., Triangle Counting) or a normalization step after each iteration (e.g., Label Propagation), then they violate the Reordering Property because a particular order must be imposed upon the evaluation of the contributions through some edges. These algorithms cannot be implemented in GraphPulse and, hence, JetStream. It should be noted that some algorithms that are not supported in their common iterative forms may have variations that may be suitable for event-driven implementations. For example, PageRank and Adsorption have incremental forms that are supported in GraphPulse and JetStream. As a rule of thumb, algorithms supported by this model *often* have the characteristic that a single edge can update a vertex, and the updates are monotonic.

3.2 Streaming Graph Computation Objective

GraphPulse computes the final converged state of a static graph. We want to find the new converged state of the graph using JetStream after some mutation is applied to the graph structure. To formally describe the objective of JetStream, we consider a graph $G^0 = (V, E^0)$ being initialized to a set of values $I_G = \langle \forall j : i_j = \text{Identity} \rangle$ and converging to $C^0_G = \langle c_0, c_1, ..., c_{n-1} \rangle$ for its final state. The Identity

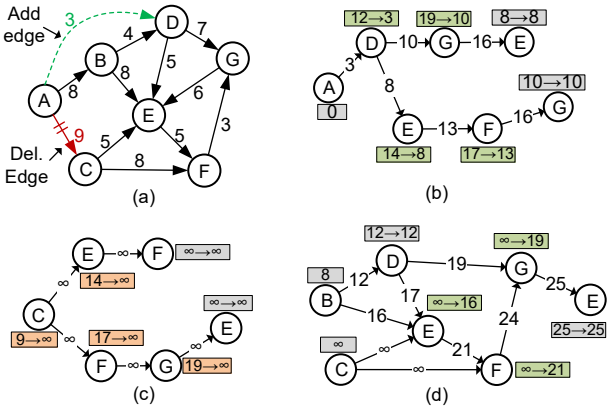

Figure 4: Propagation of events during processing of streaming edges in SSSP. (a) An example graph. (b) Propagation and updates from the insertion of edge $A \rightarrow D$ in the graph. (c) Propagation of deletes and resetting impacted vertices due to the deletion of edge $A \rightarrow C$ in the graph. (d) Recovery of approximate state after request events are processed.

parameter is application-specific for the graph algorithm; it is the *initial value* of the vertices and the non-dominant value for the Reduce() operation (Algorithm 1). For streaming algorithms, we need to compute a new convergence state C^1_G for the mutated graph, $G^1(V, E^1)$, using a recoverable approximation A_G based on C^0_G. The approximation $A_G = \langle a_0, a_1, ..., a_{n-1} \rangle$ is *recoverable* if convergence can be reached for algorithm S starting from this approximation (Section 2.2). For the *selection-type* algorithms, The vertex states progress from the initial value (Identity) to the direction of convergence monotonically. A *more progressed* value dominates the Reduce operation. In a valid approximation, all elements in A_G must be *less progressed than* or *equal to* the corresponding elements in the eventual converged state, C^1_G. An approximation, $A = \langle \forall i, a_i = \text{Identity} \rangle$, set to the initial value is a valid recoverable approximation but an inefficient one since it is equivalent to computing the graph from the beginning. Hence, finding a good approximation is critical for performance. Our proposed approaches in JetStream accomplish this by expressing the graph mutation as events and restoring the mutated graph to a recoverable approximation for subsequent processing using the event-driven model.

3.3 Event Representation of Graph Mutation

Any modification to the graph structure is expressed using an event in JetStream. We assume that the modifications are batched, consistent with prior works on streaming graphs. A batch will be queued as events that are released once the ongoing processing iteration is complete. This choice to separate the update phase from the processing phase eliminates the need for resolving race conditions between old and new values as the computation proceeds. Each modified edge is expressed as an event from the source to the destination of the edge. The payload (*delta*) carried by the event is generated by reading the previous converged state of the source vertex (which is *approximate* with respect to the mutated graph) and computing the propagation value based on this state and the edge attribute.

Algorithm 2 Converting edge-insertions to events

```
1: procedure ProcessInserts(G(V, E), Q, A⟨u → v, w⟩)
2:     for each ⟨u → v, w⟩ ∈ A do          ▷ A = list of added edges
3:         δ_v ← V[u] + w                  ▷ Propagate(a, b)
4:         Q ← insert(⟨v, δ_v⟩)
5:     end for
6: end procedure
```

Algorithm 3 Converting deletions to events for PageRank

```
1: procedure ProcessDeleteCumulative(G(V, E), Q, D⟨u → v, w⟩)
2:     for each ⟨u → v, w⟩ ∈ D do          ▷ D = list of deleted edges
3:         δ_v ← −1 × V[u] × (1 − α)/deg(u)   ▷ Propagate(a, b)
4:         Q ← insert(⟨v, δ_v⟩)
5:     end for
6: end procedure
```

This event represents the effect of the modified edge with respect to the previous graph structure. Events are queued and held until all the modified edges have generated a corresponding event. At this point, the new graph structure is active, and the events are processed from the queue. We demonstrate the processing of edge insertion and deletion events next.

Edge Insertions: are supported naturally by the event-driven model. The inserted edge did not exist in the previous graph and had no effect that needs to be reverted. An update along an edge can be applied to a vertex at any time in the asynchronous model. Hence, an update coming along a newly-inserted edge is conceptually similar to an update along an existing edge that was delayed; it has the same effect and gets processed in the same way. JetStream computes an update using the old converged state of the source vertex and the weight of the inserted edge, and queues it as an event for the destination vertex along with regular events (Algorithm 2). Fig. 4(b) shows how an edge insertion triggers a chain of updates. As the new edge ($A \rightarrow D$) contributes to vertex D, the vertex gets updated and propagates further with more events ($D \rightarrow G$). Propagation ultimately stops due to monotonicity when the event arrives at a more progressed receiver via ($G \rightarrow E$). If the state of the source vertex A itself is not stable, subsequent updates to the vertex will be propagated using the mutated graph along the new edges and send the correct values downstream eventually. Hence, a graph always remains in a correct or recoverable state after edge insertions.

Edge Deletions: are not supported by most streaming systems (the exceptions being Kickstarter and GraphBolt). JetStream supports deletions as in KickStarter while overcoming some of its performance limitations when handling a batch of deletions. Specifically, JetStream queues edge deletions as events in the same way as insertions. However, edge deletion is more complicated since the deleted edge's contribution to the previous converged state must be reversed. For algorithms with *accumulative* updates, reverting the effect of deleted edges is simpler. A vertex propagates an update downstream for all the updates it receives and accumulates. As a result, we can infer the combined value of all updates it sent along an edge during the previous evaluation by looking at its accumulated state and using the Propagate function. Sending the inverse of its previous converged state, transformed by the Propagate function, negates the cumulative effect of all updates over this edge. Further propagation downstream of negative events from the receiver vertices leads to the rollback of all contributions from this edge and puts the graph in a recoverable state. We create negative events for the deleted edges as shown in Algorithm 3 to initiate recovery.

