Accelerating Dynamic Graph Analytics on GPUs

Technical Report

Mo Sha, Yuchen Li, Bingsheng He and Kian-Lee Tan

April 1, 2017

Contents

Abstract

As graph analytics often involves compute-intensive operations, extensive efforts have been made for using GPUs to accelerate analytics on graphs. However, in many applications such as social networks, cyber security, and fraud detection, their representative graphs evolve frequently and one has to perform a rebuild of the graph structure on GPUs against each single update. Hence, rebuilding the graphs becomes the bottleneck of processing high-speed graph streams. In this paper, we propose a GPU-based dynamic graph storage framework to support existing graph algorithms with ease. Furthermore, we propose parallel update algorithms to support efficient stream updates so that the maintained graph is immediately available for blazing fast analytic processing on GPUs. Our extensive experiments with three streaming applications on large-scale real and synthetic datasets demonstrate the superior performance of our proposed approach.

1 Introduction

Due to the rising complexity of data generated in the big data era, graph representations are used ubiquitously. Massive graph processing have emerged as the de facto standard of analytics on web graphs, social networks (e.g., Facebook and Twitter), sensor networks (e.g., Internet of Things) and many other application domains which involve high-dimensional data (e.g., recommendation systems). These graphs are often highly dynamic: network traffic data averages 10^9 packets/hour/router for large ISPs [23]; Twitter has 500 million tweets per day [40]. Since real-time analytics is fast becoming the norm [26, 12, 35, 42], it is thus critical for operations on dynamic massive graphs to be processed efficiently.

Dynamic graph analytics has a wide range of applications. Twitter can recommend information based on the up-to-date TunkRank (similar to PageRank) computed based on a dynamic attention graph [14] and cellular network operators can fix traffic hotspots in their networks as they are detected [27]. To achieve real-time performance, there is a growing interest to offload the graph analytics to GPUs due to its much stronger arithmetical power and higher memory bandwidth compared with CPUs. Although existing solutions, e.g. Medusa [57] and Gunrock [48], have explored GPU graph processing, we are aware of only one work [29] that has considered a dynamic graph scenario which is a major gap for running analytics on GPUs. In fact, a delay in updating a dynamic graph may lead to undesirable consequences. For instance, consider an online travel insurance system that detects potential frauds by running ring analysis on profile graphs built from active insurance contracts [5]. Analytics on an outdated profile graph may fail to detect frauds which can cost millions of dollars. However, updating the graph will be too slow for issuing contracts and processing claims in real time, which will severely influence legitimate customers' user experience. This motivates us to develop a update-efficient graph structure on GPUs to support dynamic graph analytics.

There are two major concerns when designing a GPU-based dynamic graph storage framework. First, the proposed storage framework should handle both insertion and deletion operations efficiently. Though processing updates against insertion-only graph stream could be handled by reserving extra spaces to accommodate the updates, this naïve approach fails to preserve the locality of the graph entries and cannot support deletions efficiently. Considering a common sliding window model on a graph edge stream, each element in the stream is an edge in a graph and analytic tasks are performed on the graph induced by all

edges in the up-to-date window [49, 15, 17]. A naïve approach needs to access the entire graph in the sliding window to process deletions. This is obviously undesirable against high-speed streams. Second, the proposed storage framework should be general enough for supporting existing graph formats on GPUs so that we can easily reuse existing static GPU graph processing solutions to take over the analytic workloads. Large graphs are inherently *sparse*. To maximize the efficiency, existing works [6, 32, 31, 29, 51] on GPU sparse graph processing rely on optimized data formats and arrange the graph entries in certain sorted order, e.g. CSR [32, 6] sorts the entries by their row-column ids. However, to the best of our knowledge, no schemes on GPUs can support efficient updates and maintain a sorted graph format at the same time, other than a rebuild. This motivates us to design an update-efficient sparse graph storage scheme on GPUs while keeping the locality of the graph entries for processing massive analytics instantly.

In this paper, we introduce a GPU dynamic graph analytic framework followed by proposing the dynamic graph storage scheme on GPUs. Our preliminary study shows that a cache-oblivious data structure, i.e., Packed Memory Array (`PMA` [10, 11]), can potentially be employed for maintaining dynamic graphs on GPUs. `PMA`, originally designed for CPUs [10, 11], maintains sorted elements in a partially contiguous fashion by leaving gaps to accommodate fast updates with a constant bounded gap ratio. The simultaneously sorted and contiguous characteristic of `PMA` nicely fits the scenario of GPU streaming graph maintenance. However, the performance of `PMA` degrades when updates occur in locations which are close to each other, due to the unbalanced utilization of reserved spaces. Furthermore, as streaming updates often come in batches rather than one single update at a time, `PMA` does not support parallel insertions and it is non-trivial to apply `PMA` to GPUs due to its intricate update patterns which may cause serious thread divergence and uncoalesced memory access issues for GPUs.

We thus propose two GPU-oriented algorithms, i.e. `GPMA` and `GPMA+`, to support efficient parallel batch updates. `GPMA` explores a lock-based approach which becomes increasingly popular due to the recent GPU architectural evolution for supporting atomic operations [18, 28]. While `GPMA` works efficiently for the case where few concurrent updates conflict, e.g., small-size update batches with random updating edges in each batch, there are scenarios where massive conflicts occur and hence, we propose a lock-free approach, i.e. `GPMA+`. Intuitively, `GPMA+` is a bottom-up approach by prioritizing updates that occur in similar positions. The update optimizations of our proposed `GPMA+` are able to maximize coalesced memory access and achieve linear performance scaling w.r.t the number of computation units on GPUs, regardless of the update patterns.

In summary, the key contributions of this paper are the following:

- We introduce a framework for GPU dynamic graph analytics and propose, the first of its kind, a GPU dynamic graph storage framework to pave the way for real-time dynamic graph analytics on GPUs.
- We devise two GPU-oriented parallel algorithms: `GPMA` and `GPMA+`, to support efficient updates against high-speed graph streams.
- We conduct extensive experiments to show the performance superiority of `GPMA` and `GPMA+`. In particular, we design different update patterns on real and synthetic graph streams to validate the update efficiency of our proposed algorithms against their CPU counterparts as well as the GPU rebuild baseline. In addition, we implement three real world graph analytic applications on the graph streams to demonstrate the efficiency

and broad applicability of our proposed solutions.

The remainder of this paper is organized as follows. The related work is discussed in Section 2. Section 3 presents a general workflow of dynamic graph processing on GPUs. Subsequently, we describe GPMA and GPMA+ in Sections 4-5 respectively. Section 6 reports results of a comprehensive experimental evaluation. We conclude the paper and discuss some future works in Section 7.

2 Related Work

In this section, we review related works in three different categories as follows.

2.1 Graph Stream Processing

Over the last decade, there has been immense interest in designing efficient algorithms for processing massive graphs in the data stream model (See [35] for a detailed survey). This includes the problems of PageRank-styled scores [38], connectivity [21], spanners [20], counting subgraphs e.g. triangles [46] and summarization [44]. However, these works mainly focus on the theoretical study to achieve the best approximation solution with linear bounded space. Our proposed methods can incorporate existing graph stream algorithms with ease as our storage framework can support most graph representations used in existing algorithms.

Many systems have been proposed for streaming data processing, e.g. Storm [45], Spark Streaming [54], Flink [1]. Attracted by its massively parallel performance, several attempts have successfully demonstrated the advantages of using GPUs to accelerate data stream processing [47, 56]. However, the aforementioned systems focus on general stream processing and lack support for graph stream processing. Stinger [19] is a parallel solution to support dynamic graph analytics on a single machine. More recently, Kineograph [14], CellIQ [27] and GraphTau [26] are proposed to address the need for general time-evolving graph processing under the distributed settings. However, we are aware of only one work [29] which explores the direction of using GPUs to process real-time analytics on dynamic graphs.

2.2 Graph Analytics on GPUs

Graph analytic processing is inherently data- and compute-intensive. Massively parallel GPU accelerators are powerful means to achieve supreme performance of many applications. Compared with CPU, which are general-purpose processors featuring large cache size and high single core processing capability, GPU devotes most of its die area to a large number of simple Arithmetic Logic Units (ALUs), and executes code in a SIMT (Single Instruction Multiple Threads) fashion. With the massive amount of ALUs, GPU offers orders of magnitude higher computational throughput than CPU for applications with ample parallelism. This leads to a spectrum of works which explore the usage of GPUs to accelerate graph analytics and demonstrate immense potentials. Examples include breath-first search (BFS) [32], subgraph query [31], PageRank [6] and many others. The success of deploying specific graph algorithms on GPU motivates the design of general

GPU graph processing systems like Medusa [57] and Gunrock [48]. However, the aforementioned GPU-oriented graph algorithms and systems assume static graphs. To handle dynamic graph scenario, existing works have to perform a rebuild on GPUs against each single update. DCSR [29] is the only solution, to the best of our knowledge, which is designed for insertion-only scenarios as it is based on linked edge block and rear appending technique. However, it does not support deletions or efficient searches. We propose `GPMA` to enable efficient dynamic graph updates (i.e. insertions and deletions) on GPUs in a fine-grained manner. In addition, existing GPU-optimized graph analytics and systems can replace their storage layers directly with ease since the fundamental graph storage schemes used in existing works can be directly implemented on top of our proposed storage scheme.

2.3 Storage Formats on GPUs

Sparse matrix representation is a popular choice for storing large graphs on GPUs [3, 2, 57, 48] The Coordinate Format [16] (COO) is the simplest format which only stores non-zero matrix entries by their coordinates with values. COO sorts all the non-zero entries by the entries' row-column key for fast entry accesses. CSR [32, 6] compresses COO's row indices into an offset array to reduce the memory bandwidth when accessing the sparse matrix. To optimize matrices with different non-zero distribution patterns, many customized storage formats were proposed, e.g., Block COO [50] (BCCOO), Blocked Row-Column [7] (BRC) and Tiled COO [52] (TCOO). Existing formats require to maintain a certain sorted order of their storage base units according to the unit's position in the matrix, e.g. entries for COO and blocks for BCCOO, and still ensure the locality of the units. As mentioned previously, few prior schemes can handle efficient sparse matrix updates on GPUs. To the best of our knowledge, `PMA` [10, 11] is a common structure which maintains a sorted array in a contiguous manner and supports efficient insertions/deletions. However, `PMA` is designed for CPU and no concurrent updating algorithm is ever proposed. Thus, we are motivated to propose `GPMA` and `GPMA+` for supporting efficient concurrent updates on all existing storage formats.

