MEGA Evolving Graph Accelerator

Chao Gao*
cgao037@ucr.edu
CSE Department, UC Riverside
USA

Mahbod Afarin*
mafar001@ucr.edu
CSE Department, UC Riverside
USA

Shafiur Rahman
mrahm008@ucr.edu
CSE Department, UC Riverside
USA

Nael Abu-Ghazaleh
nael@cs.ucr.edu
CSE Department, UC Riverside
USA

Rajiv Gupta
rajivg@ucr.edu
CSE Department, UC Riverside
USA

ABSTRACT

Graph Processing is an emerging workload for applications working with unstructured data, such as social network analysis, transportation networks, bioinformatics and operations research. We examine the problem of graph analytics over evolving graphs, which are graphs that change over time. The problem is challenging because it requires evaluation of a graph query on a sequence of graph snapshots over a time window, typically to track the progression of a property over time. In this paper, we introduce MEGA, a hardware accelerator designed for efficiently evaluating queries over evolving graphs. MEGA leverages CommonGraph, a recently proposed software approach for incrementally processing evolving graphs that gains efficiency by avoiding the need to process expensive deletions by converting them into additions. MEGA supports incremental event-based streaming of edge additions as well as execution of multiple snapshots concurrently to support evolving graphs. We propose *Batch-Oriented-Execution* (BOE), a novel batch-update scheduling technique that activates snapshots that share batches simultaneously to achieve both computation and data reuse. We introduce optimizations that pack compatible batches together, and pipeline batch processing. To the best of our knowledge, MEGA is the first graph accelerator for evolving graphs that evaluates graph queries over multiple snapshots simultaneously. MEGA achieves 24×-120× speedup over CommonGraph. It also achieves speedups ranging from 4.08× to 5.98× over JetStream, a state-of-the-art streaming graph accelerator.

CCS CONCEPTS

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

KEYWORDS

evolving graphs, iterative graph algorithms, common graph, redundancy removal, temporal locality, batch oriented execution

*Both authors contributed equally to this research.

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

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614260

ACM Reference Format:
Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. MEGA Evolving Graph Accelerator. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614260

1 INTRODUCTION

Graphs are fundamental data structures used to represent unstructured data, with objects as vertices and relationships as edges. Graphs arise across numerous application domains. Real-world graphs such as social networks and web graphs, are large and irregular, posing challenges for graph analytics workloads. Significant research has been conducted to create high-performance graph analytics frameworks for different platforms including CPUs, GPUs and custom accelerators to enhance performance and scalability [3, 4, 9, 16, 19, 23, 25, 26, 29–31, 37–39, 45, 50, 52, 58].

In real-world scenarios, graphs are frequently dynamic, as the data represented by the graph continues to change [43]. There are two primary categories of analyses for dynamic graphs: *streaming* graphs analytics and *evolving* graphs analytics. Streaming graph analytics continuously update query results as the graph changes due to incoming updates arriving in real-time. For example, one might want to maintain shortest paths to destinations as traffic conditions vary. We consider changes represented as edges being added or deleted from the graph (other changes such as adding and removing vertices can be modeled using edge additions and deletions as well). Incremental algorithms are typically utilized to update query results in response to the streaming graph changes, thereby avoiding the need to recompute the query from scratch with every update.

Our focus is the second type of analysis, *evolving graph* analytics, which aims to evaluate a query over a sequence of snapshots of the graph captured over an extended time period. In this case, the batches of changes were received in the past, and are already known. Generally, an evolving graph computation executes a query over a long time scale by analyzing different snapshots within the specified time window. For example, Covid-19 contact tracing data, represented as a graph of people that came in contact with each other, changes continuously as new contacts are reported, infection status of patients changes, and so on. Recent work exploits this temporal graph data to study characteristics such as number of contacts and infections over a time window, for example, after a certain variant appeared, or when a mitigation action such as

limiting mobility is introduced [53]. Having to evaluate the query on many snapshots makes the problem computationally expensive.

A number of algorithms and software systems have been proposed to support dynamic graphs. A simple approach is to evaluate the query independently on each snapshot; however, in the common case where the changes between snapshots represent a small fraction of the size of the graph and recomputing the full query is wasteful. Streaming uses incremental computation, starting from a fully computed graph snapshot, we move to a subsequent snapshot by *streaming* the edge additions and deletions incrementally updating the state of the graph [49]. Tegra [20] proposes to use these streaming algorithms, which are known to be substantially faster than redoing the computation from scratch, to support evolving graph computation by computing the initial graph, then using streaming/incremental computation to reach each snapshot in turn. Aspen [14] provides a data structure for storing dynamic graphs to support incremental computation.

In this paper, we propose MEGA, the first evolving graph accelerator. Algorithmically, MEGA starts from a recently proposed representation and processing model for evolving graph processing called CommonGraph [2]. For each group of snapshots, CommonGraph keeps an initial graph representing the edges that are common across all the snapshots (i.e., removing all edges that are either added or deleted). Starting from the CommonGraph, we can reach any snapshot simply by adding the set of edges that are missing. This approach has two primary advantages: (1) It gets rid of expensive edge deletion operations; and (2) It exposes significant parallelism by removing the sequential streaming dependency present in the streaming approach.

In terms of architecture, MEGA builds on a prior event-driven streaming graph accelerator, Jetstream [40], which employs event-driven asynchronous processing to support streaming. We show that directly implementing the two execution flows mentioned in the CommonGraph, namely Direct-Hop and Work-Sharing [2] using JetStream leaves significant opportunities for improving performance: we are unable to execute multiple snapshots concurrently; and we are unable to exploit reuse among the different snapshots. MEGA supports execution of multiple snapshots concurrently using a space efficient representation. We derive schedules to maximize data reuse using a data representation that supports all snapshots within the same graph. We also implement a number of other optimizations such as pipelining the execution of different snapshots, and support for operation on larger graphs, to further improve performance. MEGA outperforms the software implementation of CommonGraph by 12.3×-51.2×. It also achieves up to 4.08×-5.98× improvement in performance over JetStream.

The key contributions of our work are as follows:

- We present MEGA: the first accelerator for evolving graph workloads. MEGA provides support for multiple snapshots executing at the same time.
- We propose a new processing workflow for batch-oriented execution of identical batches across all snapshots. Batch-oriented execution exploits the similarity of the graph across snapshots to reuse similar edge-fetches and minimize redundant execution of batches compared to the Direct-hop and Work-sharing execution flows from CommonGraph.
- We explore optimizations to the workflow to improve concurrency such as allowing multiple concurrent batches, and using pipelining across batches to achieve additional speedups.
- We develop an event-driven datapath to support the overall execution flow. MEGA achieves 24-120x speedup over Software CommonGraph. It also achieves 4-6x improvement over JetStream.

2 BACKGROUND AND MOTIVATION

In this section, we describe the evolving graph problem and present the CommonGraph framework which we use as our starting implementation. We also present some motivating results to show opportunities for an accelerator to improve on CommonGraph.

2.1 Evolving Graphs and CommonGraph

Most real-world graphs change over time, leading to dynamic graphs [43]. Dynamic graph queries can be divided into two categories, **streaming graph** and **evolving graphs**. Streaming graph systems apply a query to the latest version of the graph as dynamically it evolves. They initially solve the query on the current graph, but perform incremental computation to update the solution as added and deleted edges stream in. On the other hand, evolving graph queries typically extract information from historical versions of the graph (called snapshots), for example tracking a property (e.g., the shortest path between two points) as the graph evolves.

A naive approach to processing evolving graphs is to execute the query on each instance independently, which can be inefficient since the snapshots can be substantially similar. Alternatively, its possible to leverage streaming, solving the query on the earliest snapshot and then use using streaming to compute subsequent snapshots one by one, leveraging known incremental streaming algorithms [15, 49]. Thus, existing evolving graph support has primarily focused on graph representations. For example, GraphOne[27] and Aspen[14] build representations to improve graph mutation (changing of the graph when new addition or deletions are introduced) to facilitate the construction and retrieval of multiple snapshots using one of the two approaches above.