Algorithm 4 Processing deletes and recovering approximations of vertices impacted by deletions for SSSP.

```
1: procedure ProcessDeletesSelective(G(V, E), Q, D⟨u → v, w⟩)
2:     for each ⟨u → v, w⟩ ∈ L do
3:         Q ← insert(⟨v, 0⟩)
4:     end for
5: end procedure

6: procedure ResetImpacted(G(V, E), Q)
7:     X ← ∅                               ▷ List of impacted vertices
8:     while Q is not empty do
9:         (i, δ_i) ← pop(Q)
10:        if V[i] ≠ Identity then
11:            V[i] ← Identity             ▷ Tag vertex
12:            X ← X ∪ {i}
13:            for each (u → v, w) ∈ E | u = i do
14:                Q ← insert(⟨v, 0⟩)      ▷ Propagate delete
15:            end for
16:        end if
17:    end while
18: end procedure

19: procedure Reapproximate(G(V, E), Q, X)
20:    for each i ∈ X do                   ▷ Create events with request flag(ρ)
21:        for each (u → v, w) ∈ E | v = i do
22:            Q ← insert(⟨u, Identity, ρ⟩)
23:        end for
24:    end for
25: end procedure
```

For algorithms having *selective* updates, it is more difficult to identify which edges contributed to a vertex. The destination vertex of a deleted edge is reset to its initial value so that it can be updated later in the reevaluation phase. We queue events with a *delete flag* as shown in Algorithm 4. A vertex, upon receiving an event with a delete flag, will reset itself. This change in the state goes against the direction of monotonicity. Therefore, when this vertex propagates its updates to its neighbors, the update events will be discarded by the receivers in the Reduce function since they already have a more progressed state. However, this more progressed state may have resulted from the contribution of the deleted edge. Hence, the graph stays in an incorrect state if these vertices are not corrected. To solve this problem, we devise an event-driven edge deletion algorithm that identifies the potentially affected vertices and efficiently resets them to acquire a recoverable approximation as we describe next.

3.4 Impacted Vertex Detection and Recovery

To handle an edge deletion correctly, the vertices impacted by a deletion must be identified, and their states reset to a recoverable value. Impacted vertices are identified by propagating a delete tag to all outgoing neighbors of an impacted vertex and tagging them as impacted in a manner similar to KickStarter [45]. When a deletion event first arrives at a vertex, we set the vertex state to the initial

Algorithm 5 Overall processing flow for SSSP.

```
1: procedure ProcessStream(G(V, E), Q, A⟨u → v, w⟩, D⟨u → v, w⟩)
2:     ProcessDeleteSelective(G(V, E), Q, D⟨u → v, w⟩)
3:     X ← ResetImpacted(G(V, E), Q)                ▷ Queue is empty
   ▷ Delete phase ends
4:     Reapproximate(G(V, E), Q, X)
5:     ProcessInsertions(G(V, E), Q, A⟨u → v, w⟩)
   ▷ Switch to new graph structure
6:     Compute(G(V, E), Q)
7: end procedure                                    ▷ V holds correct result
```

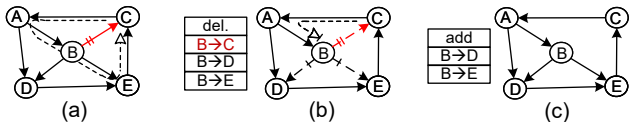

Figure 5: Showing an edge deletion for accumulative algorithms: (a) initial graph with B→C to be deleted; (b) intermediate representation; (c) mutated graph.

Identity value (*tag* it) as shown for vertex C in Figure 4(c). Hence, textittagged vertices can react to updates from future events. Delete events are propagated along each outgoing edge. A delete event cycling back to an already tagged vertex (e.g., $G \rightarrow E$) will not propagate. Multiple delete events queued for the same vertex can be *coalesced* since tagging a vertex once is sufficient. When a vertex is reset, the vertex Id is added to a list. Hence, the set of vertices tagged this way contains all vertices whose states could have been potentially influenced by the deleted edge. The process is shown in Algorithm 4. The list is used to revisit these vertices to recompute their approximate states as described next.

A new recoverable approximation for the impacted vertices must be found in case the query cannot progress to some impacted vertices. For example, in Fig. 4(a), a SSSP query running from A cannot reach E because vertices B and D are already in a correct state, and will not propagate new events along $B \rightarrow E$ and $D \rightarrow E$ after edge deletion. KickStarter solves this problem by reading all neighbors states again to reestablish an approximate state for an impacted vertex. Unfortunately, this approach generates many memory reads with a random access pattern. Many of the vertices are also read by multiple deleted vertices creating opportunities for data reuse. Instead of reading the states of the neighboring vertices directly, we create a *request* event to request updates from the neighbors. The *request* event has a *request-flag bit* set and the payload set to Identity in order to avoid affecting any other events and vertices. When a vertex detects the *request-flag*, it must propagate to its neighbors, even if it does not update itself. The request events are coalesced, hence, combining the reads for each vertex. Also, when they pass through the queue, the events are sorted by their destination vertex ID so that a sequential memory access pattern occurs when they are processed. Upon receiving the response to the *request* event, the impacted vertex will reestablish an approximate state closer to convergence based on its neighbors' approximate states.

A second inefficiency persists in other approaches because computing an approximate state from neighbors' approximate states is often wasteful since these approximate states may change again during query evaluation. To address this problem, we exploit the asynchronous nature of the model – we can delay the vertex reads or recomputation until after the effect of the initial events and inserted edges are applied. We overlap the execution of request events with query events and edge insertions, so the vertex updates move the vertex closer to the final converged states.

After the delete phase is over, JetStream revisits each vertex in the list of impacted vertices and sends *request events* along each incoming edge of a vertex at the beginning of the processing phase. If the impacted vertices are on the path of a propagating query, their states update to the correct states since their approximate state (Identity) can be updated by all contributions. If the vertex does not belong to the query propagation path, the responses to request events set them to the correct state. Thus, a graph always remains in a correct state after deletion is processed in this technique. The pseudocode for processing deletes is shown in Algorithm 4.

3.5 Recomputaion of the Mutated Graph

JetStream execution process uses the original computation technique of GraphPulse to recompute the graph after setting up the approximate state and populating the event queue with appropriate events as described above. Because the recovery after delete is handled differently in the two different types of algorithms (accumulative vs. monotonic), the processing phases are scheduled differently for them. We discuss both of them next.

Algorithms with Selective Update. After receiving a batch of edge updates, we first process the deleted edges and insert deletion events in the queue to reset the target vertices. In the next phase, the events are allowed to execute on the previous version of the graph; all potentially impacted vertices are reset to their initial value. Afterward, events with *request-flags* are queued for all the neighbors of the impacted vertices. We process the inserted edges at this point to create and queue the events for them. The insertion events can *coalesce* with the *request* events existing in the event-queue simply by setting their *request-flag* bit. The graph is then switched to the new version, and the events in the queue are allowed to process in the typical computation flow of GraphPulse. *The only difference is that whenever any vertex receives an event with a request flag, it propagates its state to all its outgoing neighbors even if it does not change its state.* These responses to the reapproximation request allow the impacted vertices to set their new state using the states of their neighbors. At the end of this phase, when the queue is empty, the graph arrives at a correct state, and the process of reevaluation concludes. The process is shown in Algorithm 5.