3 A dynamic framework on GPUs

To address the need for real-time dynamic graph analytics, we offload the tasks of concurrent dynamic graph maintenance and its corresponding analytic processing to the GPU. In this section, we introduce a general GPU dynamic graph analytic processing framework. The design of the framework takes two major concerns: the framework should not only handle graph updates efficiently but also support existing GPU-oriented graph analytic algorithms without forfeiting their performance.

Model: We adopt a common sliding window graph stream model [35, 27, 44]. The sliding window model consists of an unbounded sequence of elements $(u, v)_t$ [1] which indicates the edge (u, v) arrives at time t, and a sliding window which keeps track of the most recent edges. As the sliding window moves with time, new edges in the stream keep being inserted into the window and expiring edges are deleted. In real world applications, the sliding window of a graph stream can be used to monitor and analyze fresh social actions appeared on Twitter [49] or the call graph formed by the most recent CDR data [27]. In

[1] Our framework handles both directed and undirected edges.

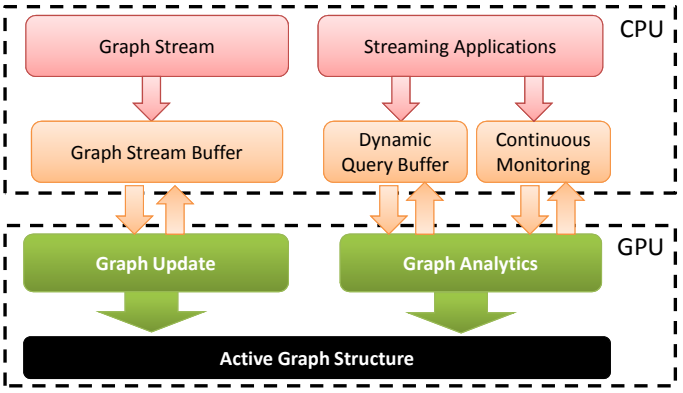


Figure 1: The dynamic graph processing framework

this paper, we focus on presenting how to handle edge streams but our proposed scheme can also handle the dynamic *hyper graph* scenario with hyper edge streams. Moreover, insertions/deletions of vertices are supported as well. To simplify the presentation, we leave the discussion of handling those scenarios in Section 5.3.

Apart from the sliding window model, the graph stream model which involves explicit insertion and deletion operations (e.g., a user requests to add or delete a friend in the social network) is also supported by our scheme as the proposed dynamic graph storage structure is designed to handle random update operations. That is, our system supports two kinds of updates, *implicit* ones generated from the sliding window mechanism and *explicit* ones generated from upper level applications or users.

The overview of the dynamic graph processing framework is presented in Figure 1. Given a graph stream, there are two types of streaming tasks supported by our framework. The first type is the ad-hoc queries such as neighborhood and reachability queries on the graph which is constantly changing. The other type is the monitoring tasks like tracking PageRank scores. We present the framework by illustrating how to handle the graph streams and corresponding queries while hiding data transfer between CPU and GPU, as follows:

Graph Streams: The graph stream buffer module batches the incoming graph streams on the CPU side (host) and periodically sends the updating batches to the graph update module located on GPU (device). The graph update module updates the "active" graph stored on the device by using the batch received. The "active" graph is stored in the format of our proposed GPU dynamic graph storage structure. The details of the graph storage structure and how to update the graph efficiently on GPUs will be discussed extensively in later sections.

Queries: Like the graph stream buffer, the dynamic query buffer module batches ad-hoc queries submitted against the stored active graph, e.g., queries to check the dynamic reachability between pairs of vertices. The tracking tasks will also be registered in the continuous monitoring module, e.g., tracking up-to-date PageRank. All ad-hoc queries and monitoring tasks will be transferred to the graph analytic module for GPU accelerated processing. The analytic module interacts with the active graph to process the queries and the tracking tasks. Subsequently, the query results will be transferred back to the host. As most existing GPU graph algorithms use optimized array formats like CSR to accelerate the performance [18, 28, 34, 52], our proposed storage scheme provides an interface for

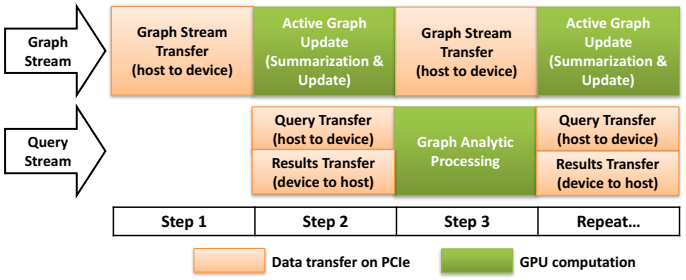


Figure 2: Asynchronous streams

storing the array formats. In this way, existing algorithms can be integrated into the analytic module with ease. We describe the details of the integration in Section 5.4

Hiding Costly PCIe Transfer: Another critical issue on designing GPU-oriented systems is to minimize the data transfer between the host and the device through PCIe. Our proposed batching approach allows overlapping data transfer by concurrently running analytic tasks on the device. Figure 2 shows a simplified schedule with two asynchronous streams: graph streams and query streams respectively. The system is initialized at Step 1 where the batch containing incoming graph stream elements is sent to the device. At Step 2, while PCIe handles bidirectional data transfer for previous query results (device to host) and freshly submitted query batch (host to device), the graph update module updates the active graph stored on the device. At Step 3, the analytic module processes the received query batch on the device and a new graph stream batch is concurrently transferred from the host to the device. It is clear to see that, by repeating the aforementioned process, all data transfers are overlapped with concurrent device computations.

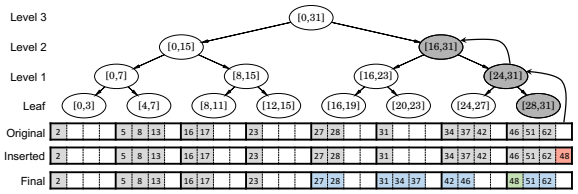
4 GPMA Update

The core of the dynamic graph processing framework is the active graph storage module (see Figure 1). In this section, we present how to support efficient graph updates while still preserving the locality of graph entries for high-speed analytic processing. We first discuss the requirements of graph updates on GPUs in Section 4.1. Subsequently, we introduce GPMA and present how GPMA handles update operations in Section 4.2.

4.1 Graph Update Operations

To enable efficient large scale (a.k.a sparse) graph processing, existing works store graph entries in certain sorted order to achieve high locality. In this paper, we use the most widely used CSR [32, 6] as an example to illustrate our storage scheme, but our scheme is certainly not limited to CSR and can be easily applied to other storage formats, such as COO [16], JAD [39] and HYB [9, 34]. CSR uses three arrays to store the edges: the row offset pointer array (starting vertices), the column index array (ending vertices) and the value array (edge values/attributes). The entries stored in each array must be sorted by their corresponding row-column key to ensure locality.

For dynamic graph maintenance, the designed scheme should ensure the sorted order of graph entries in the CSR format. Meanwhile, there are three types of elementary update



	Leaf	Level 1	Level 2	Level 3
segment size	4	8	16	32
density lower bound ρ	0.08	0.19	0.29	0.40
density upper bound τ	0.92	0.88	0.84	0.80
min # of entries	1	2	4	8
max # of entries	3	6	12	24

Figure 3: PMA insertion example (Left: PMA for insertion; Right: predefined thresholds)

operations which need to be considered:

- insertion of an edge;
- deletion of an edge;
- value modification of an edge.

While value modification only requires locating the physical location of the corresponding edge, edge insertions and deletions substantially change the landscape of device memory usage. Thus, our discussion mainly focuses on insertions and deletions. A straightforward update solution is to run a rebuild, i.e. re-sort the graph entries with the updates. However, such a naïve strategy is often prohibitively expensive for many real-time applications, especially in the case with high update rates. In the remainder of this section, we introduce the dynamic graph structure and the corresponding update strategies to minimize the overhead of dynamic graph updates.

4.2 Demystify GPMA

Background of PMA. GPMA is primarily motivated by a novel structure, Packed Memory Array (PMA [10, 11]), which is proposed to maintain sorted elements in a partially continuous fashion by leaving gaps to accommodate fast updates with a bounded gap ratio. PMA is a self-balancing binary tree structure. Given an array of N entries, PMA separates the whole memory space into *leaf segments* with $O(\log N)$ length and defines *non-leaf segments* as the space occupied by their descendant segments. For any segment located at height i (leaf height is 0), PMA designs a way to assign the lower and upper bound density thresholds for the segment as ρ_i and τ_i respectively to achieve $O(\log^2 N)$ amortized update complexity. Once an insertion/deletion causes the density of a segment to fall out of the range defined by (ρ_i, τ_i), PMA tries to adjust the density by re-allocating all elements stored in the segment's parent. The adjustment process is invoked recursively and will only be terminated if all segments' densities fall back into the range defined by PMA's density thresholds.

Example 1. *Figures 3 presents an example for PMA insertion. Each segment is uniquely identified by an interval (starting and ending position of the array) displayed in the corresponding tree node, e.g., the root segment is* ***segment-[0,31]*** *as it covers all 32 spaces. All values stored in PMA are displayed in the array. The table in the figure shows predefined parameters including the segment size, the assignment of density thresholds (ρ_i, τ_i) and the corresponding maximum and minimum entry sizes at different heights of the tree. We use these setups as a running example throughout the paper. To insert an entry. i.e. 48, into PMA, the corresponding leaf segment is firstly identified by a binary search, and the new entry is placed at the rear of leaf segment. The insertion causes the density of the leaf segment* 4 *to exceed the threshold* 3*. Thus, we need to identify the nearest ancestor segment which can accommodate the insertion without violating the thresholds, i.e., the*

Algorithm 1 GPMA Concurrent Insertion

```
1: procedure GPMAInsert(Insertions I)
2:     while I is not empty do
3:         parallel for i in I
4:             Seg s ← BinarySearchLeafSegment(i)
5:             TryInsert(s, i, I)
6:         synchronize
7:         release locks on all segments

8: function TryInsert(Seg s, Insertion i, Insertions I)
9:     while s ≠ root do
10:        synchronize
11:        if fails to lock s then
12:            return                                  ▷ insertion aborts
13:        if (|s| + 1)/capacity(s) < τ then
14:            s ← parent segment of s
15:        else
16:            Merge(s, i)
17:            re-dispatch entries in s evenly
18:            remove i from I
19:            return                                  ▷ insertion succeeds
20:    double the space of the root segment
```

***segment-[16,31]**. Finally, the update is completed by re-distpatching all entries evenly in **segment-[16,31]**.*

It is evident that PMA could be employed for dynamic graph maintenance as it maintains sorted elements efficiently with high locality on CPU. However, although the amortized update complexity of PMA is proved to be $O(\log^2 N)$ in the worst case and $O(\log N)$ in the average case, the update procedure described in [11] is inherently sequential and no concurrent algorithms have been proposed. To support batch updates of edge insertions/deletions for existing graph stream analytic processing, we devise GPMA to support concurrent efficient PMA updates on GPUs.