Recently, a new abstraction for representing and processing evolving graphs called the CommonGraph[2] was introduced. CommonGraph provides new opportunities for parallelism, as well as more efficient execution workflows. For a group of snapshots to be processed, a CommonGraph represents the set of edges that will not be affected by either additions or deletions across all snapshots, and is therefore shared among all the snapshots. Consider ICG_1 in Figure 1(a) which is the common graph across snapshots G_i and G_{i+1}. To move from G_i to G_{i+1} using conventional streaming algorithms we have to add edges Δ^i_+ and delete edges Δ^i_-. The common graph, ICG_1 has all the edges common to both G_i and G_{i+1} (for example, all edges G_i excluding the deleted edges Δ^i_-, or alternatively all edges G_{i+1} excluding the added edges Δ^i_+). As a result, we can go from ICG_1 to either G_i or G_{i+1} by *adding* the missing set of edges (Δ^i_- or Δ^i_+ respectively).

CommonGraph offers a number of advantages [2]: (1) It eliminates the expensive deletion operations which require backpropagation and recomputation for many algorithms; and (2) It breaks the sequential dependency between snapshots present in streaming

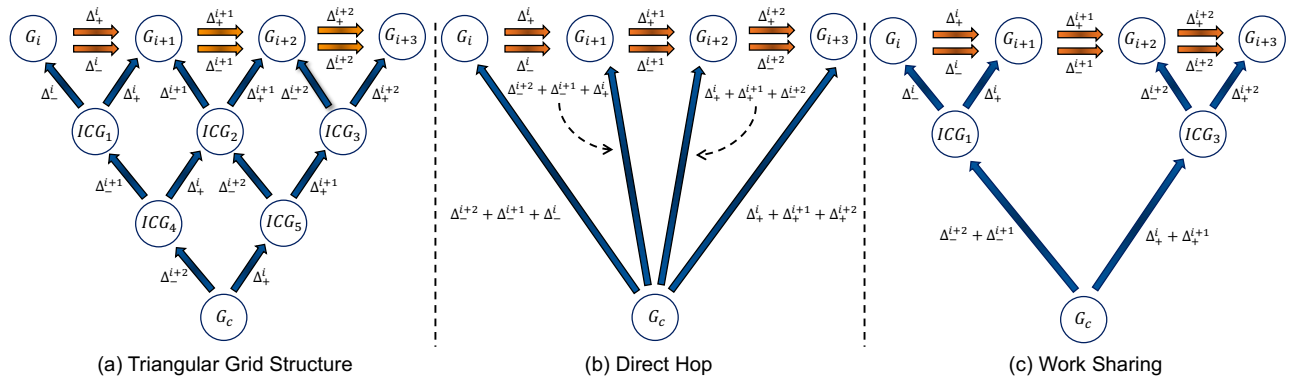


Figure 1: CommonGraph: Triangular Grid representation and alternative processing workflows.

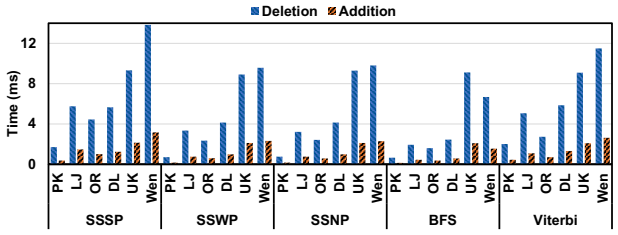


Figure 2: High cost of deletions in JetStream.

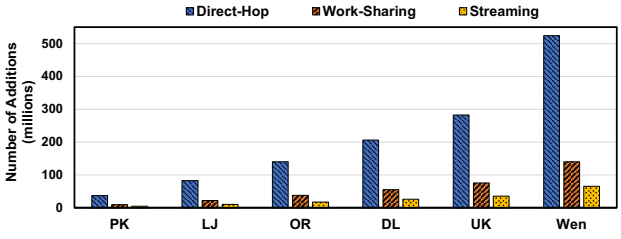


Figure 3: Number of additions in SSSP.

implementation of evolving graphs; we can go from the CommonGraph to any snapshot directly. We in Finding the common set of edges can be done recursively to create CommonGraphs that cover multiple snapshots. This recursive construction, which is called the Triangle-Grid structure (shown in Figure 1 (a) further provides parallel query opportunities. The paper considers two processing workflows: (1) Direct hop (Figure 1(b)) goes from the CommonGraph to each individual snapshot directly (potentially in parallel) and; (2) Work sharing (Figure 1(c) where intermediate CommonGraphs are used to avoid processing the same edge additions independently for each snapshot resulting higher work efficiency.

2.2 Motivating MEGA

CommonGraph represents the state of the art with respect to software evolving graph analytics, outperforming streaming implementations by up to 8x [2]. First, we verify whether one of the primary reasons behind CommonGraph's performance advantage, turning deletions into additions to avoid the high cost of deletion, also translates to streaming accelerators. Figure 2 shows the cost of processing a batch of edge additions vs. a batch of the same size of deletions when executed on the JetStream streaming accelerator [40]. Across many algorithms and graphs, deletions are substantially more expensive than additions, making it likely that CommonGraph will provide superior performance to streaming by replacing additions with deletions.

We next present some data to motivate some of the opportunities that are exploited by MEGA. We consider an evolving graph scenario with 16 snapshots. The size of the batch of edges additions or deletions to move from one snapshot to the next consists of 0.5 percent of the total edges in the graph, with an equal number of additions and deletions. Figure 3 shows the number of additions for *direct hop*, and *work sharing* CommonGraph processing strategies, as well as additions and deletions for baseline *streaming* for five different graphs and SSSP algorithm. *Direct hop* has 8 times more additions than streaming (scaling with $\frac{1}{2}$ the number of snapshots). While *work sharing* reuses edge operations across snapshots, the number of operations remains approximately double that of streaming. The reason is that some edge additions need to be repeated across different branches of the triangular grid. For example, note that in Figure 1 Δ_+^i is processed only once in streaming (from G_i to G_{i+1}), but since these added edges are part of all but the leftmost snapshot, it is processed twice in work sharing to cover all the snapshots (from G_c to ICG_3 and from ICG_1 to G_{i+1}). In conclusion, *CommonGraph* execution strategies, while eliminating expensive deletes, also increase the overall number of operations needed across all snapshots.

Finally, we show that CommonGraph results in poor locality as it processes snapshot by snapshot. Incremental graph processing used in streaming results in poorer memory locality than full evaluation of a query on a graph; only a small subset of edges are typically modified, resulting in poor spatial locality [8, 40]. CommonGraph processing workflows also lead to poor reuse as we apply different batches to a snapshot before moving on to the next (as can be

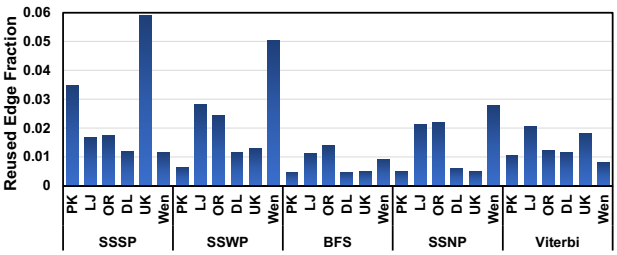


Figure 4: Edge reuse: different batches, same snapshot.

Figure 5: Edge reuse: same batch, different snapshots.

seen in Figure 4, there is very little edge reuse between fetched edges for different batches within the same snapshot. This low reuse motivated us to propose a batch-oriented execution workflow applying each batch to all the snapshots that need it. Since batches add the same edges to substantially similar graph instances, the execution workflow will result in high reuse in fetched edges. As can be seen in Figure 5, for the same batch applied to different snapshots the reuse is extremely high, on average, exceeding 98%.

3 MEGA DESIGN

MEGA uses the asynchronous execution model that has proven to be highly effective for static and streaming graph processing. As opposed to the Bulk Synchronous Parallel model, it achieves faster convergence and eliminates synchronization overhead at iteration boundaries. Additionally, its ability to reorder messages is leveraged to optimize utilization of memory bandwidth. MEGA builds upon Jetstream that implements event-driven asynchronous execution based on delta-accumulative incremental computation (DAIC), where delta-events arriving from different edges can be independently applied without any fixed order to compute the vertex state. In this model, lightweight messages known as events carry the deltas to their intended vertices. A vertex recomputes its state once receives an event (delta).