Algorithms with Accumulative Update. These algorithms do not need reset since a deleted edge can be negated with a regular event with negative polarity. After creating events for the deleted edges, we load an intermediate version of the graph without the deleted edges to break any cyclic path in the graph. Algorithms that propagate updates based on degree, such as PageRank, undergo changes in the weight of all edges when an edge is added or deleted. To handle this, we first delete all outgoing edges of the vertex having an edge added or deleted, turning it into a complete sink for the intermediate version of the graph. In the example of Fig. 5(a), any cyclic propagation through vertex B is stopped by deleting

Algorithm 6 Overall processing flow for PageRank.

1: **procedure** ProcessStream($G(V, E)$, Q, $A\langle u \rightarrow v, w\rangle$, $D\langle u \rightarrow v, w\rangle$)
2: ProcessDeleteCumulative($G(V, E)$, Q, $D\langle u \rightarrow v, w\rangle$)
▷ Switch to intermediate graph structure
3: Compute($G(V, E)$, Q) ▷ Q empty : Delete phase ends
4: ProcessInsertions($G(V, E)$, Q, $A\langle u \rightarrow v, w\rangle$)
▷ Switch to new graph structure
5: Compute($G(V, E)$, Q)
6: **end procedure** ▷ V holds correct result

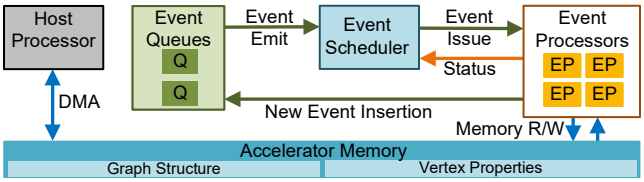

Figure 6: Fundamental architecture of GraphPulse [33]

edges to D and E too. All outgoing edges of vertex B are added to the batch of deleted edges (Fig. 5(b)). We next process these deleted edges to populate the event queue with negative events. Next, a computation phase on this intermediate graph effectively removes all contributions of vertex B from the graph. Creating the intermediate graph is not expensive since it can be achieved simply by adjusting the pointers to the edge list to skip the deleted vertices. We then add back all the edges of vertex B (except the actually deleted edge B→C) to the batch of inserted edges so that it resembles the new graph structure (Fig. 5(c)). This batch of edge additions is processed to create events in the queue. When the compute phase is rerun on the new version of the graph, the result is correct for the mutated graph. The steps in this model are shown in Algorithm 6. We note that the manipulation of the edge addition or deletion batch only affects the preparation of the streaming batch; the actual vertex computation remains the same as GraphPulse.

4 JETSTREAM ARCHITECTURE

JetStream is an asynchronous graph processing accelerator leveraging the event-driven execution model to operate on streaming graphs. The decoupled nature of event-driven execution allows the accelerator to extract abundant parallelism for the computation flow and utilize memory bandwidth efficiently. A significant performance boost comes from the efficient utilization of low-latency on-chip memory resources for the transient short-lived communication data. In addition, specialized communication paths and scheduling primitives allow the accelerator to operate with very little overhead for control and synchronization. JetStream extends the datapath of GraphPulse, an accelerator for static graphs, to accommodate the model described in Section 3. JetStream adds new modules for reading and processing streaming data, as well as re-implements the coalescing queue, and vertex update and propagation logic to account for the new types of events.

This section describes the architectural components of the GraphPulse core and highlights the extensions for JetStream. JetStream's architectural changes do not disrupt the regular computation on static graphs. As a result, JetStream can perform both the initial non-incremental evaluation (like GraphPulse) and streaming evaluation efficiently. We describe the complete execution flow of JetStream later in this section. Furthermore, JetStream derives its functional module from the same programming API defined for GraphPulse; so minimal additional user effort is necessary to program JetStream.

4.1 GraphPulse Architecture

Since JetStream builds on GraphPulse, we start by over-viewing the GraphPulse base architecture shown in Fig. 6. The primary components of the datapath are Event Queues, Event Scheduler, Processors, and the on-chip routing network connecting the components. The processors are connected to the off-chip system memory for accessing the graph structure and vertex states. Any computation starts with the vertices set to an *identity* value and a number of initial events crafted for setting the vertices to their initial state. The events are dequeued and processed from the event queues for processing. The event queue consists of several individual queues, each holding events for a subset of the vertices, to increase queueing and dequeuing bandwidth. Each event updates a vertex that may trigger new events, one for each outgoing edge, that are inserted in the event queues. The event queues hold one entry for each vertex; multiple events destined to the same vertex are coalesced by the queuing logic in the event queue, which is defined as part of the application. For example, the coalescing logic will retain the incoming event with the least cost for *Shortest Path*. Processing continues until no more events are available or another user-specified termination condition is reached. The size of the event queues limits the size of the graph being processed since they hold one entry per vertex; GraphPulse supports larger graphs by partitioning them into multiple slices and swapping in one slice at a time for processing. GraphPulse incorporates a number of additional optimizations; for more details, please refer to the GraphPulse paper [33].

The accelerator is designed to work alongside a host as an ASIC/FPGA-based co-processor with dedicated DRAM memory and independently addressable memory space. The host processor allocates and initializes the graph and the initial events in the accelerator memory as defined by the programmer via a provided API. The accelerator performs the graph computation independently based on configurations received from the host. It alerts the host when computation finishes so that the graph state can be read back.

In the remainder of this section, we describe the primary GraphPulse components and how JetStream extends them. JetStream retains the GraphPulse datapath and adds a *Stream Reader* module for creating events from streaming data as described in section 3.3. It extends the vertex update module with a vertex reset logic, a scheduler with multiple policies, and coalescer logic incorporating *delete event* coalescing described in Section 3.4. A detailed view of the JetStream datapath is shown in Fig. 7, where the shaded components indicate modules added to or extended from GraphPulse.

4.2 Event Management

All computations are expressed as contributions along edges and propagated using events in the event-driven model. Events are lightweight messages that trigger vertex computation at the destination vertex. GraphPulse events are tuples containing a target vertex Id

and a payload. The payload contains the vertex contribution along the edge. In JetStream, event payloads also contain some flags indicating special tasks (e.g., request flag mentioned in Algorithm 4). We describe optimizations in Section 5 that add extra data to the event payload in JetStream.

The event queue is the storage for active events in the system representing the set of active vertices. GraphPulse employs a fast on-chip queue capable of in-place coalescing. The queue contains multiple bins. Each bin is structured into a grid of rows and columns, and only one vertex is mapped into each cell by vertex index. The bins behave similar to a direct-mapped cache. During event insertion, if another event already exists in its mapped cell in the queue, the events are combined with the Reduce operation (coalescing). Thus, only one event for a vertex can exist in the queue at any time.

The queue is capable of fast parallel insertion of events received on the input bus. The bins are implemented on *Simple Dual-Ported* on-chip memory where one row can be read and written in each cycle. Furthermore, each bin is equipped with a coalescer pipeline that can insert one event every cycle even though coalescing may have multi-cycle latency. During insertion, the coalescer reads the existing event (if any) in the mapped block on the first cycle. Then, the existing event is *reduced* with the new events in the following cycles and written back.