Concurrent Updates in GPMA. Motivated by PMA in CPU, we propose GPMA to handle a batch of updates concurrently on GPUs. Intuitively, GPMA assigns an update to a thread and concurrently executes PMA algorithm for each thread with a lock-based approach to ensure consistency. More specifically, all leaf segments of insertions are identified in advance, and then each thread checks whether the inserted segments still satisfy their thresholds from bottom to top. For each particular segment, it is accessed in a mutually exclusive fashion. Moreover, all threads are synchronized after updating all segments located at the same tree height to avoid possible conflicts as segments at a lower height are fully contained in the segments at a higher level.

Algorithm 1 presents the pseudocode for GPMA concurrent insertions. As shown in line 2, all entries in the insertion set are iteratively tried until all of them take effect. For each iteration shown in line 9, all threads start at leaf segments and attempt the insertions in a bottom-up fashion. If a particular thread fails the mutex competition in line 11, it aborts

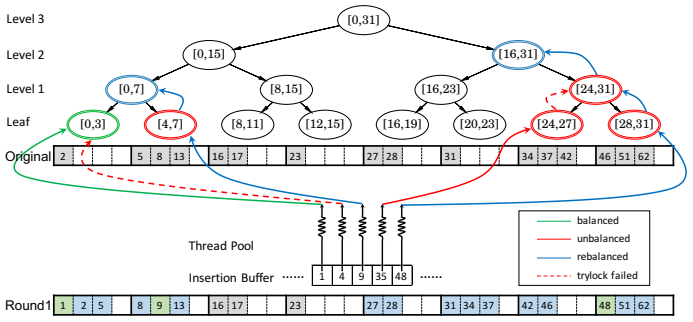


Figure 4: GPMA concurrent insertion

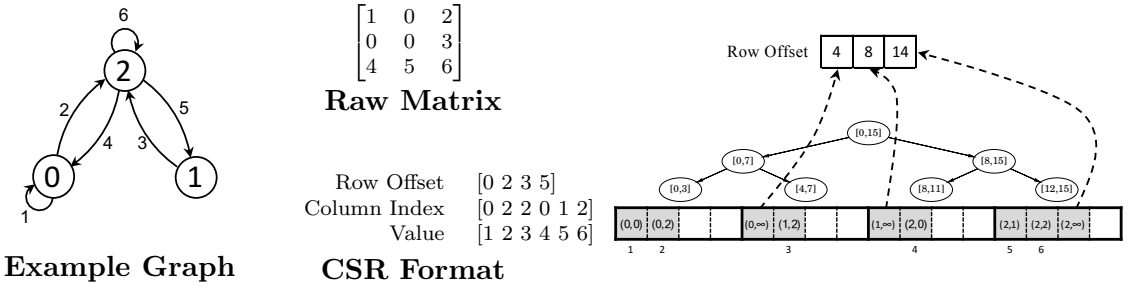


Figure 5: CSR on GPMA

immediately and waits for the next attempt. Otherwise, it inspects the density of the current segment. If the current segment does not satisfy the density requirement, it will try the parent segment in the next loop iteration (lines 13-14). Once an ancestor segment is able to accommodate the insertion, it merges the new entry in line 16 and the entry is removed from the insertion set. Subsequently, the updated segment will re-dispatch all its entries evenly and the process is terminated.

Example 2. *Figure 4 illustrates an example with five insertions, i.e.* $\{1, 4, 9, 35, 48\}$*, for concurrent GPMA insertion. The initial structure is the same as the one in Example 1. After identifying the leaf segment for insertion, threads responsible for* ***Insertion-1*** *and* ***Insertion-4*** *compete for the same leaf segment. Assuming* ***Insertion-1*** *succeeds in getting the mutex,* ***Insertion-4*** *is aborted. Due to enough free space of the segment,* ***Insertion-1*** *is successfully inserted. Even though there is no leaf segment competition for* ***Insertions-9,35,48****, they should continue to inspect the corresponding parent segments because all the leaf segments do not satisfy the density requirement after the insertions.* ***Insertions-35,48*** *still compete for the same level-1 segment and* ***Insertion-48*** *wins. For this example, three of the insertions are successful and the results are shown in the bottom of Figure 4.* ***Insertions-4,35*** *are aborted in this iteration and will wait for the next attempt.*

CSR as a case study. From the physical aspect, GPMA maintains its elements contiguous and ordered. This means that GPMA supports most logical formats which describe graphs in existing works on static graph computations. As a case study, we present how to implement CSR on GPMA, which is demonstrated in the following example.

Example 3. *In Figure 5, we have a graph of three vertices and six edges. The number on each edge denotes the weight of the corresponding edge. The graph is represented as a sparse matrix and is further transformed to the CSR format shown in the middle. The right part denotes the GPMA representation of this graph. In order to maintain the row offset*

array without synchronization among threads, we add a guard entry whose column index is ∞ during concurrent updates. That is to say, when the guard is moved, the corresponding element in row offset array will change.

Logical Deletions in GPMA. There are many stream application scenarios where edges are frequently expired when the stream advances, e.g., applications based on the sliding window stream model. A naïve way for handling deletions is to run Algorithm 1 once the deletion requests are invoked. However, such an explicit approach incurs considerable costs for deletions. We believe that, under such scenarios, it is unnecessary to reclaim deleted spaces before insertions attempt to occupy particular segments. Hence, the logical deletion strategy, i.e. tombstone technique, is suitable. Once a set of deletion requests are invoked, we simply locate the edges on GPMA and set them invalid without deleting them physically. In the case of graph storing in the CSR format, we set the value of the edge as zero when the edge requests to be deleted. GPMA will only recover the space occupied by the invalid edges when new edges are inserted. We would also like to highlight that the logical deletions on GPMA unify the update operations (insertion/deletion) to insertion operations only. Thus, in the remaining part of this paper, we only focus on GPMA insertions.

5 GPMA+ Update

Although GPMA can support concurrent graph updates on GPUs, the update algorithm is basically a lock-based approach and can suffer from serious performance issue when different threads compete for the same lock. In this section, we propose a lock-free approach, i.e. GPMA+, which makes full utilization of thousands of GPU multiprocessors. We carefully examine the performance bottleneck of GPMA in Section 5.1. Based on the issues identified, we propose GPMA+ for optimizing concurrent GPU updates with a lock-free approach in Section 5.2.

5.1 Bottleneck Analysis

The following four critical performance issues are identified for GPMA:

- **Uncoalesced Memory Accesses**: Since GPMA is a full binary tree, each thread has to traverse the tree from the root to identify the corresponding leaf segment to be updated. For a group of threads which share the same memory controller (including access pipelines and caches), memory accesses are uncoalesced and thus, cause additional IO overheads.
- **Atomic Operations for Acquiring Lock**: To produce consistent update results, each thread in GPMA needs to acquire the lock before it can perform the insertion. As the design of GPU architecture is not optimized for atomic operations, frequently invoking atomic operations for acquiring locks will bring huge overheads, even if there are no conflicts.
- **Possible Thread Conflicts**: As shown in Figure 4, if two threads conflict on a segment, one of them has to abort and waits for the next attempt. In the case where the insertions occur on segments which are located proximately, GPMA will end up with low parallelism. As most real world large graphs have the power law property, the effect of thread conflicts can be exacerbated.

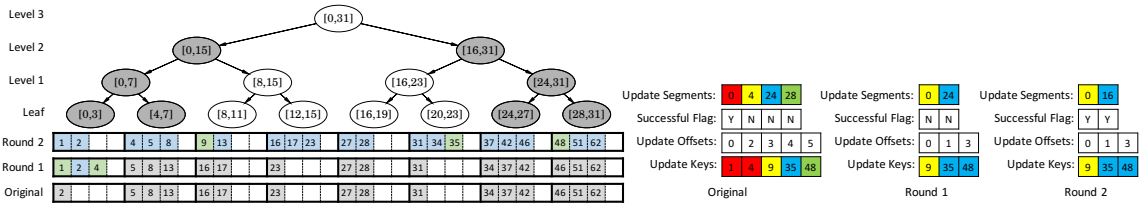


Figure 6: GPMA+ concurrent insertion

- **Unpredictable Thread Workload**: Workload balancing is another major concern for optimizing concurrent algorithms [43]. The workload for each thread in GPMA is unpredictable because: (1) It is impossible to obtain the last non-leaf segment traversed by each thread in advance; (2) The result of mutex competition is random. The unpredictable nature triggers the imbalanced workload issue for GPMA. In addition, threads are grouped as warps in the GPU. If a thread has a heavy workload, the remaining threads of the same warp are idle and cannot be re-scheduled.

5.2 GPMA+ Segment-Oriented Updates

Based on the discussion above, we propose GPMA+ to lift all bottlenecks identified. The proposed GPMA+ does not rely on lock mechanism and achieves high thread utilization simultaneously.

Compared with GPMA, which handles each insertion separately, GPMA+ concurrently processes updating operations based on the segments involved. It breaks the complex update pattern into existing concurrent GPU primitives to achieve maximum parallelism. There are three major components in the GPMA+ update algorithm:

(1) The updating operations are first sorted by their keys and then dispatched to GPU threads for locating their corresponding leaf segments according to the sorted order.
(2) The update operations belonging to the same leaf segment are grouped for processing and GPMA+ processes the updates level by level in a bottom-up manner.
(3) In any particular level, we leverage GPU primitives to invoke all computing resources for segment updates.

We note that, the issue of *uncoalesced memory access* in GPMA is resolved by component (1) as the updating threads are sorted in advance to achieve similar traversal paths. Component (2) completely avoids the use of locks, which solves the problem of *atomic operations* and *thread conflicts*. Finally, component (3) makes use of GPU primitives to achieve *workload balance* among all GPU threads.