Consider an initial graph $G_0(V, E)$ obtained by solving the query on an initial graph snapshot. A streaming algorithm takes an incremental edge-batch $E < src, dst, wt, add/del >$, and incrementally updates the solution in $G_0(V, E)$ resulting in a modified graph $G_1(V, E)$. A straightforward strategy for using Jetstream in the evolving graph scenario is to use streaming to solve the query one snapshot at a time in sequence. This approach has a number of limitations: (i) we are restricted to solving one snapshot at a time; (ii) deletions that are considerably more resource-intensive and

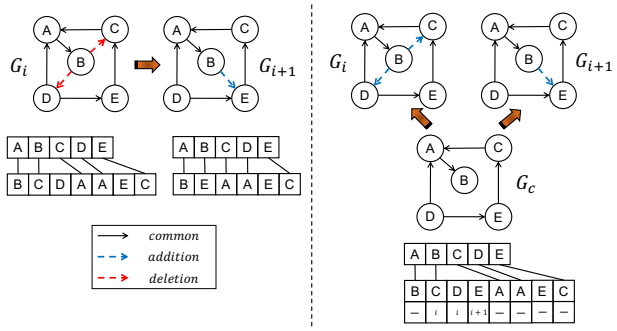


Figure 6: Unified Evolving Graph CSR Representation.

computationally expensive must be processed; and (iii) streaming algorithms are known to have poor locality [8, 40].

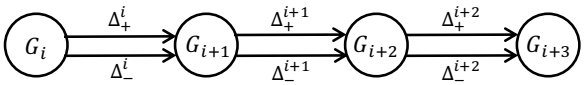
By building our work on CommonGraph we achieve deletion-free processing. However, straightforward use of CommonGraph on Jetstream still leaves two unresolved issues: serial processing of snapshots; and poor graph locality. To overcome these issues we introduce *batch-oriented-execution* (BOE). BOE employs a compact and *unified evolving graph representation* that allows query evaluation on multiple snapshots in a memory locality aware fashion.

An example of this representation is shown in the Figure 6. The left half of the figure shows two snapshots (G_i and G_{i+1}) and their CSR representations. The right half shows the CSR representation of the union of edges in G_i and G_{i+1} and an additional array which for each edge contains: "-" if it belongs to the CommonGraph G_c; "i" if it belongs to the additions batch resulting in snapshot G_i; and "i+1" if it belongs to the additions batch resulting in snapshot G_{i+1}. In other words, this unified graph representation contains G_c, G_i, and G_{i+1}.

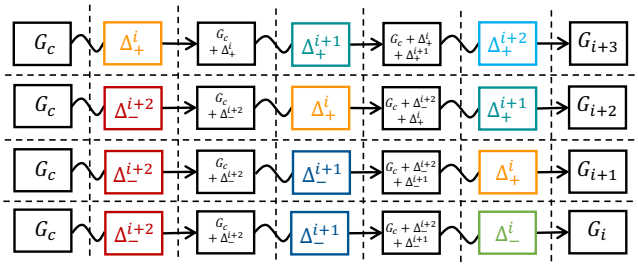
Creating the common graph representation is straightforward; it involves removing the deleted edges present in all batches from the initial graph. We measured this cost to be around 10% of the average SSSP query execution time in Risgraph [15]. However, we assume that the unified graph representation is the default storage format for our system, making this an offline cost.

3.1 Batch-Oriented-Execution

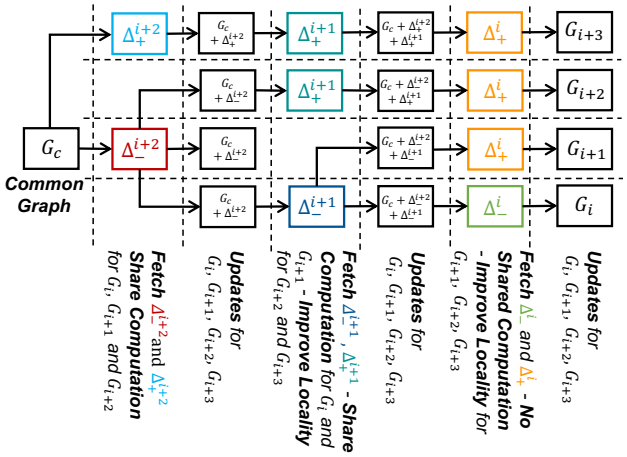
Consider a series of snapshots $G_i, G_{i+1}, G_{i+2}, G_{i+3}$, created by additions and deletions as shown in Figure 7(a). To solve a query on Jetsream, first the query is solved on G_i and then its results are incrementally updated to obtain results for G_{i+1} and so on till results for evaluations have been computed. Figure 7(b) shows deletion-free query evaluation by first evaluating it on the CommonGraph G_c and then incrementally applying batches of *additions* till results for G_i are obtained. Starting from G_c and repeating the above process with appropriate batches of additions, we can also obtain results for G_{i+1}, G_{i+2}, and G_{i+3}. We observe that even though this approach eliminates the processing of expensive deletions, it computes results for one snapshot at a time, which causes two inefficiencies: *redundant computation*; and *poor locality*. Consider the incremental update of results for G_c following the red batch of additions represented by Δ^{i+2}. In Figure 7(b), this computation is performed three times, resulting in redundant work. Consider the use of the orange



(a) Query Evaluation in Sequence (Kickstater).



(b) Deletion-Free Evaluation using CommonGraph.



(c) Batch-Oriented-Execution using CommonGraph.

Figure 7: Strategies for Incremental Query Evaluation.

batch of additions Δ_+^i that also takes place three times – first for G_{i+3}, then for G_{i+2}, and finally for G_{i+1}. Since the three uses take place at different times, their accesses lack of temporal locality.

Batch-oriented-execution (BOE) shown in Figure 7(c), eliminates redundant work and poor temporal locality, while *maximizing parallelism* by simultaneously computing the results of all four snapshots. First, let us consider the removal of *redundant work* via BOE. The first step computes the query on G_c and these results are used as the starting point for all snapshots. Next, we see that additions batch Δ^{i+2} is used by three snapshots – G_i, G_{i+1}, and G_{i+2}. Therefore, we incrementally update the results of query for G_c using Δ^{i+2} once and then use the results of this *shared computation* to update the results of three snapshots G_i, G_{i+1}, and G_{i+2}. The results computed in the preceding step are then incrementally updated using additions batch Δ^{i+1} and used to update results of snapshots G_i and G_{i+1}, resulting in further elimination of redundant work.

Second, let us observe how poor locality is eliminated by BOE. Note that at each incremental update in the schedule representing BOE, whenever an addition batch is to be used by more than one

Algorithm 1 Generating MEGA Execution Schedule

```
1:  Inputs: Common Graph (G_c);
2:  Addition Edges Batches: Δ+_1, Δ+_2, Δ+_3, ..., Δ+_{n-1};
3:  Deletion Edges Batches: Δ-_1, Δ-_2, Δ-_3, ..., Δ-_{n-1}.
4:  Output: Schedule for Computing Results for G_0, G_1, ..., G_{n-1}.

5:  function MEGA-EXECUTION-SCHEDULE
6:      for i in range of [N − 2, 0] do
7:          parallel for (Δ+_i and Δ-_i) do
8:              UPDATE-QUERY (Δ+_i, add)
9:              UPDATE-QUERY (Δ-_i, del)
10:         end for
11:     end for
12: end function

13: function UPDATE-QUERY(Δ_j, batch_type)
14:     if batch_type == add then
15:         parallel for i in range of [N − 1, j + 1] do
16:             GEN [val-array[i] = INCREMENTAL-QUERY (i, Δ_j)]
17:         end for
18:     else if batch_type == del then
19:         GEN [temp_val = INCREMENTAL-QUERY (i, Δ_j)]
20:         parallel for i in range of [j, 0] do
21:             GEN [val_array[i] = temp_val]
22:         end for
23:     end if
24: end function
```

snapshot, the computation for the snapshots are performed at the same time. That is, multiple users of an additions batch access the batch simultaneously creating temporal locality.

Therefore, BOE delivers maximal parallelism, minimal redundant work, and maximal temporal locality. We have developed a general algorithm for the *offline generation* of the BOE schedule for N snapshots as shown in Figure 8. In Algorithm 1, `GEN [...]` statements generate the calls to incremental update of query results following addition of a batch of edges. For simplicity, we have not explicitly identified the graphs but rather only identified the incremental query updates that are performed. Note that in some cases, incremental updates on different versions of a graph can be performed in parallel. Additionally, in Figure 8, for N snapshots there are N-1 stages in the schedule, and each stage has exactly two addition batches. The loop in function `MEGA-EXECUTION-SCHEDULE` iterates N-1 times handling a pair of addition and deletion batches for which `incremental-Query` evaluations are generated by function `update-Query`. Lines 14-17 handle addition batches, while lines 18-23 handle deletion batches.