Events are emitted in batches for processing. Since GraphPulse supported algorithms allow reordering of edge contributions, events can be emitted in any order. GraphPulse reads one full row of events at a time from a bin and puts it into a *drain buffer*. Events are drained from one bin at a time in a round-robin fashion. The vertices are mapped in such a way that a group of vertices whose states reside in the same DRAM page is also mapped in the same row in the queue. Thus, processing the events in one row of the queue within a short period provides a high spatial locality for the graph memory.

JetStream leverages the same queue architecture as GraphPulse. The coalescer pipelines are extended to combine delete events as well during the *recovery phase*. Two delete events can be merged since they do not carry any data. Additionally, fewer vertices can be mapped to the queue (for the same on-chip memory size) since the event payload in JetStream is bigger than GraphPulse. Hence, JetStream uses smaller-sized graph partitions than GraphPulse.

4.3 Event Scheduler

The GraphPulse event scheduler dequeues events from the queue and puts them in a buffer. It keeps track of processor occupancy, and arbitrates events to the processors with the least workload. It issues the events in the same *queue row* to the same processor for enhancing spatial locality. The scheduler also tracks the progress of the processing engines and the occupancy of the queue. When all the bins have been drained once, we say that a *round* is completed. The scheduler waits for the processors to idle before starting a new round. Since only one event for a vertex can exist at the time of emitting event, there cannot be more than one event scheduled for the same vertex in one round; this eliminates the need for atomic operations and simplifies memory access and synchronization. When the scheduler detects that the queue is empty and all processors have completed their assigned workload, it indicates the end of the computation phase and terminates the application.

In JetStream, the scheduler is extended to run the execution in multiple phases. When a streaming batch is ready, the scheduler starts processing for the recovery phase that precedes the regular computation phase. The recovery phase starts with populating the queue with *delete events* from graph mutation. Then it proceeds like a regular computation phase and ends when there is no *delete event* remaining in the queue. At the end of this phase, the graph is in a recoverable approximation state. Finally, the scheduler triggers the creation of *addition events* from added edges and runs the a regular computation phase (reevaluation) to obtain the final graph state.

4.4 Event Processing Engine

GraphPulse event processors are independent, parallel, and simple state machines. They continuously process events that are placed in their input FIFO buffers by the scheduler. The processors compute the vertex states using the user-defined Reduce() method and apply the updates to the vertex memory. Since the processors receive events that are closely located in the memory in one batch, they can prefetch the vertex properties for these events. Each processor is equipped with an on-chip scratchpad prefetcher that can prefetch vertex data for all the events in the processing buffer. The prefetcher scans the buffer and reads the off-chip memory in such a way that vertex properties residing in the same DRAM memory page are read in a group, thus increasing memory access efficiency. The processors read and write vertex data through the scratchpad memory. The scratchpads can access any memory channel through an efficient memory bus.

When vertex states change, the processors pass the updates to one of their event generation streams. The generation streams read the edges and compute the contributions using the Propagate() method to pass along the edges. Event generation streams also read the edge data through an edge cache connected to off-chip memory bus. Since edge lists are contiguous in memory, a prefetcher requests next memory blocks smartly based on the edge pointers and the number of edges in the *Edge ID Buffer*. The generation streams are connected to the queue using a crossbar. 32 generators of 8 processing engines share the input ports of the 16×16 crossbar, and the output ports are shared among the queue bins.

JetStream utilizes the same event processor system during its regular computation phase. The apply logic is extended with a *reset logic* that sets a vertex to Identity (Algorithm 4, line 11) when it receives a valid *delete event* during the approximation phase. It also, writes the vertex index to the *Impact Buffer* if a vertex resets its state from a *delete event*. Additionally, the processing buffer is increased in width to accommodate larger event size for JetStream.

4.5 Stream Processing Modules

JetStream adds a *Stream Reader* module that reads the lists of deleted and inserted edges from main memory and schedules them to the processing engines during approximation. The list of deletes is read first as ⟨*source, destination, weight*⟩ and events are created from these edges according to Section 3.3. Next, these events are used to find the sets of impacted vertices. Finally, added edges are read, and events are created after the approximation is complete.

The *Impact Buffer* stores the indices of the vertices impacted by a delete during the approximation phase. The *Apply unit* sends the

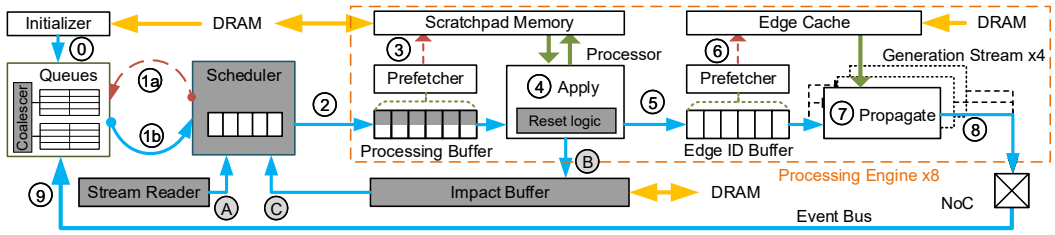

Figure 7: Detailed JetStream datapath. Blue shows data flow, red shows control signals, green and yellow represent on-chip and off-chip memory transfers respectively. Shaded modules are new or modified in JetStream.

index of an impacted vertex to the *Impact Buffer* module that writes to a list in its internal buffer. The list is written from the buffer to the main memory in batches. The *Impact Buffer* module also reads back the list and creates *request events* for the impacted vertices as described in Section 3.5.

4.6 JetStream Execution Flow

Fig. 7 shows the steps and direction of the dataflow during the life-cycle of an event. The dataflow differs for the initial (static) and incremental evaluation.

4.6.1 Initial Evaluation. The regular computation phase is inherited from GraphPulse and it is used for the initial static evaluation.

Initialization. We assume that the accelerator starts with the host processor writing the graph structure, initial vertex states, and a list of initial events corresponding to the application to the main memory. Then, during step ⓪, the *Initializer* module reads and inserts the initial events into the queue to make the system ready for processing.

Event Issue. In step ①a, the scheduler requests events from the queue, and the queue emits events (if any) in batches in ①b. The steps in ① execute in a continuous loop. The scheduler holds the events in a buffer and passes them to the processing buffer in ② where they are staged for execution.

Vertex Update. While the events wait in the queue, the prefetcher scans the vertex id, computes the memory addresses, and prefetches all vertex properties (typically located in the same memory page) to the scratchpad memory in ③. The *Apply module* takes the event at the head of the buffer, reads vertex states and edge pointers from the scratchpad, and applies the update to the event in ④. After writing back the updated value to memory via the scratchpad, ⟨*update value, edge pointer, number of edges*⟩ for a vertex is pushed to the *Edge Buffer* in ⑤ to generate the outgoing events only if the vertex requires propagation (i.e., its state has been updated).

Event Generation. During step ⑥, the prefetcher computes the edge address range to be read, and fetches all needed edges (typically within a single memory page) to the cache. Each generation stream takes the head of the buffer and loops over all the edges for the vertex to generate new events in ⑦. The events are pushed to an event bus through an on-chip routing network in ⑧. In step ⑨ the event queue continuously scans the event bus to pick up and insert the events in corresponding bins. This processing cycle repeats until the queue is empty; this marks the end of evaluation where the initial graph has been updated to the converged state.