We present the pseudocode for GPMA+'s segment-oriented updates in the procedure GpmaPlusUpdate of Algorithm 2. The updating entries are first sorted by their keys in line 2 and the corresponding segments are then identified in line 3. Given the update set U, GPMA+ processes updating segments level by level in lines 4-15 until all updates are executed successfully (line 11). In each iteration, UniqueInsertion in line 7 groups update entries belonging to the same segments into unique segments, i.e., S^*, and produces the corresponding index set I for quick accesses of updates entries located in a segment from S^*. As shown in lines 19-20, UniqueSegments only utilizes standard GPU primitives, i.e. RunLenghtEncoding and ExclusiveScan. RunLenghtEncoding compresses an input array by merging runs of an element into a single element. It also outputs a

Algorithm 2 GPMA+ Segment-Oriented Updates

```
1: procedure GpmaPlusUpdate(Updates U)
2:     Sort(U)
3:     Segs S ← BinarySearchLeafSegments(U)
4:     while root segment is not reached do
5:         Indices I ← ∅
6:         Segs S* ← ∅
7:         (S*, I) ← UniqueSegments(S)
8:         parallel for s ∈ S*
9:             TryInsert+(s, I, U)
10:        if U = ∅ then
11:            return
12:        parallel for s ∈ S
13:            if s does not contain any update then
14:                remove s from S
15:            s ← parent segment of s
16:    r ← double the space of the old root segment
17:    TryInsert+(r, ∅, U)

18: function UniqueSegments(Segs S)
19:    (S*, Counts) ← RunLengthEncoding(S)
20:    Indices I ← ExclusiveScan(Counts)
21:    return (S*, I)

22: function TryInsert+(Seg s, Indices I, Updates U)
23:    n_s ← CountSegment(s)
24:    U_s ← CountUpdatesInSegment(s,I,U)
25:    if (n_s + |U_s|)/capacity(s) < τ then
26:        Merge(s, U_s)
27:        re-dispatch entries in s evenly
28:        remove U_s from U
```

count array denoting the length of each run. ExclusiveScan calculates, for each entry e in an array, the sum of all entries before e. Both primitives have very efficient parallelized GPU-based implementation which makes full utilization of the massive GPU cores. In our implementation, we use the NVIDIA CUB library [4] for these primitives. Given a set of unique updating segments, TryInsert+ first checks if a segment s has enough space for accommodating the updates by summing the valid entries in s (CountSegment) and the number of updates in s (CountUpdatesInSegment). If the density threshold is satisfied, the updates will be materialized by merging the entries to be updated with existing entries in the segment. Subsequently, all entries in the segment will be re-dispatched to balance the densities. After TryInsert+, the algorithm will terminate if there are no entries to be updated. Otherwise, GPMA+ will advance to higher levels by setting all remaining segments to their parent segments (lines 12-15). The following example illustrates GPMA+'s segment-oriented updates.

Example 4. *Figure 6 illustrates an example for GPMA+ updates with the same setup as*

example 2. The left part is GPMA+'s snapshots in different rounds during this batch of insertions. The right part denotes the corresponding array information after the execution of each round. Five insertions are grouped into four corresponding leaf segments. In Round 1, ***Insertions-1,4*** *are inserted into* ***Segment-[0,4]*** *because the density threshold is not violated after the insertion. The remaining three insertions will be dispatched to their parent segments. It should be noted that the blue and the green grids belong to the same parent segment and therefore, will be merged and then dispatched to their shared parent segment. In Round 2, all remaining insertions successfully take effect.*

In Algorithm 2, TryInsert+ is the most important function as it handles all the corresponding insertions with no conflicts. Moreover, it achieves a balanced workload for each concurrent task. This is because GPMA+ handles the updates level by level and each segment to be updated in a particular level has exactly the same capacity. However, segments in different levels have different capacities. Intuitively, the probability of updating a segment with a larger size (segments closer to the root) is much lower than that of a segment with a smaller size (segments closer to the leaf). To optimize towards the GPU architecture, we propose the following optimization strategies for TryInsert+ for segments with different sizes. Those optimizations take advantage of the massive thread parallelism of the GPU in different levels.

- **Warp-Based**: For any segment with entries not larger than the warp size, TryInsert+ for the segment will be handled by a warp. Due to the fact that all threads in the same warp are tied together and warp-based data is held by registers, updating a segment by a warp does not require explicit synchronization and will obtain superior efficiency. Warp-based approach is extremely efficient which can approximate the theoretically computational ability of GPUs and can handle the vast majority of insertions.
- **Block-Based**: For segments whose data can be loaded in GPU's shared memory, block-based approach is chosen. Block-based approach executes all updates in the shared memory. As shared memory has much larger size than warp registers, block-based approach can handle large segments efficiently.
- **Device-Based**: For segments which have size larger than the size of the shared memory, we handle them via global memory accesses and rely on device-wide synchronization. Device-based approach is slower than the two approaches above, but it has no memory size limitations and is not invoked frequently.

We refer interested readers to Appendix A for the detailed algorithm of the optimizations above.

Theorem 1. *Given there are K computation units in the GPU, the amortized updating performance of GPMA+ is $O(1 + \frac{log^2 N}{K})$, where N is the maximum number of edges in the dynamic graph.*

Proof. Let X denote the set of updating entries contained in a batch. We consider the case where $|X| \geq K$ as it is rare to see $|X| < K$ in real world scenarios. In fact, our analysis works for cases where $|X| = O(K)$. The total update complexity consists of three parts: (1) sorting the updating entries; (2) searching the position of the entries in GPMA; (3) inserting the entries. We study these three parts separately.

For part (1), the sorting complexity of $|X|$ entries on the GPU is $O(\frac{|X|}{K})$ since parallel radix sort is used (keys in GPMA are integers for storing edges). Then, the amortized sorting complexity is $O(\frac{|X|}{K})/|X| = O(1)$.

Algorithm 3 Standard GPU-based Neighbor Gathering

```
1: procedure GATHER(Vertice frontiers, Int csr_offset)
2:     v_id ← frontiers[thread_id]
3:     {r, r_end} ← csr_offset[v_id, v_id + 1]
4:
5:     // vie for control of device
6:     if r < r_end and IsEntryExist(r) then
7:         winner ← thread_id
8:
9:     if winner == thread_id then
10:        // winner broadcast its adjlist to other threads
11:        // each thread pick one edge from adjlist
12:        // and do corresponding tasks in parallel
```

For part (2), the complexity of concurrently searching $|X|$ entries on GPMA is $O(\frac{|X| \cdot logN}{K})$ since each entry is assigned to one thread and the depth of traversal is the same for one thread (GPMA is a balanced tree). Thus, the amortized searching complexity is $O(\frac{|X| \cdot logN}{K})/|X| = O(\frac{logN}{K})$.

For part (3), we need to conduct a slightly complicated analysis. We denote the total insertion complexity of X with GPMA+ as $c^X_{\text{GPMA+}}$. As GPMA+ is updated level by level, $c^X_{\text{GPMA+}}$ can be decomposed into: $c^X_{\text{GPMA+}} = c_0 + c_1 + ... + c_h$ where h is the height of the PMA tree. Given any level i, let z_i denote the number of segments to be updated. Since all segments at level i have the same size, we denote p_i as the sequential complexity to update any segment $s_{i,j}$ at level i (TRYINSERT+ in Algorithm 2). GPMA+ evenly distributes the computing resources to each segment. As processing each segment only requires a constant number of scans on the segment by GPU primitives, the complexity for GPMA+ to process level i is $c_i = \frac{p_i}{K/z_i}$. Thus we have:

$$c^X_{\text{GPMA+}} = \sum_{i=0,..,h} \frac{p_i}{K/z_i} = \frac{1}{K} \sum_{i=0,..,h} p_i \cdot z_i \leq \frac{1}{K} \sum_{x \in X} c^x_{\text{PMA}}$$

where c^x_{PMA} is the complexity for PMA to process the update of a particular entry $x \in X$. Given the amortized update complexity of PMA is $c^x_{\text{PMA}} = O(log^2 N)$, we have $c^X_{\text{GPMA+}} = O(\frac{|X| \cdot log^2 N}{K})$. Then the amortized complexity to update one single entry under the GPMA scheme naturally follows as $O(1 + \frac{log^2 N}{K})$.

Finally, we conclude the proof by combining the complexities from all three parts. □

5.3 Handle Other Graph Updates

In this subsection, we discuss how our scheme can support other dynamic graph stream scenarios. The proposed scheme supports dynamic hyper graph maintenance. The hyper graph structure is often represented as a sparse matrix where each row of the matrix is an hyper edge and all vertices in the hyper edge are stored as entries in the row. As the graph formats supported by our scheme are fundamentally identical to sparse matrix

Table 1: Statistics of Datasets

Datasets	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	$\|E_s\|$	$\|E_s\|/\|V\|$
Reddit	2.61M	34.4M	13.2	17.2M	6.6
Pokec	1.60M	30.6M	19.1	15.3M	9.6
Graph500	1.00M	200M	200	100M	100
Random	1.00M	200M	200	100M	100

representations, all proposed algorithms immediately support the dynamic hyper graph scenario.

Vertex insertions/deletions can also be supported. We take the CSR format as an example. Instead of using a fixed size vector for the row offsets as shown in Figure 5, we use a linked list structure where each element in the list is a fixed size vector to store the offsets. Whenever there is a new vertex to insert, we allocate the vertex to the trailing vector of the linked list. In case the trailing vector does not have enough space to store the vertex, we allocate a new vector and append it to the list. To handle a vertex deletion, we logically delete all edges associated with the vertex using techniques described in Section 4.2. Moreover, the row offset data structure can be periodically refreshed as vertex deletions are not common in practice.

5.4 Adapt Existing Solutions Into GPMA+

Last but not least, we explain how to adapt existing GPU-based graph algorithms into GPMA/GPMA+ with ease. For efficiency, many graph processing algorithms are developed with array structures on the GPU [18, 28, 34, 52]. Basically, GPMA/GPMA+ are arrays with gaps. In the study, we are able to efficiently replace the array operations with operations of GPMA/GPMA+.