3.2 Other Optimizations

Locality for Partitioned Graphs. We have shown how BOE accommodates multiple snapshots along with a value array that holds values for all snapshots corresponding to each vertex. We map the on-chip memory in MEGA to node properties using a direct-mapping format. However, as the graph sizes increase, eventually graph partitioning becomes necessary. As demonstrated in Figure 5,

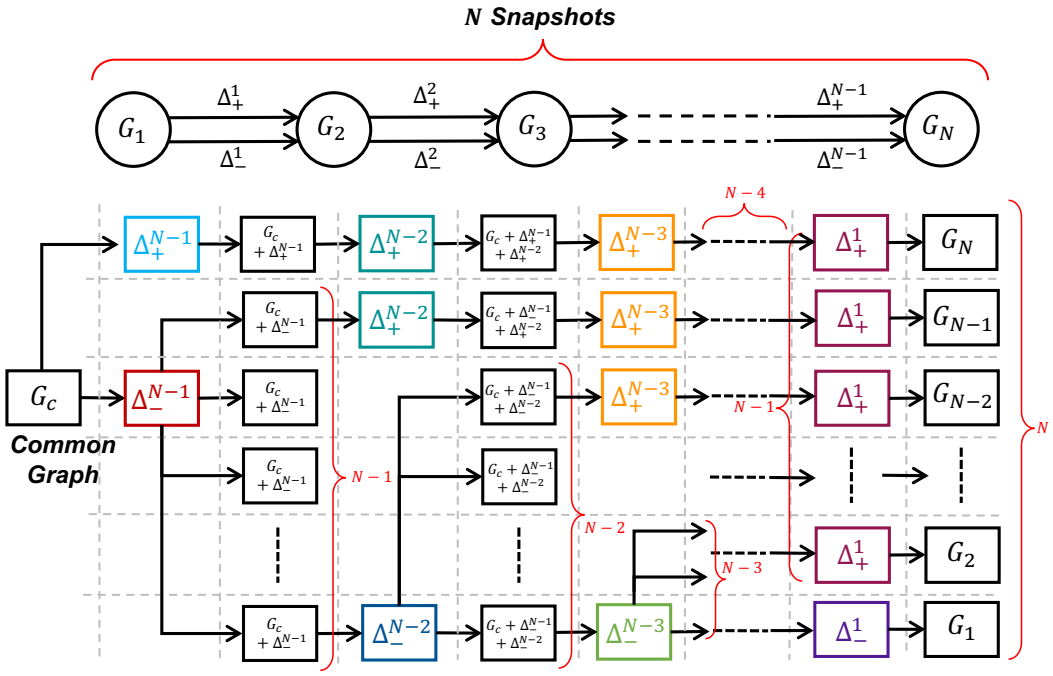


Figure 8: BOE: (Left) Offline Schedule Generation Algorithm; (Right) BOE Schedule for N Snapshots.

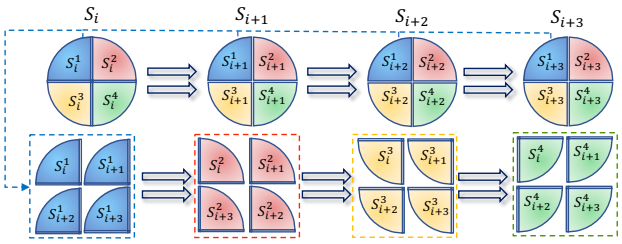


Figure 9: The top part illustrates CommonGraph execution as it transitions from one snapshot to the next. The bottom illustrates scheduling in MEGA where we execute the same batch on all the snapshots that need it at the same time, resulting in high locality. In this example, we cannot fit all 4 snapshots on the accelerator so we apply the batches to the same partition of the graph across different snapshot concurrently.

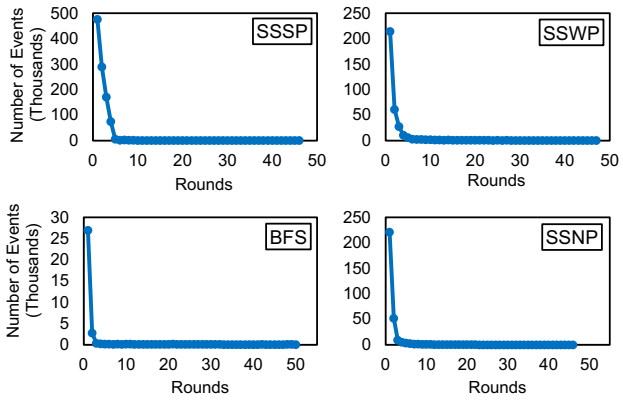


Figure 10: Number of events for each round for four algorithms (Wen graph using JetStream); number of events drops rapidly during the initial rounds.

with BOE around 98% of edges fetched across different snapshots are the same. To capitalize on this observation, we propose a partition-scheduling approach, as depicted in Figure 9, to enhance locality in presence of partitioning. Assuming we have four snapshots and only one snapshot can fit on-chip at a time, we divide the snapshots into four separate partitions. At the start of the computation process, we will retrieve partition 1 for all four snapshots and store it in the on-chip memory. Once the first partition's computation is complete, we move to the next partition and so on. This approach exploits temporal locality of BOE even when the graph requires partitioning to fit multiple snapshots.

Batch Pipelining (BP). We define one *round of computation* (also referred to as one hop incremental computation in the previous Figure 1(a)) as an execution, which comprises multiple iterations of computation. The illustration in Figure 10 clearly demonstrates that the number of events occurring during a single execution decreases as the number of rounds increases, creating long tails where the capacity to manage more events is available. As we approach the "long-tails" of a single execution, we can introduce another execution into the accelerator. Rounds in the asynchronous

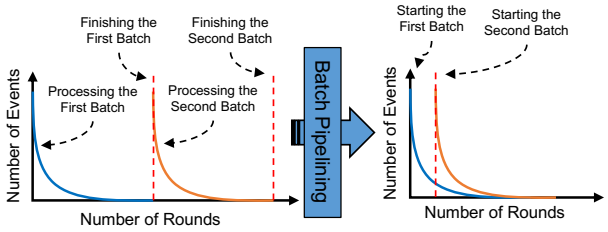


Figure 11: The figure shows Batch Pipelining for two addition batches in Batch-Oriented-Execution scenario.

model correspond to iterations in synchronous graph processing, and the set of active events correspond to the frontier. Note that the later rounds have fewer events and therefore are processed quickly; however, there remains an opportunity to overlap the execution time of the "tails" with initial rounds from another batch execution, to improve parallelism as shown in Figure 11. The initiation of a new execution fed to the hardware accelerator is triggered when the events number decreases to a specific threshold. Note that this trigger can be easily supported in hardware. This process effectively eliminates the extended tail.

Generality: Although we demonstrate BOE in the context of an asynchronous accelerator, the observation that applying a batch to all the instances together improves locality is independent of the execution model. In addition, the order of applying the batches does not affect the correctness of the final result provided that the incremental update algorithm is correct since the final graph is the same regardless of the order of edge additions. Algorithm 1 is also independent of the execution model.

4 MEGA ARCHITECTURE

MEGA uses an event-driven execution to support operations on dynamic graphs, similar to the GraphPulse and Jetstream accelerators [39, 40]. Event driven execution offers a number of advantages over bulk-synchronous processing, and is especially suited for dynamic graphs where graph changes can be expressed as events. At a high level, MEGA incorporates a number of ideas to support efficient processing of evolving graphs: (1) Operation on multiple versions of the graph concurrently to improve parallelism and data reuse; (2) Batch-oriented execution to reuse computation and memory, and optimize scheduling of the graph processing; and (3) Pipelining between different versions of the graph, enabling one dependent version to start before the snapshot it depends on has fully stabilized.

4.1 MEGA Architecture Overview

Figure 12 shows an overview of the datapath. The primary datapath components include Event Queues, Event Scheduler, Processors, and the on-chip routing network that interconnects these elements. During full operation on a graph, all computation is represented as events represented as lightweight event messages. An event triggers computation at the destination vertex and multiple events targeted towards the same vertex are coalesced in the event queues. Events messages are tuples consisting of a target vertex identifier, a payload, and specific flags used to indicate special purpose events, such as those used to support edge deletion. The event queue is composed of multiple individual bins, each containing events for a subset of vertices, to improve both queuing and dequeuing bandwidth. Event processors use parallel event generation streams to assist in generating outgoing events, considering that some vertices may have a large number of outgoing edges in a power-law graph.