4.6.2 Incremental Evaluation. The incremental evaluation is added in JetStream and required for fast evaluation of streaming graphs.

Delete Setup and Preparation. Edge additions are directly supported as regular events since they do not affect the monotonicity of the algorithm; we focus on the more difficult deletion support. The *Stream Reader* reads the deleted edges first in Ⓐ and passes them to the processing engines through the scheduler (②). Reusing steps ③ - ⑤, the vertex state for the source vertex is read (but not updated) and the ⟨*vertex state, destination, edge weight*⟩ is passed to the generation unit. Step ⑦ is used to find the propagated value, and create a delete event for the destination vertex that is forwarded to the queue using ⑧, ⑨. Note that the computing elements of ④ and ⑦ are not necessary for the basic model. But they are used during the optimizations described in Section 5.

Delete Propagation. After all the delete events are queued, a normal computation cycle (steps ⓪-⑨) is executed until there remains no delete events in the queue. The Apply unit and Propagation unit use the logic defined in Algorithm 4, line 11 and 14. The Apply unit also writes the Id of a deleted vertex in step Ⓑ to the *Impact Buffer* during step ④.

Finalizing Approximation. After the delete propagation step concludes, we reschedule the vertices from the *Impact Buffer* (step Ⓒ) and reuse steps ②-⑨ once to create request events for their incoming edges. In this phase, step ④ reads the incoming edge pointers from the memory (in contrast to the outgoing edge pointer as in other phases). Following this, the *Stream Reader* reads the inserted edges, and creates insertion events using ②-⑨ the same way as deleted events. This completes the approximation phase. At this point, the regular computation phase (⓪-⑨) can execute again to evaluate the modified graph. As further streaming updates are received, the engine keeps finding recoverable approximation and rerun computation phase keep processing streaming data.

4.7 Graph Representation and Partition

GraphPulse stores the graph structure in a *Compressed Sparse Row* (CSR) format and the vertex states in simple contiguous arrays. JetStream assumes the same CSR graph storage format. However, different from GraphPulse, JetStream requires access to the incoming edges for each vertex, which are stored in another CSR structure. Since the host processor maintains the graph structure, we leave the task of maintaining the evolving edge list to a suitable software graph versioning framework. In the simplest case, we assume the host writes a new CSR for the mutated graph version to the accelerator memory and swaps the pointer to the CSR after each

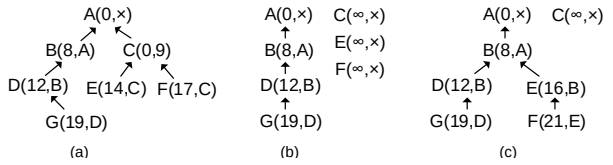

Figure 8: Dependency tree for the example in Figure 4: (a) before deletion; (b) after reset; (c) after reevaluation for the deleted edge A→C.

batch iteration. Thus, JetStream can start using the new version of the graph. In practice, any graph versioning storage, such as Version Traveler [20] or GraphOne [21], can be used. JetStream can interface with any framework that allows a CSR abstraction to access the internal evolving graph structure, and only the *address translation logic* needs to be extended for interfacing.

The hardware queue can accommodate events for a limited number of vertices. So large graphs are partitioned into slices using a minimum edge-cut strategy to avoid overwhelming the queue. GraphPulse processes one slice of the graph at a time in a round-robin manner and temporarily stores the cross-partition events to the off-chip memory. After one *round* over a slice, it is swapped out by writing the pending events to the off-chip memory. Then, a new slice is activated; its events are read back from memory and inserted into the queue. We keep the same partitioning and swapping technique of GraphPulse, as JetStream extensions are not dependent on graph structure. Note that the partitions may not remain optimal as the graph continues to evolve. To reduce the fraction of edge-cuts, we can periodically re-partition the graphs or deploy dynamic graph partitioning tools [15, 43] without affecting the JetStream workflow.

5 OPTIMIZATIONS

We have described a system that uses a tagging approach during edge deletion (Section 3). Next, we describe extensions to the delete propagation algorithms to capture a smaller set of impacted vertices.

5.1 Value Aware Propagation (VAP)

A fundamental property of monotonic algorithms is that the updates propagated from a vertex along its outgoing edges are always *less progressed* (closer to Identity) than the vertex itself. For example, in a Shortest Path (SSSP) algorithm, all the distances transmitted via edges are longer than the vertex's distance from the root. In typical selection-based algorithms, a vertex selects only the incoming edge with the most *progressed* update to set its state. VAP exploits the observation that any source vertex that propagates an update that is *less progressed* than the destination's state, can not be the contributor to its state. Thus, when a vertex is impacted, VAP avoids resetting any neighbor that is more progressed than the resulting contribution from the impacted source.

Implementing VAP requires changes to the event propagation and update logic. The JetStream engine already uses a Propagate logic to compute the value of the events generated along outgoing edges. This same logic is used to compute the propagated value along the deleted edge during the creation of delete events. Upon receiving this event, a receiver vertex compares the event payload to its current state. If the received value is *less progressed* than the receiver, it can be safely discarded. Otherwise, the vertex resets itself to the initial value and propagates the updates along its edges using its previous state. The delete events with value can be coalesced in the queue using the same reduce() function as the one for regular events. Only the most progressed event will remain, and if that does not impact the destination vertex, the delete event is not propagated. This substantially reduces the number of impacted vertices in the system for applications with distinct edge weights and vertex states

5.2 Dependency Aware Propagation (DAP)

Comparing values in applications where clustering vertices settle to the same value is futile. For example, a BFS algorithm sets all nodes to the same value, and VAP cannot exclude any vertex based on value. For such algorithms, we exploit another observation that the vertex states depend on the contribution of only one incoming edge for each vertex. The first contribution that sets a vertex state to the final value is the one that the vertex depends on. Subsequent contributions carrying the same update value cannot affect the vertex. Therefore, deletes propagated along these edges can be safely discarded. The approximate state is recoverable as long as the first contributing vertex remains stable. We adapt the notion of *Dependency Tree* introduced in KickStarter [45] to the event-based model for these kinds of applications.

Formalization. We capture the flow of *useful* contributions across the graph to identify dependency. We use the notion of a `Leads-To` relationship ($\Longmapsto$) that represents the effect of a vertex on the transition of a neighbor's state. Specifically, $A \Longmapsto B$ if the state of B transitions from the contribution of A. In a cyclic path $A \rightarrow B \rightarrow C \rightarrow A$ with a BFS query, if $A \Longmapsto B$ and $B \Longmapsto C$, then $C \not\Longmapsto A$ because A would have already reached the final state and would not transition from the contribution (*futile*) from C. Discarding all delete propagation $u \rightarrow v$ where $u \not\Longmapsto v$ still produces a recoverable approximation. We can represent the `Leads-To` relationship in the form of a *tree*. Note that multiple valid versions of the dependency tree may exist depending on the order in which events are processed.