Algorithm 3 illustrates the pseudocode of the *Neighbor Gathering* parallel procedure, which is a general primitive for most GPU-based vertex-centric graph processing models [36, 18, 22]. The procedure follows the SIMT manner: each thread is assigned to one of the vertex frontiers according to its thread's id (shown in line 2) and the corresponding adjacency list range (shown in line 3), and then tries to vie the control of the group of threads (shown in line 7). All threads will pick one neighbor from the winner's adjlist and perform particular workloads, which means that tasks belonging to the winner's neighbors will be executed in parallel by all involved threads. Compared with the CSR format in static manner, GPMA+ arranges its edges in a continuous space in the similar way but with reserved gaps. In other words, the only change to adapt existing static solutions is to validate the existence of particular elements as highlighted in line 6 of Algorithm 3.

6 Experimental Evaluation

In this section, we present the experimental evaluation of our proposed methods. First, we present the setup of the experiments. Second, we examine the update costs of different schemes for maintaining dynamic graphs. Finally, we implement three different applications to showcase the superior performance of the proposed solutions.

Table 2: Experimented Graph Algorithms and the Compared Approaches

Compared Approaches	Graph Container	BFS	ConnectedComponent	PageRank
CPU Approaches	AdjLists	Standard Single Thread Algorithms		
	PMA [10, 11]			
	Stinger [19]	Stinger built-in Parallel Algorithms		
GPU Approaches	cuSparseCSR [3]	D. Merrill et al.[36]	J. Soman et al.[41]	CUSP SpMV [2]
	GPMA/GPMA+			

6.1 Experimental Setup

Datasets. We collect two real world graphs (Reddit and Pokec) and synthesize two random graphs (Random and Graph500) to test the proposed methods. The datasets are described as follows and their statistics are summarized in Table 1.

- Reddit is an online forum where user actions include post and comment. We collect all comment actions from a public resource[2]. Each comment of a user b to a post from another user a is associated with an edge from a to b, and the edge indicates an action of a has triggered an action of b. As each comment is labeled with a timestamp, it naturally forms a dynamic influence graph.
- Pokec is the most popular online social network in Slovakia. We retrieve the dataset from SNAP [30]. Unlike other online datasets, Pokec contains the whole network over a span of more than 10 years. Each edge corresponds to a friendship between two users.
- Graph500 is a synthetic dataset obtained by using the Graph500 RMAT generator [37] to synthesize a large power law graph.
- Random is a random graph generated by the Erdős-Renyi model. Specifically, given a graph with n vertices, the random graph is generated by including each edge with probability p. In our experiments, we generate a Erdős-Renyi random graph with 0.02% of non-zero entries against a full clique.

Stream Setup. In our datasets, Reddit has a timestamp on every edge whereas the other datasets do not possess timestamps. As commonly used in existing graph stream algorithms [55, 53, 38], we randomly set the timestamps of all edges in the Pokec, Graph500 and Random datasets. Then, the graph stream of each dataset receives the edges with increasing timestamps.

For each dataset, a dynamic graph stream is initialized with a subgraph consisting of the dataset's first half of its total edges according to the timestamps, i.e., E_s in Table 1 denotes the initial edge set of a dynamic graph before the stream starts. To demonstrate the update performance of both insertions and deletions, we adopt a sliding window setup where the window contains a fixed number of edges. Whenever the window slides, we need to update the graph by deleting expired edges and inserting arrived edges until there is no new edge left in the stream.

After completing the update procedure, the graph analytic module (see Section 3) takes over and runs the algorithms on the updated graph.

Applications. We conduct experiments on three most widely used graph applications to showcase the applicability and the efficiency of GPMA+.

- **BFS** is a key graph operation which is extensively studied in previous works on GPU graph processing [24, 33, 13]. It begins with a given vertex (or *root*) of an unweighted

[2]https://www.kaggle.com/reddit/reddit-comments-may-2015

graph and iteratively explores all connected vertices. The algorithm will assign a minimum distance away from the root vertex to every visited vertex after it terminates. In the streaming scenario, after each graph update, we select a random root vertex and perform `BFS` from the root to explore the entire graph.

- **Connected Component** is another fundamental algorithm which are extensively studied under both CPU [25] and GPU [41] μ environment. It partitions the graph where all vertices in a partition can reach the others in the same partition and cannot reach vertices from other partitions. In the streaming context, after each graph update, we run the `ConnectedComponent` algorithm to maintain the up-to-date partitions.
- **PageRank** is another popular benchmarking application for large scale graph processing. Power iteration method is a standard method to evaluate the PageRank where the Sparse Matrix Vector Multiplication (`SpMV`) kernel is recursively executed between the graph's adjacency matrix and the PageRank vector. In the streaming scenario, whenever the graph is updated, the power iteration is invoked and it obtains the up-to-date PageRank vector by operating on the updated graph adjacency matrix and the PageRank vector obtained its previous iteration. In our experiments, we follow the standard setup by setting the damping factor to 0.85 and we terminate the power iteration once the 1-norm error is less than 10^{-3}.

Those three applications have different memory and computation requirements. BFS requires little computation but performs frequent random memory accesses, and PageRank using `SpMV` accesses the memory sequentially and it is the most compute-intensive task among all three applications.

Maintaining Dynamic Graph. We adopt the CSR [32, 6] format to represent the dynamic graph maintained. Note that all approaches proposed in the paper are not restricted to CSR but are general enough to incorporate any popular representation format like COO [16], JAD [39], HYB [9, 34] and many others. To evaluate the update performance of our proposed methods, we compare different graph data structures and respective approaches on both CPU and GPU.

- `AdjLists` (CPU). `AdjLists` is a basic approach for CSR graph representation. As the CSR format sorts all entries according to their row-column indices, we implement `AdjLists` with a vector of $|V|$ entries for $|V|$ vertices and each entry is a RB-Tree to denote all (out)neighbors of each vertex. The insertions/deletions are operated by TreeSet insertions/deletions.
- `PMA` (CPU). We implement the original CPU-based `PMA` and adopt it for the CSR format. The insertions/deletions are operated by `PMA` insertions/deletions.
- `Stinger` (CPU). We compare the graph container structure used in the state-of-the-art CPU-based parallel dynamic graph analytic system, `Stinger` [19]. The updates are handled by `Stinger` internal logic[3].
- `cuSparseCSR` (GPU). We also compare with the GPU-based CSR format used in the NVIDIA cuSparse library [3]. The updates are executed by calling the rebuild function in the cuSparse library.
- `GPMA`/`GPMA+`. Our proposed approaches.

Note that we do not compare with DCSR [29] because, as discussed in Section 2.2, the scheme can neither handle deletions nor support efficient searches, which makes it incomparable to all schemes proposed in this paper.

[3]The sources codes are downloaded from `http://www.stingergraph.com/`

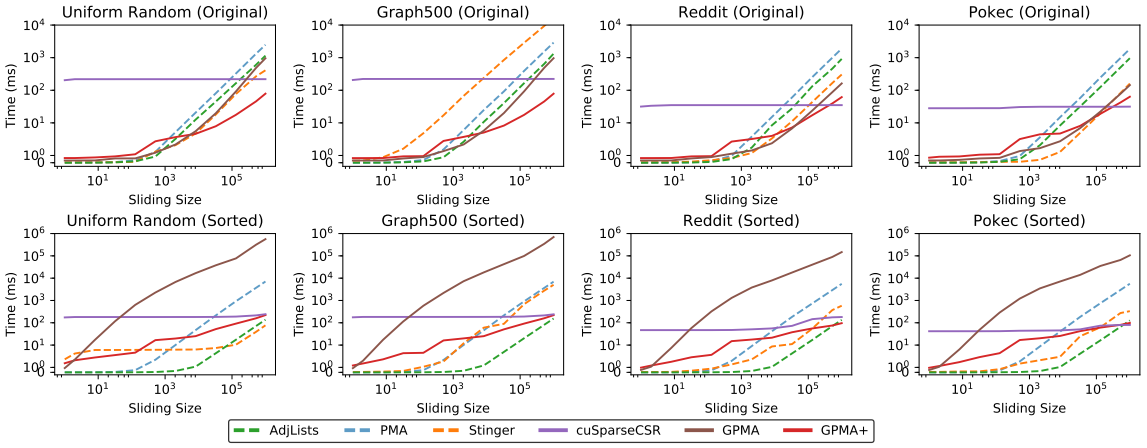


Figure 7: Performance comparison for updates with different batch sizes. The dashed lines represent CPU-based solutions whereas the solid lines represent GPU-based solutions.

To validate if using the dynamic graph format proposed in this paper affects the performance of graph algorithms, we implement the state-of-the-art GPU-based algorithms on the CSR format maintained by `GPMA/GPMA+` as well as `cuSparseCSR`. Meanwhile, we invoke `Stinger`'s built-in APIs to handle the same workloads of the graph algorithms, which are considered as the counterpart of GPU-based approaches in highly parallel CPU environment. Finally, we implement the standard single-threaded algorithms for each application in `AdjLists` and `PMA` as baselines for thorough evaluation. The details of all compared solutions for each application is summarized in Table 2.

Experimental Environment. All algorithms mentioned in the remaining part of this section are implemented with CUDA 7.5 and GCC 4.8.4 with -O3 optimization. All experiments except `Stinger` run on a CentOS server which has Intel(R) Core i7-5820k (6-cores, 3.30GHz) with 64GB main memory and GeForce TITAN X GPU with 12GB device memory, connected with PCIe v3.0 x16. `Stinger` baselines run on a multi-core server which is deployed 4-way Intel(R) Xeon(R) CPU E7-4820 v3 (40-cores, 1.90GHz) with 128GB main memory.

6.2 The Performance of Handling Updates

In this subsection, we compare the update costs for different update approaches. As previously mentioned, we start with the initial subgraph consisting of each dataset's first half of total edges. We measure the average update time where the sliding window iteratively shifts for a batch of edges. To evaluate the impact of update batch sizes, the batch size is set to range from one edge and exponentially grow to one million edges with base two. The update pattern is a consideration for evaluating the performance of update maintenance. Thus, we evaluate the performance under different update patterns: *original* order and *sorted* order. The original order is according to the timestamps whereas, for the sorted order, we sort the edges according to the row-column indices specified in the CSR format and feed them to the streams. Figure 7 shows the average latency for all approaches with different sliding batch sizes under the original and the sorted stream order. Note that the x-axis and y-axis are plotted in log scales.