MEGA's computational model follows the event based processing model introduced by GraphPulse [39] and later adapted for processing streaming graphs in Jetstream [40]. We first carry out the computation on the common graph, which is shared among all the snapshots. For each batch, the batch reader reads the edges for the batch and creates corresponding events for each of the active snapshots for this batch and inserts them into the event queues for execution as described next.

4.2 Execution and Datapath

MEGA supports multiple active snapshots: events are marked with a version tag to allows separation of events destined to different snapshots. We also add a batch tag, in order to be able to detect when a batch is over to support batch scheduling. The event queue is a central structure in MEGA, holding all active events within the system. This queue is designed with multiple sub-queues (or bins) to improve the bandwidth of queuing and dequeuing, and to support partitioning. Changing active partitions/snapshots is carried out at the granularity of bins, partitions being swapped out can be streamed from their bin to memory, and newly activated partitions are streamed from memory to available bins. Each bin is organized as a direct mapped matrix of rows and columns, with each cell representing a vertex for a specific graph snapshot, like a direct-mapped cache.

When inserting events in the event queue, the decoder in Figure 13 identifies the location of the event based on its version id. The queue is dual-ported and pipelined, allowing for one read and one insertion per cycle. During insertion, if an existing event is detected in the target cell, the events are coalesced using a reduction operation such that each vertex has at most one active event (coalescing is part of the insertion pipeline and does not cause additional delays). This design gains efficiencies by reducing the storage and processing for events and also removes the need for synchronization with at most one event for each vertex.

Since MEGA supports multiple active graph versions concurrently, it is important to schedule execution in a way that promotes data reuse. The different versions share most of the graph structure (Figure 6), benefiting from data reuse when they are accessing similar parts of the graph. However, their active state, consisting of events and vertex values, must be maintained separately once the snapshots diverge, causing the events to be stored in different parts of the event queue enabled by the decoder logic in Figure 13.

To accommodate multiple batches, all instances on which the batches operate must be resident in the accelerator (which is ensured by the Batch Scheduler). Once the execution starts, the implementation is straightforward since the snapshots are independent and events/snapshots are isolated by version tags. Note that it is possible that multiple events for the same vertex/snapshot would

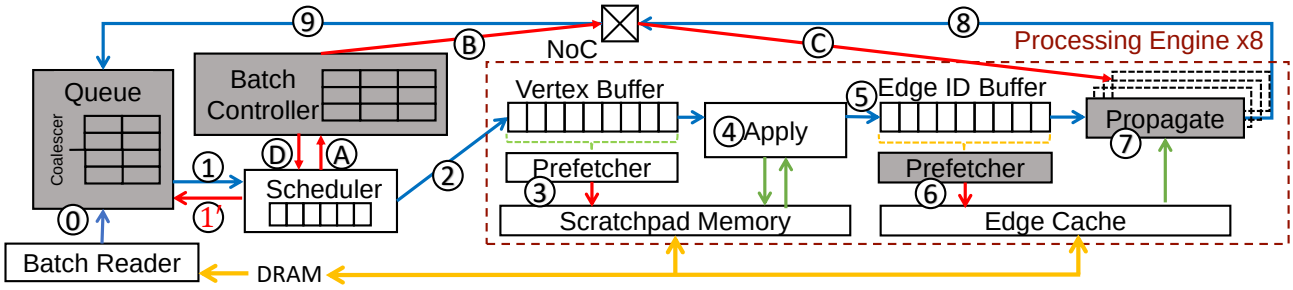


Figure 12: MEGA datapath: blue lines indicate data-flow; red represent control signals; and green/yellow signify on-chip and off-chip memory transfers respectively.

be generated from different batches that are concurrently active, for example due to batch pipelining. However, since the events target the same snapshot, they can safely be coalesced. The asynchronous execution model ensures correct execution regardless of event order.

The overall event execution proceeds as shown in Figure 12. When a new batch is scheduled, the batch reader brings the batch in from off chip and generates corresponding update events to all the snapshots needing that batch. These events are inserted into the event queue. MEGA's processing engines first pull events from their queues after the event scheduler places them there. The Batch-Reader first reads one batch of additions and generates their corresponding events (Step ⓪). Next, the scheduler will pull events from the Queue, and the Queue emits events in Step ①, and places them in the vertex buffers in Step ②. Event execution requires reading the vertex state (which is prefetched) in ③. In Step ④, the edge computation representing the algorithm is executed to update the vertex state (see Table 1 for the computation function corresponding to the different algorithms). The PE fetches the output edges from the edge cache for generating output events in Step ⑤. If the outgoing edge set is not cached, it is prefetched prior to event execution in Step ⑥. Outgoing events are generated in Step ⑦ to the respective snapshots using 4 parallel event generation units for each processing element to reduce delays associated with executing events on high out-degree vertices.

4.3 Batch Scheduling and Version Control

The batch scheduling logic implements the execution workflow of the accelerator. It controls which batches are active on which instances/partitions of each instance of the graph. Along with the event scheduler, this ensures that the batch processing for all the instances proceed at an even pace. This logic also manages the allocation of event bins to implement workflow schedules such as that shown in Figure 9. The schedule of these allocations is pre-determined as described in Section 3.1.

As illustrated in Fig 7, all snapshots are composed of a common graph and a sequence of addition-only batches. To manage these snapshots, MEGA's computation scheduler includes a hardware version table: a look-up-table containing information about the composition of different snapshots and their processing status. When a computation batch begins, the scheduler marks its entry in

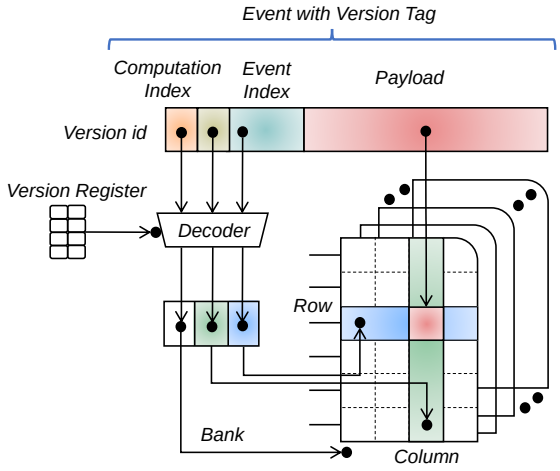


Figure 13: Queue support for multiple snapshots.

the version table as active (Step Ⓐ in Figure 12) The version table broadcasts updates to all processing elements (PE) and event-queue banks (Step Ⓑ and Step Ⓒ), updating the version register in the PEs. To support the Batch Oriented Execution workflow, we schedule all the active snapshots for each batch together to promote spatial locality. Once the scheduler identifies that a batch is entering the long-tail phase based on event queue occupancy, the version table updates other batches and notifies the scheduler through Step Ⓓ to initiate a new computation batch.When events from different instances are destined for the same vertex, edge prefetching is done by the first event destined to the vertex, but is reused by subsequent snapshots. The event generation streams are interconnected with the queues via a network on a chip implemented as a 16x16 crossbar with each port shared among two of the 32 event generators (four per PE). On the other side, output ports of the NoC lead to the event bins where the newly generated events get queued for future execution (Steps ⑧ and ⑨ in Figure 12).

The MEGA datapath is based on that of the JetStream streaming accelerator; the components in grey in Figure 12 are either new or modified. Specifically, JetStream works on a single graph at a time

and supports both edge additions and deletions. MEGA supports the unified graph representation, BOE scheduling, and multiple active graph instances (reflected in queue design, prefetcher design, event generation and propagation, as well as the NoC). Moreover, since MEGA uses CommonGraph to eliminate the need for edge deletions, we remove the expensive event deletion logic.

5 PERFORMANCE EVALUATION

In this section, we evaluate MEGA's performance and overheads. We first describe our experimental setup.

Table 1: Benchmarks and their edge functions.