Implementation. We add a dependency field to the vertex state to record the source of the first event that updates it to a stable value. We also add a field to the event payload that carries the Id of the source of that event. When an event updates a vertex, the vertex changes its dependency field to match the source of this event.

While coalescing two events in the queue during regular computation, we retain the source of the event that is *dominant* in the Reduce function. We disable coalescing during the *recovery phase* not to lose delete events. We extend the queue with an overflow buffer that stores the extra events when multiple events are received for the same vertex. The overflow buffer writes to the off-chip memory in blocks when full and reads back in blocks when issuing events. These off-chip accesses have low overhead as the number of delete events is far smaller than the events in a regular computation.

During event processing, a vertex only resets itself and propagates the delete if the dependency field matches the source ID of the delete event. Other delete events are discarded, greatly pruning the set of impacted vertices. Fig. 8 shows the vertex states and dependency trees during different stages of the incremental evaluation for the example graph of Fig. 4.

Table 1: Experimental configurations.

	Software Framework	JetStream
Compute Unit	36× Intel Core i9 @3GHz	8× JetStream Processor @ 1GHz
On-chip memory	24MB L2 Cache	64MB eDRAM @22nm 1GHz, 0.8ns latency
Off-chip Bandwidth	4× DDR4 19GB/s Channel	4× DDR3 17GB/s Channel

Table 2: Input graphs used in the experiments.

Graph	Nodes	Edges	Description
Wikipedia(Wk) [10]	3.56M	45.03M	Wikipedia Page Links
Facebook(FB) [41]	3.01M	47.33M	Facebook Social Net.
LiveJournal(LJ) [4]	4.84M	68.99M	LiveJournal Social Net.
UK-2002(UK) [6]	18.5M	298M	.uk Domain Web Crawl
Twitter(TW) [22]	41.65M	1.46B	Twitter Follower Graph

Overheads. This approach changes the data structure requiring more memory for vertex states and on-chip events compared to VAP. However, the dataflow architecture and the control sequence remain intact. Only the vertex update logic and event coalescing logic need to be modified. Not coalescing events during recovery raises the concern of transaction safety if multiple events are issued to processors concurrently. This is not an issue. Because in this approach, only one event matching the dependency field can reset a vertex, and thus only one vertex process will write back to memory.

6 EVALUATION

JetStream is implemented on a cycle-accurate microarchitectural simulator based on the Structural Simulation Toolkit (SST) [35]. The off-chip memory is modeled with DRAMSim2 [36]. We use a detailed bus communication, scratchpad, and cache memory model built within SST to evaluate communication and memory access characteristics. The event processing and memory system configuration of the modeled framework is shown in Table 1. For large workloads unable to fit in the on-chip memory, we followed the same partitioning technique as GraphPulse. We used PulP [40] for edge-cut-based slicing of the graphs.

6.1 Experimental Setup

Our comparison is focused on showing both the advantage stemming from algorithmic support and hardware acceleration. First, we show the benefit of the incremental reevaluation by comparing the performance with *"cold-start"* computation of GraphPulse, where the whole graph is processed from initial states after each batch of updates. We used the same hardware configuration for GraphPulse and JetStream. Then, we compare the performance and characteristics with two software frameworks to show the benefit of accelerating a streaming graph analytics engine. We compare with GraphBolt [26] for accumulative algorithms and KickStarter [45] for monotonic algorithms with selective updates. The system configuration for software benchmarks is shown in Table 1.

Table 3: Execution time (in ms) per query on streaming graphs and speedup over other frameworks.

		WK	FB	LJ	UK	TW	GMean
SSWP	Jet	1.63	1.21	4.17	3.87	22.55	
	GP	10.4×	9.3×	16.7×	66.7×	43.2×	21.6×
	KS	12.4×	13.1×	8.4×	24.2×	5.2×	11.1×
SSSP	Jet	4.76	4.31	5.36	6.23	15.17	
	GP	9.4×	9.95×	13.3×	73.4×	35.5×	20.1×
	KS	21.8×	8.7×	6.5×	25.6×	11.2×	12.9×
BFS	Jet	2.74	1.24	1.61	8.12	17.75	
	GP	3.10×	5.35×	7.80×	8.18×	15.1×	6.9×
	KS	30.1×	8.31×	11.7×	11.5×	5.57×	11.3×
CC	Jet	1.64	1.44	2.59	5.07	11.73	
	GP	12.9×	13.2×	12.4×	21.4×	23.4×	16×
	KS	7.62×	8.60×	5.25×	9.38×	8.51×	7.72×
PageRank	Jet	5.17	4.29	6.62	6.99	169	
	GP	12.8×	19.5×	19.9×	56.6×	9.70×	19.4×
	GB	143×	231×	180×	402×	51.6×	165×
Adsorption	Jet	4.19	5.27	9.84	12.10	65.30	
	GP	5.78×	3.90×	5.08×	5.95×	9.41×	5.77×
	GB	12.7×	14.4×	15.9×	12.8×	38.6×	17.1×

Workloads. To demonstrate the performance of realistic workloads, we select five real-world graph datasets (see Table 2). Among these workloads, Wikipedia and UK-2002 domains graphs represent narrow graphs with long paths, and Facebook, Livejournal, and Twitter graphs represent large, highly connected networks. We run 6 graph algorithms on these datasets for our evaluation. ShortestPath (SSSP), WidestPath (SSWP), Breadth-First Search (BFS) and Connected Components (CC) are the representative applications for selection based update functions. Incremental PageRank and Adsorption are evaluated to show the performance of accumulative algorithms. We note that, for our optimization technique with the embedding of dependency information in events (DAP), the event size is bigger than GraphPulse and thus requires a smaller graph slice to fit in the memory. We run 6 slices on Twitter and 3 slices on UK-domain graph for the selective algorithms in JetStream compared to 3 and 2 slices respectively for GraphPulse.

6.2 Performance and Characteristics

Overall Performance. Table 3 shows the execution time of JetStream with different workloads for batches of 100K edge updates. Each batch contains 70% insertions and 30% deletions of edges. The table also shows the speedup over GraphPulse (GP), KickStarter (KS), and GraphBolt (GB) for comparative workloads. GraphPulse demonstrates the cost of complete recomputation of the graph in an accelerator. JetStream takes 3 to 74 times less than GraphPulse (13× on average) to reevaluate a graph. This advantage primarily comes from heavily reduced vertex computation and edge communication required in JetStream. Fig. 9 shows that JetStream limits the number of vertex accesses to less than 54% and as low as 3% of what GraphPulse would require with less than 30% events generated.

Similar speedup in comparison to KickStarter and GraphBolt shows that the event-driven model is effective across incremental techniques. We observe up to 30× speedup over KickStarter and

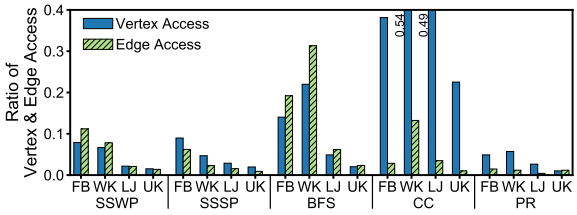

Figure 9: Number of vertex and edge accesses in JetStream normalized to GraphPulse.

