MEGA Evolving Graph Accelerator

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

Mahbod Afarin∗
mfar001@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).
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 eliminates the need for expensive edge deletion operations; and (2) it exploits 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 24-120× speedup over Software CommonGraph. It also achieves 4-6× 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, it is possible to leverage streaming, solving the query on the earliest snapshot and then use 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 $IG[1]$ in Figure 1(a) which is the common graph across snapshots $G_1$ and $G_{i+1}$. To move from $G_1$ to $G_{i+1}$ using conventional streaming algorithms we have to add edges $\Delta^+_i$ and delete edges $\Delta^+_i$. The common graph, $IG[1]$ has all the edges common to both $G_1$ and $G_{i+1}$ (for example, all edges $\Delta$ excluding the deleted edges $\Delta^+_i$, or alternatively all edges $G_{i+1}$ excluding the added edges $\Delta^+_i$). As a result, we can go from $IG[1]$ to either $G_1$ or $G_{i+1}$ by adding the missing set of edges ($\Delta^+_i$ or $\Delta^-_i$ 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
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 in 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 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 Δ𝑖 is processed only once in streaming (from 𝐺𝑖 to 𝐺𝑖+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 𝐺𝑖 to 𝐼𝐶𝐺 and from 𝐼𝐶𝐺 to 𝐺𝑖+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 1: CommonGraph: Triangular Grid representation and alternative processing workflows.](image1)

![Figure 2: High cost of deletions in JetStream.](image2)

![Figure 3: Number of additions in SSSP.](image3)
As opposed to the Bulk Synchronous Parallel model, it achieves limitations: (i) we are restricted to solving one snapshot at a time; (ii) deletions that are considerably more resource-intensive and 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 Jetstream, 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 $\Delta_{i+2}$. In Figure 7(b), this computation is performed three times, resulting in redundant work. Consider the use of the orange

- **Figure 4**: Edge reuse: different batches, same snapshot.
- **Figure 5**: Edge reuse: same batch, different snapshots.
- **Figure 6**: Unified Evolving Graph CSR Representation.
The computation for the snapshots are performed at the same time. That is, multiple users of an addition 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.

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
MEGA Evolving Graph Accelerator

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.

4.2 Execution and Datapath

MEGA supports multiple active snapshots: events are marked with a version tag to allow 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
When a computation batch begins, the scheduler marks its entry in the version table: a look-up-table containing information about the composition of different snapshots and their processing status. When a new batch is scheduled, the batch reader brings the batch 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 3). 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 the version table as active (Step 2 in Figure 12). The version table broadcasts updates to all processing elements (PE) and event-queue banks (Step 3 and Step 5), 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 D 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.

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.

Figure 13: Queue support for multiple snapshots.
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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>EdgeFunction (e(u, v))</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>CASMIN(Val(v), min(Val(u) + Val(v)))</td>
</tr>
<tr>
<td>SSWP</td>
<td>CASMAX(Val(v), min(Val(u), wt(u, v)))</td>
</tr>
<tr>
<td>SSNP</td>
<td>CASMIN(Val(v), max(Val(u), wt(u, v)))</td>
</tr>
<tr>
<td>SSSP</td>
<td>CASMIN(Va(Val(v), Val(u) + wt(u, v)))</td>
</tr>
<tr>
<td>Viterbi</td>
<td>CASMAX(Val(v), Val(u)/wt(u, v))</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Input Graph</th>
<th>Edges</th>
<th>Vertices</th>
<th>Batch Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pokec (PK)</td>
<td>30M</td>
<td>1.6M</td>
<td>0.3M</td>
</tr>
<tr>
<td>LiveJournal (LJ)</td>
<td>70M</td>
<td>4M</td>
<td>0.7M</td>
</tr>
<tr>
<td>Orkut (OR)</td>
<td>117M</td>
<td>3M</td>
<td>240K</td>
</tr>
<tr>
<td>DBpediaLinks (DL)</td>
<td>170M</td>
<td>18M</td>
<td>1.7M</td>
</tr>
<tr>
<td>UK-2002 (UK)</td>
<td>260M</td>
<td>18M</td>
<td>2.6M</td>
</tr>
<tr>
<td>WikipediaLinks (Wen)</td>
<td>400M</td>
<td>13M</td>
<td>4M</td>
</tr>
</tbody>
</table>

Table 3: Experimental Configurations.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>MEGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Unit</td>
<td>60×Intel(R)Xeon(R)@3.1GHz</td>
<td>8×MEGA Processor @ 1GHz</td>
</tr>
<tr>
<td>On-chip memory</td>
<td>49.5MB L3</td>
<td>64MB eDRAM @22nm</td>
</tr>
<tr>
<td>Off-chip Bandwidth</td>
<td>17GB/s Channel</td>
<td>4x DDR4</td>
</tr>
</tbody>
</table>

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, 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, Kickstarter, 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 68GB/s, which is less than a third of the bandwidth available on the K80 GPUs (240 GB/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.
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. 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,
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>
<thead>
<tr>
<th>Table 5: Power and area of MEGA components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power(mW)</td>
</tr>
<tr>
<td>Queue</td>
</tr>
<tr>
<td>Scratchpad</td>
</tr>
<tr>
<td>Network</td>
</tr>
<tr>
<td>Proc. Logic</td>
</tr>
<tr>
<td>Total</td>
</tr>
</tbody>
</table>

### 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 query 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 intermediate 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], Naiada [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 [54], 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 to handle expensive edge deletions. We implement a new scheduler 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.

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-2026448 to the University of California, Riverside.

REFERENCES