Algorithm	EdgeFunction $(e(u,v))$
BFS	$CASMIN(Val(v), min(Val(u)+1, val(v)))$
SSWP	$CASMAX(Val(v), min(Val(u), wt(u,v)))$
SSNP	$CASMIN(Val(v), max(Val(u), wt(u,v)))$
SSSP	$CASMIN(Val(v), Val(u)+wt(u,v))$
Viterbi	$CASMAX(Val(v), Val(u)/wt(u,v))$

Table 2: Edges and Vertices of the Input Graphs and the Batch Size for Motivation Data.

Input Graph	Edges	Vertices	Batch Size
Pokec (PK) [47]	30M	1.6M	0.3M
LiveJournal (LJ) [6]	70M	4M	0.7M
Orkut (OR) [33]	117M	3M	240K
DBpediaLinks (DL) [5]	170M	18M	1.7M
UK-2002 (UK) [10]	260M	18M	2.6M
WikipediaLinks (Wen) [28]	400M	13M	4M

Table 3: Experimental Configurations.

	CPU	MEGA
Compute Unit	60× Intel(R) Xeon(R) @3.1GHz	8× MEGA Processor @ 1GHz
On-chip memory	49.5MB L3 Cache	64MB eDRAM @22nm 1GHz, 0.8ns latency
Off-chip Bandwidth	14x DDR4 17GB/s Channel	4x DDR4 17GB/s Channel

5.1 Experimental Setup

System Modeling: We implemented the MEGA accelerator on a cycle-accurate microarchitectural simulator built using the Structural Simulation Toolkit (SST) [46]. The off-chip memory is modeled using DRAMSim2 [41]. The simulator incorporates a cycle accurate model of the NoC, scratchpad memory, cache hierarchy, event queues and other components of the data path.

Workloads We evaluate accelerator performance using five commonly used graph algorithms listed in Table 1 and six real-world input graphs listed in the Table 2. We synthesize 16 snapshots of all the datasets by randomly creating batches consisting of 1% of the edges (half additions and half deletions) to mimic the evolution of the graph. We validated the final results of MEGA executions against those of the software baselines.

Software and Hardware Baselines: For the Software baseline, we choose the streaming systems Kickstarter [49] and Risgraph[15]. We also compare against a GPU system system, Subway [42], which uses an asynchronous execution model similar to our accelerator. We implement CommonGraph within each of these baselines [2]. We execute these on a shared memory system on Google Cloud with C2-standard-60 compute node which has 60 Intel(R) Xeon(R) CPU processors and 240GB of memory. For the hardware baseline design, we use the same configuration outlined in the Jetstream paper [40], and we configure MEGA to support two execution flows: Direct-Hop and Work-sharing from CommonGraph [2]. For GPU experiments, we used NVIDIA Tesla K80 GPUs with 12 GB GDDR5 memory, and code was compiled with CUDA 10.2, utilizing the highest optimization level.

5.2 Performance and Characteristics

Overall Performance. Figure 14 shows the overall speedup achieved by MEGA over software implementations of CommonGraph (work sharing) implemented within different streaming systems, Kickstater, RisGraph and Subway (GPU) [15, 42, 49]. The scenario consists of executing 16 snapshots, each involving a 1% change in the graph with an equal distribution of 50% edge additions and 50% edge deletions. MEGA with BOE outperforms CommonGraph on Kickstarter and RisGraph by 51x and 29x respectively. The results from Table 4 and Figure 14 include all the partitioned graph overheads to move partitions on/off chip as discussed in Section 3.2. MEGA requires more graph partitions compared to Jetstream to support BOE on multiple snapshots concurrently. For example, with Live Journal, Jetstream does not require graph partitioning while MEGA needs to partition the graph into four parts. MEGA outperforms Subway, the GPU baseline, by an average of 12x. It is important to note that we configured MEGA with conservative memory settings with total memory bandwidth of 68GBytes/s, which is less than a third of the bandwidth available on the K80 GPUs (240 GBytes/s). To show the performance improvements from BOE in software, we implemented a version of it on the top of RisGraph as shown in Figure 14. The software version of BOE exploits parallelism from concurrent snapshots execution, but uses different processors and is not able to exploit memory locality effectively.

Table 4 compares the performance of MEGA to Jetstream as well as to different execution workflows. The first line for each graph shows the run time on the JetStream processor using streaming. The next two lines show the speedup obtained in MEGA when implementing the CommonGraph Direct-hop (DH) and Work-sharing (WS) execution flows. The final three lines show the speedup

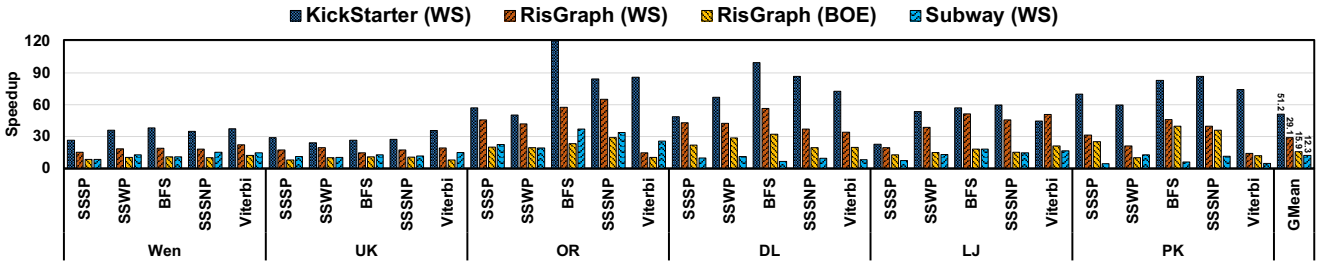


Figure 14: MEGA (BOE+BP) speedup over CommonGraph Work-Sharing implemented on top of KickStarter (software) [49], RisGraph (software, both WS and BOE) [15], and Work-Sharing on Subway (GPU) [42].

Table 4: Average Execution Time for JetStream, and the speedup of CommonGraph Direct-Hop, CommonGraph Work-Sharing, Batch-Oriented-Execution with Batch Pipelining optimizations over JetStream for 16 Snapshots.

Graph	Query Evaluation Algorithm	BFS	SSSP	SSWP	SSNP	Viterbi
PK	JetStream Time	21.31ms	77.02ms	21.52ms	24.75ms	73.81ms
	Direct-Hop Speedup	2.14×	1.70×	2.11×	1.66×	1.44×
	Work-Sharing Speedup	2.26×	1.88×	2.17×	1.86×	1.75×
	BOE Speedup	4.95×	4.93×	4.68×	4.94×	4.94×
	BOE + BP Speedup	5.03×	5.66×	4.94×	5.64×	5.98×
LJ	JetStream Time	58.574ms	262.425ms	115.725ms	101.171ms	156.522ms
	Direct-Hop Speedup	2.11×	1.91×	1.86×	1.31×	1.24×
	Work-Sharing Speedup	2.17×	1.96×	1.94×	1.68×	1.64×
	BOE Speedup	4.16×	4.67×	4.63×	4.47×	4.5×
	BOE + BP Speedup	4.43×	4.99×	5.13×	5.01×	5.17×
DL	JetStream Time	71.401ms	211.725ms	136.821ms	133.275ms	223.501ms
	Direct-Hop Speedup	1.09×	1.21×	1.04×	1.05×	1.24×
	Work-Sharing Speedup	1.55×	1.62×	1.52×	1.52×	1.73×
	BOE Speedup	4.36×	4.56×	4.48×	4.46×	4.54×
	BOE + BP Speedup	4.63×	5.7×	4.98×	4.96×	5.83×
OR	JetStream Time	39.451ms	107.475ms	114.015ms	67.875ms	79.275ms
	Direct-Hop Speedup	1.59×	1.41×	1.30×	1.07×	1.21×
	Work-Sharing Speedup	1.83×	1.74×	1.67×	1.53×	1.69×
	BOE Speedup	4.14×	4.36×	4.45×	4.25×	4.36×
	BOE + BP Speedup	4.25×	5.3×	5.08×	5.02×	5.03×
UK	JetStream Time	248.852ms	246.151ms	250.125ms	255.901ms	248.625ms
	Direct-Hop Speedup	2.05×	1.92×	2.01×	1.31×	1.98×
	Work-Sharing Speedup	2.15×	1.98×	2.11×	1.61×	2.11×
	BOE Speedup	4.85×	4.83×	4.84×	4.82×	4.81×
	BOE + BP Speedup	5.19×	5.08×	5.22×	4.96×	4.95×
Wen	JetStream Time	170.252ms	418.651ms	282.075ms	277.052ms	320.925ms
	Direct-Hop Speedup	1.73×	1.09×	1.34×	1.27×	1.30×
	Work-Sharing Speedup	2.01×	1.66×	1.88×	1.85×	1.87×
	BOE Speedup	3.74×	4.2×	4.07×	4.03×	4.05×
	BOE + BP Speedup	4.08×	4.53×	4.61×	4.48×	4.47×