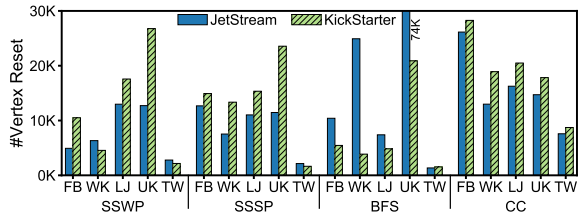

Figure 10: Number of vertices reset by 30K edge deletions.

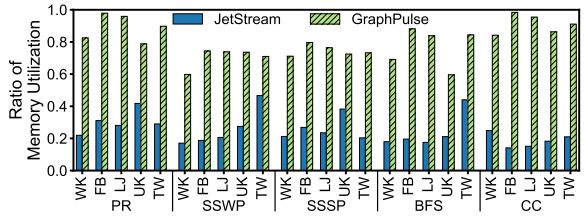

Figure 11: Utilization of off-chip memory transfers.

400× over GraphBolt. JetStream is 18× faster on average than both. JetStream's asynchronous model performs better on the narrow but long graphs (UK, WK) than the synchronous software frameworks.

Approximation Effectiveness. JetStream adopts a technique similar to KickStarter for trimming the set of vertices. KickStarter employs value-aware and dependency graph (with levels) based trimming to limit recomputations. The source-based dependency-aware propagation technique in JetStream often finds smaller set of impacted vertices. Fig. 10 shows the number of vertices reset in JetStream and KickStarter for the same 30K batch of deletions.

Memory access efficiency. The ability to prefetch and utilize memory effectively is one of the major source of speed up in GraphPulse. The caches use 64-bytes lines which may not all be accessed. We show in Fig. 11 the ratio of bytes read into the computation engine from cache/prefetcher to bytes read from memory into caches to demonstrate how efficiently the off-chip data transfers were utilized as an indication of spatial locality. JetStream uses the same memory prefetching and edge cache structures already built into the GraphPulse datapath. Since the active tasks (*events*) in JetStream are fewer and sparse in JetStream, it cannot harvest spatial locality as well as GraphPulse. As a result, the memory access utilization ratio is less than one-third of GraphPulse. However, having fewer computational tasks still makes JetStream significantly faster during incremental computation. Optimizing the memory access efficiency of JetStream is a potential avenue for future improvements.

Effects of Optimizations. We show the effects of the optimizations in terms of speedup over full recomputation in GraphPulse in Fig. 12. The baseline JetStream model is conceptually simple. However, without a mechanism to restrict tagging to only the affected vertices, it tags too many vertices in the graph, often leading to work comparable to full recomputation for most applications. VAP performs sufficiently well for SSSP and SSWP, but fails to provide a noticeable advantage for BFS and CC. The latter two applications have many vertices set to the same value, making the VAP optimization ineffective. DAP alleviates this problem and works well for all applications. However, VAP has the advantage over DAP in that it does not expand the event size to include source information.

Sensitivity to Batch Size. In Fig. 13, we have shown how the performance of the engine varies with different batch sizes. Taking a 100K batch size as the baseline, we showed the speed up for different batches for PageRank and SSSP running on LiveJournal graph. The speedup is based on JetStream's runtime for 100K batch size. JetStream speeds up significantly as the batch size gets smaller because it has little overhead for incremental data maintenance. JetStream can handle computations very fast for smaller batches where the number of changes or computations is low. JetStream's speedup grows orders of magnitude faster that KickStarter. This time is only the processing time, and the end-to-end performance may have other overheads to receive and batch the updates.

Sensitivity to Batch Composition. Edge deletions require more processing than edge additions in JetStream. An approximation phase is required to revert the effects of a deleted edge on the graph, which may propagate to many vertices for some critical edges. All the impacted vertices need to be reprocessed in the recomputation phase. Edge addition resembles regular events during the recomputation phase, and their effects are usually localized. Fig. 14 shows the effect of the composition of a batch on the run-time for SSSP and CC. Note that the run-times are normalized to JetStream's runtime for a 50-50 batch. An insertion-only batch converges 3 to 4 times faster on average than a deletion-only batch of the same size. Run-time increases as the ratio of deleted edges increases. KickStarter, too, demonstrates faster convergence with fewer deletions, but there is no concrete dependence of the run time on the ratio of deletions. KickStarter attempts to approximate the value of an impacted vertex before propagating the tag. JetStream attempts to minimize tagging using DAP optimization but only approximates after all tags are propagated. On the other hand, for PageRank and Adsorption in JetStream, the addition or deletion of one edge also mutates the other edges (weight) for a vertex, and both types of updates are handled similarly. Therefore, such algorithms are not noticeably affected by batch composition.

6.3 Hardware Cost and Power Analysis

We model JetStream using the same configuration as GraphPulse: 64MB on-chip memory for queue, and 8 processing pipelines with 2KB scratchpad and 1KB edge-cache on each. We use CACTI 7 [5] for power and area estimate for all memory elements. The queue memory is modeled in 22nm ITRS-HP SRAM logic. The biggest component of the communication network is a 16x16 NoC between

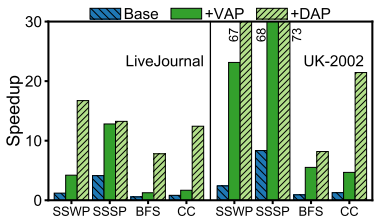

Figure 12: Speedup over GraphPulse for Baseline JetStream, VAP and DAP optimizations.

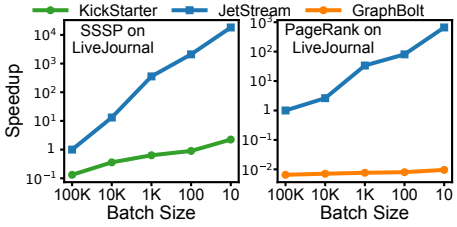

Figure 13: Sensitivity to batch size. Run-time shown as speedup over JetStream with 100K batch.

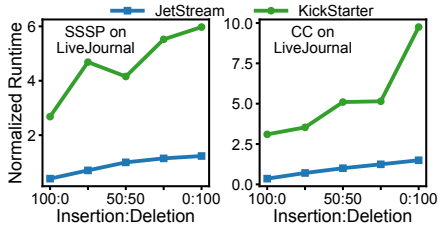

Figure 14: Run-time sensitivity to batch composition. Run-time is normalized to 50:50 composition on JetStream.

the event generation streams and the queues. Each port of the NoC is shared by several generator or queue ports.A breakdown of the total power and area estimate for the accelerator is shown in Table 4. The number in parenthesis is the increase over similarly configured GraphPulse. The overall increase in area and power is around 3% and 1% respectively.

JetStream reuses most architectural components of GraphPulse, including the event queue, prefetcher, and cache. Memory elements have the same physical size but contain fewer events due to the larger event size. As a result, there are some resource overheads due to larger buffers and interconnects. However, the dynamic energy is lower because JetStream processes fewer vertices propagating events (many vertices are already converged). Overhead from the buffers and communication buses also increases due to the larger event size. Floating point units account for the bulk of the processing and coalescing logic and remain the same in input size. Thus, the extra processing logic for JetStream adds only a small power and area overhead. The processing time in JetStream is shorter, making JetStream ~13 times more energy-efficient than full recomputation with GraphPulse. The total area of JetStream is about $200mm^2$ with a 28nm technology.