First, we analyze the update results with the original streaming orders. We observe that,

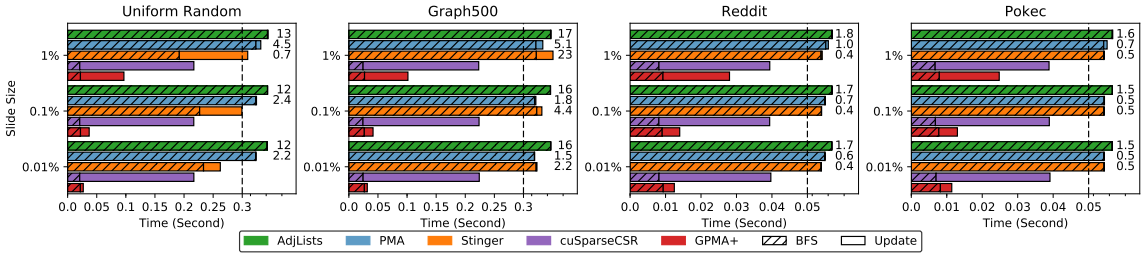


Figure 8: Streaming BFS

PMA-based approaches are very efficient in handling updates when the batch size is small. As batch size becomes larger, the performance of PMA and GPMA quickly degrades to the performance of simple rebuild. Although GPMA achieves better performance than GPMA+ for small batches since the concurrent updating entries are unlikely to conflict, thread conflicts become serious for larger batches. Due to its lock-free approach, GPMA+ shows superior performance over PMA and GPMA. In particular, GPMA+ has speedups of up to 20.42x and 18.30x against PMA and GPMA respectively. Stinger shows impressive update performance in most cases as Stinger efficiently updates its dynamic graph structure in a parallel fashion and the code runs on a powerful mult-core CPU system. For now, multi-core CPU system is considered more powerful than GPUs for pure random data structure maintainance but cost more (in our experimental setup, our CPU server costs more than 5 times that of the GPU server). Moreover, we also note that, Stinger shows extremely poor performance in the Graph500 dataset. According to the previous study [8], the phenomenon is due to the fact that Stinger holds a fixed size of each edge block. Since Graph500 is a heavily skewed graph as the graph follows the power law model, the skewness causes severe performance deficiency on the utilization of memory for Stinger.

For the update results with sorted streaming orders, AdjLists performs the best among all approaches due to its efficient balanced binary tree update mechanism. Meanwhile, a batch of sorted updates makes GPMA very inefficient as all updating threads within the batch conflict. Thanks to the non-locking optimization introduced, the update performance of GPMA+ is still significantly faster than that of the rebuild approach (cuSparseCSR) with orders of magnitude speedups for small batch sizes.

Among the original and sorted stream experiments, we observe the sharp increase for GPMA+ performance curves occur when the batch size is 512. This is because the multi-level strategy is used in GPMA+ (which is mentioned in Section 5.2) and shared-memory constraint cannot support batch size which is more than 512 on our hardware. Finally, the experiments show that, GPMA is faster than GPMA+ when the update batch is smaller and leads to minor thread conflicts, because the GPMA+ logic is more complicated and includes overheads by a number of kernel calls. However, using GPMA only benefits when the update batch is extremely small and the performance gain in such extreme case is also negligible compared with GPMA+. Hence, we can conclude that GPMA+ shows its stability and efficiency for all kinds of update patterns compared with GPMA, and we will only show the results of GPMA+ for the remaining experiments.

6.3 Application Performance

As previously mentioned, all compared application-specific approaches are summarized in Table 2. We find that integrating GPMA+ into an existing GPU-based implementation

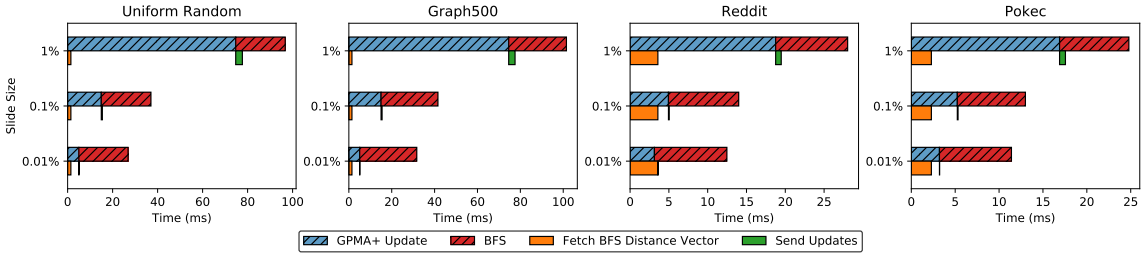


Figure 9: Concurrent data transfer and BFS computation with asynchronous stream

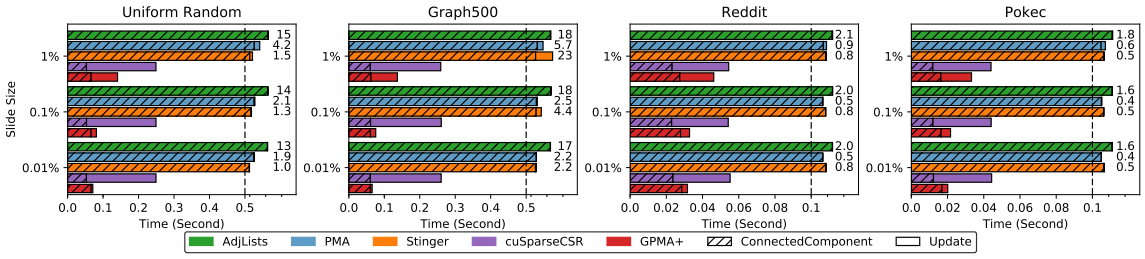


Figure 10: Streaming Connected Component

requires little modification. The major ones are transforming the array operations in the original implementation to the operations on GPMA+, as presented in Section 5.4. The intentions of this subsection are two-fold. First, we test if using the PMA-like data structure to represent the graph brings significant overheads for the graph algorithms. Second, we demonstrate how the update performance affect the overall efficiency of dynamic graph processing.

In the remaining part of this section, we present the performance of different approaches by showing their average elapsed time to process a shift of the sliding window[4] with three different batch sizes, i.e., the batches contain 0.01%, 0.1% and 1% edges of the respective dataset. We only test the stream with the original update order as the completely sorted order is unrealistic in real world scenarios. We distinguish the time spent on updates and analytics with different patterns among all figures.

BFS Results: Figure 8 presents the results for BFS. Although processing BFS only accesses each edge in the graph once, it is still an expensive operation because BFS can potentially scan the entire graph. This has led to the observation that CPU-based approach takes significant amount of time for BFS computation whereas the update time is comparatively negligible. Thanks to the massive parallelism and high memory bandwidth of GPUs, GPU-based approaches are much more efficient than CPU-based approaches for BFS computation as well as the overall performance. For the cuSparseCSR approach, the rebuild process is the bottleneck as the update requires to scan the entire group multiple times. In contrast, GPMA+ takes much shorter time for the update and has nearly identical BFS performance compared with cuSparseCSR. Thus, GPMA+ dominates the comparisons in terms of the overall processing efficiency.

We have also tested our framework in terms of hiding data transfer over PCIe by using asynchronous streams to concurrently perform GPU computation and PCIe transfer. In

[4]We have also tested the graph stream with explicit random insertions and deletions for all applications as an extended experiment. We omit the detailed results here since they are similar to results of the sliding window model and we refer interested readers to Appendix C.

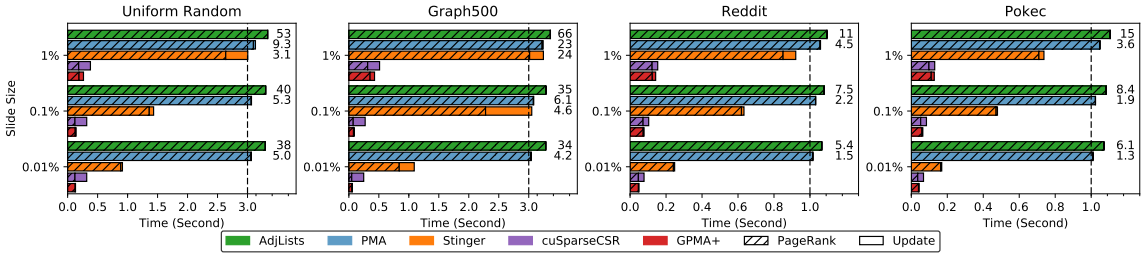


Figure 11: Streaming PageRank

Figure 9, we show the results when running concurrent execution by using the GPMA+ approach. The data transfer consists of two parts: sending graph updates and fetching updated distance vector (from the query vertex to all other vertices). It is clear in the figure that, under any circumstances, sending graph updates is overlapped by GPMA+ update processing and fetching the distance vector is overlapped by BFS computation. Thus, the data transfer is completely hidden in the concurrent streaming scenario. As the observations remain similar in other applications, we omit their results and explanations, and the details can be found in the Appendix B.

Connected Component Results: Figure 10 presents the results for running Connected- Component on the dynamic graphs. The results show different performance patterns compared with that of BFS as ConnectedComponent takes more time to process which is caused by a number of graph traversal passes to extract the partitions. Meanwhile, the update cost remains the same. Thus, GPU-based solutions enhance their performance superiority over CPU-based solutions. Nevertheless, the update process of cuSparseCSR is still expensive compared with the time spent for Connected- Component. GPMA+ is very efficient in processing the updates. Although we have observed that, in the Reddit and the Pokec datasets, GPMA+ shows some discrepancies for running the graph algorithm against cuSparseCSR due to the "holes" introduced in the graph structure, the discrepancies are insignificant considering the huge performance boosts for updates. Thus, GPMA+ still dominates the rebuild approach for overall performance.

PageRank Results: Figure 11 presents the results for Page- Rank. PageRank is a compute-intensive task where the SpMV kernel is iteratively invoked on the entire graph until the PageRank vector converges. The pattern follows from previous results as CPU-based solutions are dominated by GPU-based approaches as iterative SpMV is a more expensive process than BFS and ConnectedComponent and GPU is designed to handle massively parallel computation like SpMV. Although cuSparseCSR shows inferior performance compared with GPMA+, the improvement brought by GPMA+'s efficient update is not as significant as previous applications since the update costs are minor compared with the cost of iterative SpMV kernel calls. Nevertheless, the dynamic structure of GPMA+ does not affect the efficiency of the SpMV kernel and GPMA+ outperforms other approaches in all experiments.