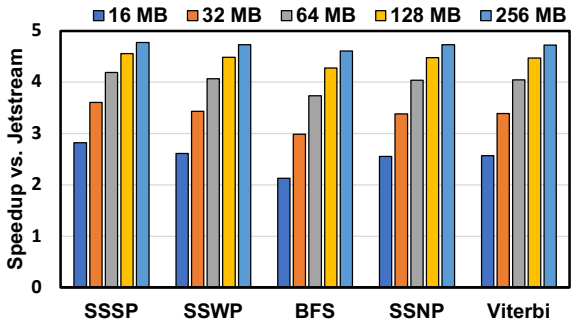


Figure 15: Effect of on-chip memory size (Wen Graph)

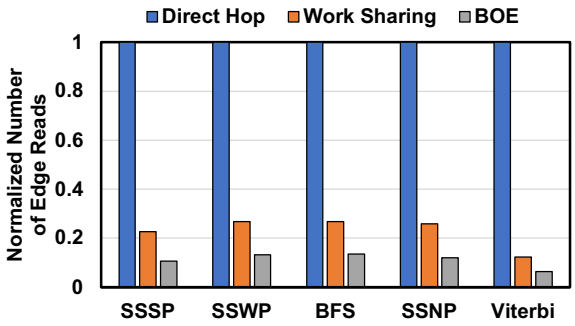


Figure 16: Normalized edge reads (Wen Graph).

achieved by BOE, with single-batch, multiple-batch, and multiple-batch with pipelining respectively. For all workflows, MEGA substantially outperforms Jetstream because of the advantage of eliminating expensive deletions. WS outperforms DH, as was also observed in software, because it reduces the overall number of executed events. BOE outperforms WS because it is able to achieve significantly better memory reuse, gain from concurrent execution of batches, while also achieving work sharing.

Sensitivity to on-chip memory size: Since MEGA executes multiple instances of the graph at the same time, when on-chip memory is limited, it must partition each instance of the graph. This incurs additional overheads as events for inactive partitions are saved to memory and later brought in when the target partition is loaded. Figure 15 shows that as the on-chip memory size increases, performance improves since larger graph partitions can fit on chip. We configured MEGA with 8 PEs; adding additional PEs did not improve performance without increasing the memory bandwidth as well as internal bandwidth of the NoC and event queues.

Memory reuse: Figure 16 shows the number of edge reads during run time for the different execution workflows. Edge reads increase with the number of events processed, but go down when there is significant reuse of the edges. Direct hop executes a very high number of events, resulting in a high number of edge reads.

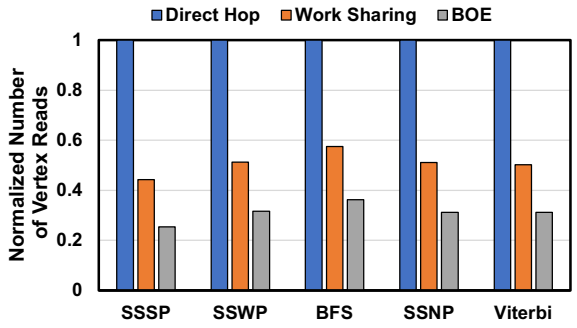


Figure 17: Normalized vertex reads (Wen Graph).

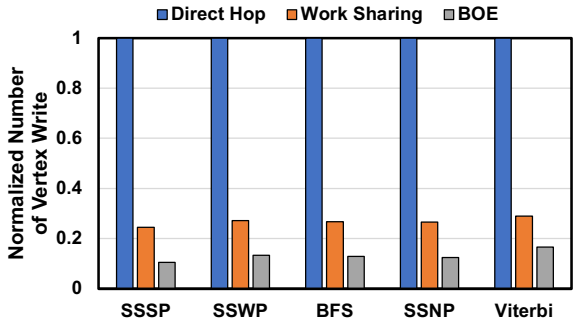


Figure 18: Normalized vertex writes (Wen Graph).

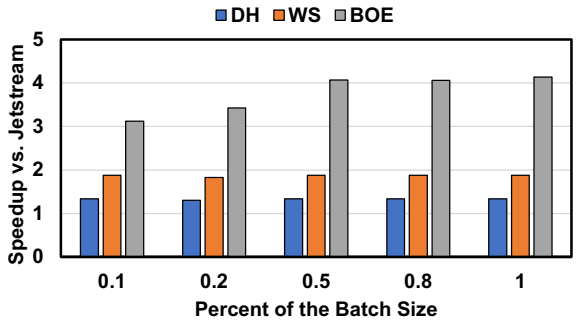


Figure 19: Effect of batch size (Wen/SSWP).

While work sharing executes less events, there is low locality between events. BOE has the lowest number of edge reads, due to the high reuse achieved by the batch oriented scheduling. We see similar trends also for the vertex reads (Figure 17) and the vertex writes (Figure 18). Since batch oriented scheduling applies the same batch to slightly different versions of the graph, it can achieve high reuse in both vertex and edge operations.

MEGA Scalability Analysis: The next experiment provides insights into how well the system can handle changes in workload,

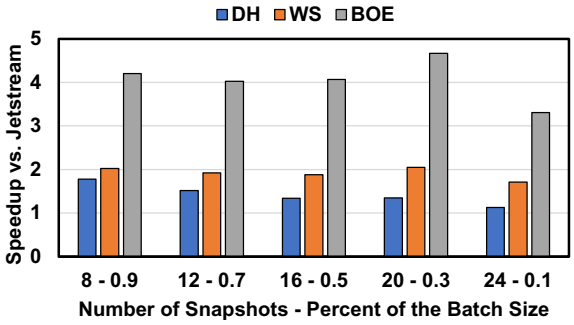


Figure 20: Effect of number of snapshots (Wen/SSWP).

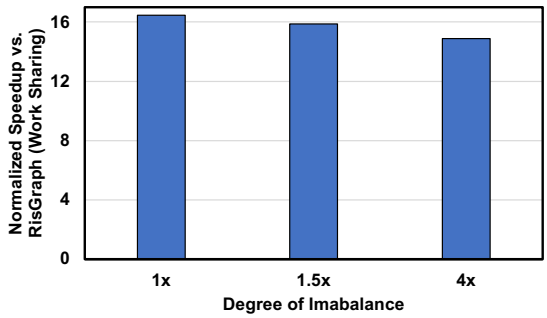


Figure 21: Effect of imbalance batches (Wen/SSWP).

with respect to the batch size and the number of snapshots. We vary the batch size from 0.1% to 1%. Figure 19 shows that MEGA consistently outperforms CommonGraph across the range of batch size, with the advantage increasing for larger batches.

Next, we vary the number of snapshots within the fixed time window. The results, as shown in the Figure 20, indicate that when there are fewer than 20 snapshots, MEGA achieves a higher speedup. However, when the number of snapshots increases to 24, MEGA's performance slows down compared to the other execution flows. This slowdown occurs because, as more snapshots are processed in MEGA, the overhead of graph partitioning becomes higher, negatively impacting performance. Finally, we study the effect of batch size imbalance on the performance of BOE in Figure 21. The first value represents the speedup when the batches are identical in size. An imbalance of 1.5x (or 4x) means that the largest batch is 1.5 times (or respectively 4 times) the size of the smallest batch. We see that speedup dips slightly, by about 10% even when large imbalance is present.

5.3 Hardware Cost and Power Analysis

We build a model of the primary MEGA resources sized similar to Jetstream, with 64MB on-chip memory for the queues and eight processing elements, each equipped with a 2KB scratchpad and a 1KB edge-cache. For power and area estimates for memory components, we use CACTI 7[7]. The queue memory is designed using 22nm ITRS-HP SRAM technology. We also model the communication network, the scheduler, and other logic components. A breakdown of power and area estimates are in Table 5. MEGA incorporates a majority of the architectural elements from Jetstream, such as the event queue, prefetcher, and cache. However, MEGA also includes additional version registers, a batch scheduler, and decoders within the event queue, which leads to some hardware overhead. The overall area and power are slightly higher than JetStream for the queues and network due to expanded event sizes with instance and batch ids. Consuming only 10 Watts, MEGA is substantially more power-efficient than our baseline GPU and CPU systems.