7 RELATED WORK

Software Frameworks for Streaming Graphs: A number of streaming graph frameworks have been developed that are based on the BSP [42] model similar to a software framework like Pregel [24]. Of these frameworks, Kineograph [8], Tornado [38], Naiad [29], and Tripoline [19] are limited to growing graphs. Kickstarter [45], GraphBolt [26], and DZiG [25] support both addition and deletion of edges. Maiter [50] is a graph analytics framework for delta-accumulative computation which is the basis of the event-driven model. There are also designs of graph representations to support high-throughput graph updates, such as Aspen [11], STINGER [12], and Version Traveller [20]. Other works handling changing graphs include GraphTau [16], Vora et al. [44], Chronos [14]. These works consider scenarios where graph versions are available a priori.

Accelerating Graph Processing: Template-based [3, 32] graph accelerators process hundreds of vertices in parallel to mask memory latency. Graphicionado [13] is a pipelined architecture that optimizes vertex-centric graph models using a fast temporary memory space to improve locality. GraphPulse [33] uses an event-driven model that expresses incremental updates as events. Swarm [18] allows speculative execution to increase parallelism and its extensions, Spatial Hints [17], improve memory locality using application knowledge to map tasks to processing elements. Chronos [1] provides another hardware acceleration framework based on speculative execution. Graph processing systems for FPGAs include ForeGraph [9], Zhou et al. [51, 52] etc. GraSU [46] provides the first FPGA-based high-throughput graph update library for dynamic graphs. PDES-A [34] is an FPGA-based accelerator for event-driven computation targeted at parallel discrete event simulation.

To efficiently handle vertex updates number of techniques have been proposed. Coup [48] exploits commutative-updates to reduce read and write traffic while PHI [28] exploits commutativity to coalesce updates in private cache to reduce on-chip traffic. HATS [27] proposes a hardware assisted traversal scheduler for locality-aware scheduling. Finally, there has been recent works that focus on architectures for PIM-based graph processing, such as Tesseract [2], GraphPIM [30], GraphP [49], and GraphQ [53].

Table 4: Power and area of the accelerator components

	#	Power(mW)			Area(mm^2)
		Static	Dynamic	Total	
Queue	64	117 (+1%)	20.7 (-6%)	8815 (~0%)	192 (+1%)
Scratchpad	8	0.35 (~0%)	1.2 (+6%)	12.1 (+4%)	0.21 (~0%)
Network		91 (+78%)	5.4 (+58%)	97 (+77%)	5.7 (+84%)
Proc. Logic		-	-	1.8 (+40%)	0.7 (+51%)
Total		-	-	8926 (+1%)	199 (+3%)

8 CONCLUDING REMARKS

We present the first accelerator for streaming graphs. JetStream extends a recently introduced event-driven accelerator, GraphPulse, to enable reuse of intermediate states to avoid a complete cold-start recomputation on the updated graph. JetStream supports edge additions and deletions for both monotonic and accumulative algorithms. It achieves average speedup of 13× over hardware accelerator and 18× over software frameworks at baseline batch sizes. This advantage increases substantially for small batch sizes.

ACKNOWLEDGMENTS

We would like to thank the paper's shepherd and reviewers for their helpful comments and suggestions. This work is supported in part by National Science Foundation grants CCF-1813173, CNS-1955650, CCF-2002554, and CCF-2028714 to the Univ. of California Riverside.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. *SIGARCH Computer Architecture News* 43, 3 (June 2015).

[3] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk. 2018. A Template-Based Design Methodology for Graph-Parallel Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 2 (Feb 2018), 420–430. https://doi.org/10.1109/TCAD.2017.2706562

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proc. International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[5] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2 (June 2017).

[6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW)*. 595–601.

[7] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Trans. on Parallel Computing (TOPC)* 5, 3 (2019).

[8] 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 *Proc. 7th ACM european conference on Computer Systems*. 85–98.

[9] 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 *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*. 217–226.

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Mathematical Software* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[11] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.

[12] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Proc. IEEE Conference on High Performance Extreme Computing*. 1–5.

[13] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[14] 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 *Proc. European Conference on Computer Systems (Eurosys)*.

[15] Jiewen Huang and Daniel J. Abadi. 2016. Leopard: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *Proc. VLDB Endowment* 9, 7 (March 2016).

[16] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proc. Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[17] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 5, 13 pages.

[18] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 228–241. https://doi.org/10.1145/2830772.2830777

[19] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proc. Sixteenth European Conference on Computer Systems*. 17–32.

[20] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. 2016. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 523–536.

[21] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proc. 19th International Conference on World Wide Web* (Raleigh, North Carolina, USA) *(WWW '10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[23] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[24] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD International Conference on Management of data*. 135–146.

[25] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proc. Sixteenth European Conference on Computer Systems*. 83–98.

[26] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proc. Fourteenth European Conference on Computer Systems*. 1–16.

[27] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proc. International Symposium on Microarchitecture (MICRO)*.

[28] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proc. International Symposium on Microarchitecture (Micro)*. 1009–1022.

[29] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[30] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[31] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[32] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proc. ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 166–177. https://doi.org/10.1109/ISCA.2016.24

[33] S. Rahman, N. Abu-Ghazaleh, and R. Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proc. 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 908–921. https://doi.org/10.1109/MICRO50266.2020.00078

[34] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2017. PDES-A: A parallel discrete event simulation accelerator for FPGAs. In *Proc. ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 133–144.

[35] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 37–42.

[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[37] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proc. European Conference on Parallel Processing (EuroPar)*. 319–333.

[38] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proc. International Conference on Management of Data*. 417–430.

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

[40] G. M. Slota, K. Madduri, and S. Rajamanickam. 2014. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *IEEE International Conference on Big Data (Big Data)*. 481–490. https://doi.org/10.1109/BigData.2014.7004265

[41] Amanda L Traud, Peter J Mucha, and Mason A Porter. 2012. Social structure of Facebook networks. *Phys. A* 391, 16 (Aug 2012).

[42] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[43] Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2014. Adaptive Partitioning for Large-Scale Dynamic Graphs. In *Proc. International Conference on Distributed Computing Systems*. 144–153.

[44] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–27.

[45] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proc. 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 237–251.

[46] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-Based Dynamic Graph Processing. In *Proc. International Symposium on Field-Programmable Gate Arrays*. 149–159.

[47] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[48] G. Zhang, W. Horn, and D. Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 13–25. https://doi.org/10.1145/2830772.2830774

[49] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053

[50] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2017. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *CoRR* abs/1710.05785 (2017). arXiv:1710.05785

[51] S. Zhou, C. Chelmis, and V. K. Prasanna. 2016. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *Proc. IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 103–110. https://doi.org/10.1109/FCCM.2016.35

[52] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. 2018. An FPGA Framework for Edge-centric Graph Processing. In *Proc. 15th ACM International Conference on Computing Frontiers* (Ischia, Italy). 69–77.

[53] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proc. International Symposium on Microarchitecture (Micro)*. 712–725.