6.4 Overall Findings

We make a summary of our findings. First, GPU-based approaches (cuSparseCSR and GPMA+) outperform CPU-based approaches thanks to our optimizations in taking advan-

tage of the superior hardware of the GPUs, even compared with `Stinger` running on a 40-core CPU server. One of the key reasons is that `GPMA+` and graph analytics can exploit the superb high memory bandwidth and massive parallelism of the GPU, as many graph applications are data- and compute-intensive. Second, `GPMA+` is much more efficient than `cuSparseCSR` as maintaining the dynamic updates is often the bottleneck of continuous graph analytic processing and `GPMA+` avoids the costly process of rebuilding the graph via incremental updates while bringing minimal overheads for existing graph algorithms running its graph structure.

7 Conclusion & Future Work

In this paper, we address how to dynamically update the graph structure on GPUs in an efficient manner. First, we introduce a GPU dynamic graph analytic framework, which enables existing static GPU-oriented graph algorithms to support high-performance evolving graph analytics. Second, to avoid the rebuild of the graph structure which is a bottleneck for processing dynamic graph on GPUs, we propose `GPMA` and `GPMA+` to support incremental dynamic graph maintenance in parallel. We prove the scalability and complexity of `GPMA+` theoretically and evaluate the efficiency through extensive experiments. It is worth noting that the current proposed solution only limits the applications to the memory size of a single GPU. To alleviate the limitations, we would like to explore the multi-GPU as well as hybrid CPU-GPU supports for dynamic graph processing as future works.

Appendices

A TryInsert+ Optimizations

Based on different segment sizes, we propose three optimizations of Function TryInsert+ in Algorithm 2. The motivation is to obtain better memory access efficiency and lower cost of synchronization by balancing between problem scale and hardware hierarchy on GPU. The key computation logic of TryInsert+ is to merge two sorted arrays, i.e., existing segment entries and entries to be inserted. Standard approach for parallel merging needs to identify the position in merged array by binary search and then to execute *parallel map*, which requires heavy and uncoalesced memory accesses. Thus, depending on the size of the merge, we wish to employ different hardware hierarchies on GPU (i.e. warp, block and device) to minimize the cost of memory accesses.

Before presenting the details of our optimizations, Algorithm 4 illustrates how to group threads according to their positions in different hierarchies of GPU architecture and how to target the groups to their assigned segments. In particular, each thread is assigned with a lane id, a block id and a global thread id to indicate the position of the thread in the corresponding warp, block and device work group. Each thread is assigned for one GPMA+ segment and the thread will ask other threads in the same work group to cooperate for its task. This means that each thread tries to drive a group of threads to deal with the assigned segment. Such a strategy lifts thread divergences caused by different execution branches. Note that this assignment policy will be used in our warp and block based optimizations as an initialization function.

Algorithm 5 shows the Warp-Based optimization for any segments with entries no larger than the warp size. This implementation has high efficiency because explicit synchronization is not needed and all data is stored in registers. For each iteration, all threads of a particular warp will compete for the control of the warp as shown in line 11. The winner will drive the other threads in this warp to handle its required computation steps of the corresponding segment. As an example, line 27 counts valid entries in the segment concurrently. Lines 32-34 omit the remaining computation steps in TryInsert+, such as merging insertions and redistributing entries of segments, as their computation paradigm is similar to counting entries.

Algorithm 6 shows the Block-Based optimization. It utilizes the shared memory, which has a higher volume than registers, to store data. Even though explicit synchronization is needed in line 12 and line 32 to guarantee consistency, synchronization in a block is highly optimized in GPU hardware and thus it does little effect to the overall performance. Both Warp-Based and Block-Based optimizations explicitly accommodate GPU features. As discussed in Section 5.2, although these two methods have limited memory for efficient access, they can handle most of the update requests.

Algorithm 7 shows the Device-Based implementation. The implementation is different from the ones in Warp and Block based approaches, because it is designed for segments having a size larger than the shared memory size. Under this scenario, we have to handle them in the GPU's global memory. One possible approach is to invoke an independent kernel for each large segment, but it will take considerable costs to initialize and schedule for multiple kernels. Hence, we propose an approach to handle a large number segments by only invoking a few unique kernel calls.

We illustrate the idea by showing how to perform counting segments which are valid for insertions as an example. As shown in lines 5 and 12, all valid entries stored in `GPMA+` segments are first marked, and then all valid entry counts are calculated by *SumReduce* in one kernel call. Line 16 generates valid indexes for segments which have enough free space to receive their corresponding insertions, which is used by the rest computation steps. Simply speaking, our approach executes in horizontal steps of the execution logic, in order to avoid load imbalance caused by branch divergences. Finally, merging and segment entries redistribution use the same mechanism and thus are omitted.

Algorithm 4 TryInsert+ Initialization

```
1  inline function ConstInit( void ) {
2    // cuda protocol variables
3    WARPS = blockDim / 32;
4    warp_id = threadIdx / 32;
5    lane_id = threadIdx % 32;
6    thread_id = threadIdx;
7    block_width = gridDim;
8    grid_width = gridDim * blockDim;
9    global_id = block_width * blockIdx + threadIdx;
10
11   // infos for assigned segment
12   seg_beg = segments[global_id];
13   seg_end = seg_beg + segment_width;
14
15   // infos for insertions belong current segment
16   ins_beg = offset[global_id];
17   ins_end = offset[global_id + 1];
18   insert_size = ins_end - ins_beg;
19
20   // the upper number that current segment can hold
21   upper_size = tau * segment_size;
22 }
```

Algorithm 5 TryInsert+ Warp-Based Optimization

```
1  kernel TryInsert+(int segments[m], int offsets[m],
2      int insertions[n], int segment_width) {
3
4    ConstInit();
5
6    volatile shared comm[WARPS][5];
7    warp_shared_register pma_buf[32];
8
9    while (WaryAny(seg_end - seg_beg)) {
10     // compete for warp
11     comm[warp_id][0] = lane_id;
12
13     // winner controls warp in this iteration
14     if (comm[warp_id][0] == lane_id) {
15       seg_beg = seg_end;
16       comm[warp_id][1] = seg_beg;
17       comm[warp_id][2] = seg_end;
18       comm[warp_id][3] = ins_beg;
19       comm[warp_id][4] = ins_end;
20     }
21
22     memcpy(pma_buf, pma[seg_beg], segment_width);
23     // count valid entries in this segment
24     entry_num = 0;
25     if (lane_id < segment_width) {
26       valid = pma_buf[lane_id] == NULL ? 0 : 1;
27       entry_num = WarpReduce(valid);
28     }
29
30     // check upper density if insert
31     if (entry_num + insert_size) < upper_size) {
32       // merge insertions with pma_buf
33       // evenly redistribute pma_buf
34       // mark all insertions successful
35       memcpy(pma[seg_beg], pma_buf, segment_width);
36     }
37   }
38 }
```

Algorithm 6 TryInsert+ Block-Based Optimization

```
1  kernel TryInsert+(int segments[m], int offsets[m],
2      int insertions[n], int segment_width) {
3
4    ConstInit();
5
6    volatile shared comm[5];
7    volatile shared pma_buf[segment_width];
8
9    while (BlockAny(seg_end - seg_beg)) {
10     // compete for block
11     comm[0] = thread_id;
12     BlockSynchronize();
13
14     // winner controls block in this iteration
15     if (comm[0] == lane_id) {
16       seg_beg = seg_end;
17       comm[1] = seg_beg;
18       comm[2] = seg_end;
19       comm[3] = ins_beg;
20       comm[4] = ins_end;
21     }
22
23     memcpy(pma_buf, pma[seg_beg], segment_width);
24     // count valid entries in this segment
25     entry_num = 0;
26     ptr = thread_id;
27     while (ptr < segment_width) {
28       valid = pma_buf[ptr] == NULL ? 0 : 1;
29       entry_num += BlockReduce(valid);
30       thread_id += block_width;
31     }
32     BlockSynchronize();
33
34     // same as lines 30-37 in Algorithm 5
35   }
36 }
```

Algorithm 7 TryInsert+ Device-Based Optimization

```
1  function TryInsert+(int segments[m], int offsets[m],
2      int insertions[n], int segment_width) {
3
4    int valid_flags[m * segment_width];
5    parallel foreach i in range(m):
6      parallel foreach j in range(segment_width):
7        if (pma[segments[i] + j] != NULL) {
8          valid_flags[i * segment_width + j] = 1;
9        }
10   DeviceSynchronize();
11   int entry_nums[m];
12   DeviceSegmentedReduce(valid_flags, m,
13       segment_size, entry_nums);
14   DeviceSynchronize();
15   int valid_indexes[m];
16   parallel foreach i in range(m):
17     if (entry_nums[i] + insert_size < upper_size) {
18       valid_indexes[i] = i;
19     }
20   DeviceSynchronize();
21   RemoveIfTrue(valid_indexes);
22   DeviceSynchronize();
23   // according to valid_indexes, segmentedly to:
24   //   merge insertions into segments
25   //   evenly redistribute segments
26   //   mark all insertions successful
27 }
```

B Additional Experimental Results For Data Transfer

We show the experimental results for using asynchronous streams for concurrently transmitting data on PCIe and running computations on the GPU. We only show the results for `GPMA+`.

In `ConnectedComponent`, the data transferred on PCIe consists of two parts: the graph updates and the component label vector to all vertices computed by `ConnectedComponent`. In `PageRank`, the result vector to be fetched indicates PageRank scores, which has the same size as `ConnectedComponent`'s. The results in Figures 12 and 13 have shown that the data transfer is completely hidden by analytic processing on GPU and `GPMA+` update.

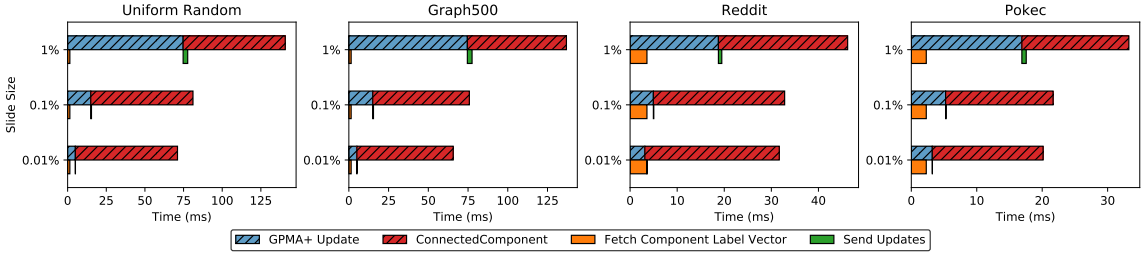


Figure 12: Concurrent data transfer and Connected Component computation with asynchronous stream

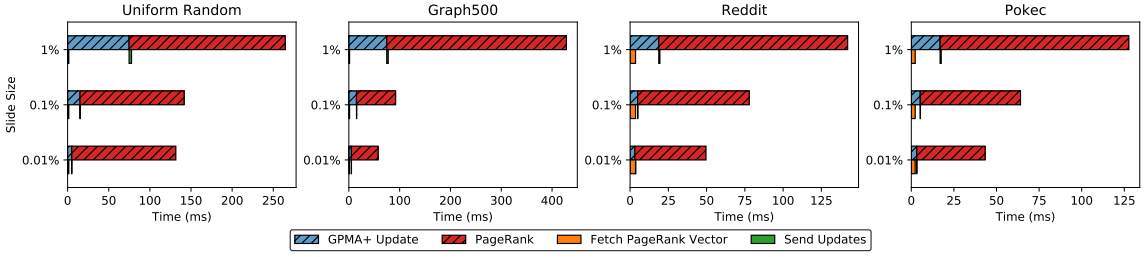


Figure 13: Concurrent data transfer and PageRank computation with asynchronous stream

C Additional Experimental Results For Graph Streams with Explicit Deletions

We present the experimental results for graph streams which involve explicit deletions. In this section, we use the same stream setup which is mentioned in Section 6.1. However, for each iteration of sliding, we will randomly pick a set of edges belonging to current sliding window insteading of the head part as edges to be deleted.

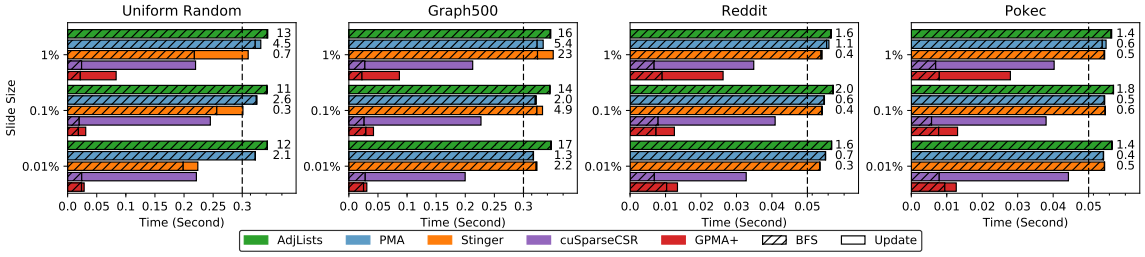


Figure 14: Streaming BFS with explicit deletions

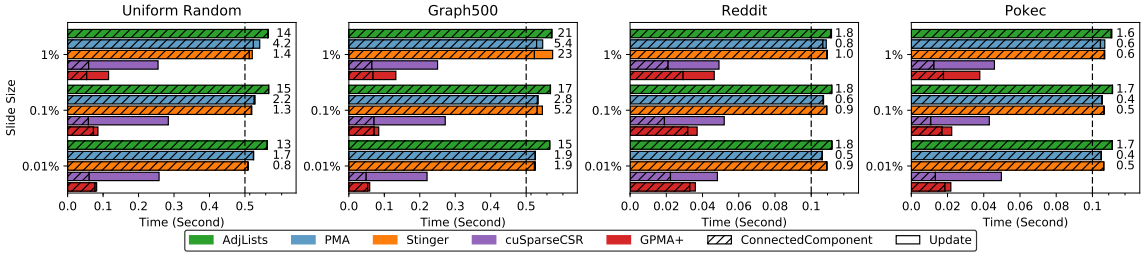


Figure 15: Streaming Connected Component with explicit deletions

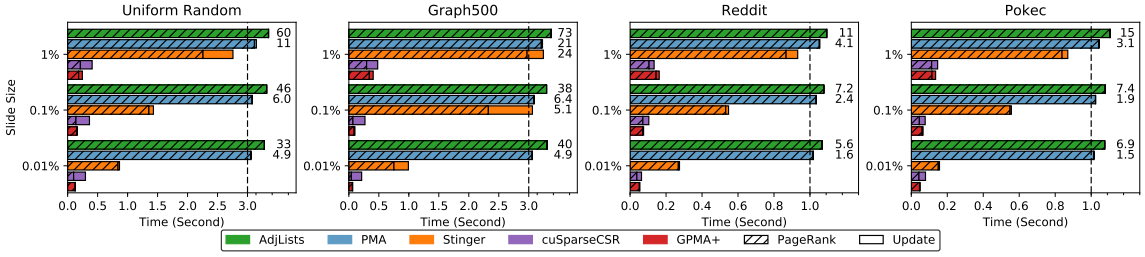


Figure 16: Streaming PageRank with explicit deletions

Figures 14, 15 and 16 illustrate the results of three streaming applications respectively. Note that we pick sets of edges to be deleted in advance, which means that for each independent baseline, it handles the same workload all the time. Since there is no intrinsic difference between expiry and explicit deletions, the results are similar to sliding window's. The subtle difference in the results are mainly due to different deletions which lead to various applications' running time.

References

[1] Apache flink. https://flink.apache.org/. Accessed: 2016-10-18.

[2] Cusp library. https://developer.nvidia.com/cusp. Accessed: 2017-03-25.

[3] cusparse. https://developer.nvidia.com/cusparse. Accessed: 2016-11-09.

[4] CUDA UnBound (CUB) library. https://nvlabs.github.io/cub/, 2015.

[5] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Min. Knowl. Discov.*, 29(3):626–688, 2015.

[6] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *SC*, pages 781–792, 2014.

[7] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus. In *ICS*, pages 273–282, 2014.

[8] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[9] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[10] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[11] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32(4), 2007.

[12] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD*, pages 251–264, 2015.

[13] F. Busato and N. Bombieri. Bfs-4k: an efficient implementation of bfs for kepler gpu architectures. *TPDS*, 26(7):1826–1838, 2015.

[14] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[15] M. S. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms*, pages 337–348. Springer, 2013.

[16] H.-V. Dang and B. Schmidt. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science*, 9:57–66, 2012.

[17] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[18] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359. IEEE, 2014.

[19] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. STINGER - High performance data structure for streaming graphs. *HPEC*, 2012.

[20] M. Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, 2011.

[21] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[22] Z. Fu, M. Personick, and B. Thompson. *MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs.* A High Level API for Fast Development of High Performance Graph Analytics on GPUs. ACM, New York, New York, USA, June 2014.

[23] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *Proc. VLDB Endow.*, 5(12):2030–2031, 2012.

[24] P. Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International Conference on High-Performance Computing*, pages 197–208. Springer, 2007.

[25] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 55–57. ACM, 1976.

[26] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 5:1–5:6, 2016.

[27] A. P. Iyer, L. E. Li, and I. Stoica. Celliq : Real-time cellular network analytics at scale. In *NSDI*, pages 309–322, 2015.

[28] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali. Synchronization trade-offs in gpu implementations of graph algorithms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 514–523. IEEE, 2016.

[29] J. King, T. Gilray, R. M. Kirby, and M. Might. Dynamic sparse-matrix allocation on gpus. In *ISC*, pages 61–80, 2016.

[30] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *TIST*, 8(1):1, 2016.

[31] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi. Efficient subgraph matching using gpus. In *ADC*, pages 74–85, 2014.

[32] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD*, pages 403–416, 2016.

[33] L. Luo, M. Wong, and W.-m. Hwu. An effective gpu implementation of breadth-first search. In *DAC*, pages 52–55, 2010.

[34] M. Martone, S. Filippone, S. Tucci, P. Gepner, and M. Paprzycki. Use of hybrid recursive csr/coo data structures in sparse matrix-vector multiplication. In *IMCSIT*, pages 327–335. IEEE, 2010.

[35] A. McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, 2014.

[36] D. Merrill, M. Garland, and A. Grimshaw. High-Performance and Scalable GPU Graph Traversal. *TOPC*, 1(2), 2015.

[37] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. 2010.

[38] N. Ohsaka, T. Maehara, and K.-i. Kawarabayashi. Efficient pagerank tracking in evolving networks. In *KDD*, pages 875–884, 2015.

[39] Y. Saad. Numerical solution of large nonsymmetric eigenvalue problems. *Computer Physics Communications*, 53(1):71–90, 1989.

[40] D. Sayce. 10 billions tweets, number of tweets per day. `http://www.dsayce.com/social-media/10-billions-tweets/`. Accessed: 2016-10-18.

[41] J. Soman, K. Kothapalli, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. *IPDPS Workshops*, 2010.

[42] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[43] J. A. Stratton, N. Anssari, C. Rodrigues, I.-J. Sung, N. Obeid, L. Chang, G. D. Liu, and W.-m. Hwu. Optimization and architecture effects on gpu computing workload performance. In *InPar*, pages 1–10, 2012.

[44] N. Tang, Q. Chen, and P. Mitra. Graph stream summarization: From big bang to big crunch. In *SIGMOD*, pages 1481–1496, 2016.

[45] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[46] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. DOULION: counting triangles in massive graphs with a coin. In *SIGKDD*, pages 837–846, 2009.

[47] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson. Scheduling processing of real-time data streams on heterogeneous multi-gpu systems. In *SYSTOR*, page 7, 2012.

[48] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 50(8):265–266, 2015.

[49] Y. Wang, Q. Fan, Y. Li, and K.-L. Tan. Real-time influence maximization on dynamic social streams. In *Proc. VLDB Endow.*, 2017.

[50] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaspmv: yet another spmv framework on gpus. In *SIGPLAN Notices*, volume 49, pages 107–118, 2014.

[51] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: Implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, 2011.

[52] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, 2011.

[53] Y. Yang, Z. Wang, J. Pei, and E. Chen. Tracking influential nodes in dynamic networks. *arXiv preprint arXiv:1602.04490*, 2016.

[54] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[55] H. Zhang, P. Lofgren, and A. Goel. Approximate personalized pagerank on dynamic graphs. *arXiv preprint arXiv:1603.07796*, 2016.

[56] Y. Zhang and F. Mueller. Gstream: A general-purpose data streaming framework on GPU clusters. In *ICPP*, pages 245–254, 2011.

[57] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, 2014.