Table 5: Power and area of MEGA components

		Power(mW)			Area(mm^2)
		Static	Dynamic	Total	
Queue	64	123 (+5%)	23.5 (+13%)	9389 (~6%)	195 (+1.5%)
Scratchpad	8	0.35 (~0%)	1.3 (+8%)	13.2 (+9%)	0.25 (~4%)
Network		120 (+31%)	7.5 (+39%)	127.5 (+31%)	10.0 (+43%)
Proc. Logic		-	-	1.9 (+6%)	1.2 (+34%)
Total		-	-	9532 (+6.8%)	203(+2%)

6 RELATED WORK

Among the most recent works on rapid analysis of evolving graphs are RisGraph [15] and Tegra [20]. RisGraph targets at achieving real-time qurey by developing a new data structure for fast edge insertion and deletions. However, this is achieved at the trade-off of memory size of 3.25x to 3.38x. Tegra provides a novel API for performance ad-hoc queries on arbitrary time windows of the graph by using a compact in-memory representation for both graph and intermeidate computation state. Both RisGraph and Tegra leverage existing algorithms developed for streaming systems to support incremental computation for handling edge additions and deletions. Other storage systems to support evolving and streaming graphs include GraphOne and Aspen while systems that amortize the cost of memory accesses and computation include Chronos [18] and FA+PA [48]. However, these frameworks are limited in the types of graph updates they can handle. In particular, they do not support edge deletions. Another category of systems that exploit graph sharing are the systems that concurrently evaluate multiple (different) queries on a *single version* of a graph [11, 54, 57].

Single version streaming graph system has been proposed also, the algorithms maintain a single graph and a standing query's results that are incrementally added up when a new batch of updates are applied to the graph. The target of these works is on incremental computation, i.e. how to efficiently update query results. Early streaming systems (such as Kineograph [12], Naiad [36], Tornado [44] and Tripoline [24]) only support incremental computations for edge additions while more recent systems (such as Kickstarter [49] and GraphBolt [32]) also support edge deletions. Although many of the above dynamic graph system support both version control and incremental computation, none of them exploit parallelism and data reuse among different snapshots. MEGA is the first accelerator that supports parallel computation across different snapshots thus accelerating the execution time significantly.

A number of hardware accelerators target acceleration of queries on static graphs (e.g., [1, 13, 17, 21, 22, 39]). Several architectural approaches have been developed to enhance graph traversal performance, such as Coup[55], which minimizes read and write traffic, PHI[35], which decreases on-chip traffic, and HATS[34], a hardware-assisted scheduler that promotes locality. A few recent works explore dynamic graph processing. GraSU[51] provides the first FPGA-based graph update library for dynamic graphs. Jetstream[40] is the first streaming graph accelerator supporting incremental algorithms. TDGraph[56] augments many-core processors to support both graph mutation (changing the graph) and graph computation. Basak et. al. [8] provide an accelerator to sort streaming edges to improve locality and make their execution faster on a conventional graph accelerator. None of these works support evolving graph processing and it is not simple to extend them to track processing of multiple concurrent versions of the graph.

7 CONCLUDING REMARKS

In this paper we introduced MEGA, the first evolving graph accelerator. The evolving graph problem is compute- and memory-intensive as it evaluates a query on many snapshots of a graph. The snapshots may be quite similar in their graph structure since the changes to the graph tend to be small relative to the overall size. MEGA uses the CommonGraph approach to eliminate the need to handle expensive edge deletions. We develop a new scheduling and execution model, Batch-oriented execution, that applies update batches concurrently when possible, and with high graph reuse. Overall MEGA achieves 24×-120× speedup over CommonGraph. It also achieves 4.08×-5.98× speedup compared to JetStream, a recent streaming graph accelerator.

ACKNOWLEDGMENTS

We thank all the reviewers for their valuable feedback. This work is supported in part by National Science Foundation Grants CNS-1955650, CNS-2053383, CCF-2028714, CCF-2002554 and CCF-2226448 to the University of California, Riverside.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1247–1262. https://doi.org/10.1145/3373376.3378454

[2] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3575693.3575713

[3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data (Abstract). In *Proceedings of the 2023 ACM Workshop on Highlights of Parallel Computing* (Orlando, FL, USA) *(HOPC '23)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/3597635.3598022

[4] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Graph Analytics on Evolving Data (Abstract). In *arXiv preprint arXiv:2308.14834*. https://doi.org/10.48550/arXiv.2308.14834

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*, Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 722–735.

[6] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. ACM, New York, NY, USA, 44–54. https://doi.org/10.1145/1150402.1150412

[7] 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 Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[8] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving streaming graph processing performance using input knowledge. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1036–1050.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. 235–248.

[10] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[11] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: A Compiler and Runtime System for Concurrent Graph Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 51, 16 pages. https://doi.org/10.1145/3458817.3476159

[12] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

[13] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. *PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators*. IEEE Press, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

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

[15] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. https://doi.org/10.1145/3448016.3457263

[16] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.

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

[18] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2592798.2592799

[19] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. 2017. MultiGraph: Efficient Graph Processing on GPUs. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*. 27–40. https://doi.org/10.1109/PACT.2017.48

[20] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 337–355. https://www.usenix.org/conference/nsdi21/presentation/iyer

[21] Mark C Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-centric execution of speculative parallel programs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[22] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proceedings of the 48th international symposium on microarchitecture*. 228–241.

[23] Xiaolin Jiang, Mahbod Afarin, Zhijia Zhao, Nael Abu-Ghazaleh, and Rajiv Gupta. 2024. Core Graph: Exploiting Edge Centrality to Speedup the Evaluation of Iterative Graph Queries. In *2024 Proceedings of the Nineteen European Conference on Computer Systems (EuroSys'24)*.

[24] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 17–32. https://doi.org/10.1145/3447786.3456226

[25] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT '15)*. 39–50. https://doi.org/10.1109/PACT.2015.15

[26] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14)*. ACM, 239–252. https://doi.org/10.1145/2600212.2600227

[27] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[28] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *In Proceedings of International Conference on World Wide Web Companion, May 13–17, 2013, Rio de Janeiro, Brazil*. ACM, 1343–1350. https://doi.org/10.1145/3575693.3575713

[29] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[30] 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).

[31] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[32] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[33] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.

[34] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2018.00010

[35] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.

[36] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 456–471. https://doi.org/10.1145/2517349.2522739

[38] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

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

[40] 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* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1091–1105. https://doi.org/10.1145/3466752.3480126

[41] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE computer architecture letters* 10, 1 (2011), 16–19.

[42] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth EuroSys Conference (EuroSys '20)*. 12:1–12:16. https://doi.org/10.1145/3342195.3387537

[43] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[44] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/2882903.2882950

[45] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[46] SST-toolkit. 2023. *http://sst-simulator.org/*. http://sst-simulator.org/ Accessed: 2023-02-20.

[47] Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, Vol. 1. Present Day Trends of Innovations Lamza Poland.

[48] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic Analysis of Evolving Graphs. *ACM Trans. Archit. Code Optim.* 13, 4, Article 32 (oct 2016), 27 pages. https://doi.org/10.1145/2992784

[49] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[50] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. 223–236. https://doi.org/10.1145/3037697.3037747

[51] 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 *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 149–159.

[52] Yangzihao Wang, Yuechao Pan, Andrew A. Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (2017), 3:1–3:49. https://doi.org/10.1145/3108140

[53] M. Wu, C. Li, and Z. et al. Shen. 2022. Use of temporal contact graphs to understand the evolution of COVID-19 through contact tracing data. *Communication Physics* (2022). Available from https://doi.org/10.1038/s42005-022-01045-4.

[54] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2022. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3567955.3567963

[55] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. 13–25.

[56] Jin Zhao, Yun Yang, Yu Zhang, Xiaofei Liao, Lin Gu, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, Xinyu Jiang, et al. 2022. TDGraph: a topology-driven accelerator for high-performance streaming graph processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 116–129.

[57] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: An Efficient Storage System for High Throughput of Concurrent Graph Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3295500.3356143

[58] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX Annual Technical Conference (USENIX ATC), July 8-10, Santa Clara, CA, USA*. USENIX Association, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu