An Efficient Data Structure for Dynamic Graph on GPUs

Lei Zou, Fan Zhang, Yinnian Lin, Yanpeng Yu

Abstract—There is a growing interest to offload dynamic graph computation to GPU and resort to its high parallel processing ability and larger memory bandwidths compared with CPUs. The existing GPU graph systems usually use compressed sparse row (CSR) as the de-facto structure. However, CSR has a critical weakness for dynamic change due to the large overhead of re-balance process after update. GPMA+ is a state-of-art dynamic PMA-based structure that uses PMA structure and segment-oriented parallel update procedure to address the dynamic weakness of CSR, but it still has a bottleneck on the array expansion. In this paper, we propose an leveled structure (called LPMA) instead of continue array to retain low time complexity and high parallel update and lift the expansion bottleneck of GPMA+. More specifically, we propose a series of optimization techniques, including bottom-up update, top-down update and on-demand hybrid update strategies as well as consistence-guaranteed parallel processing for update-query mixed workloads. We theoretically analyze the benefits of LPMA compared in terms of re-balance cost during updates. Extensive experiments on four large real-life graphs prove the superiority of LPMA compared with the-state-of-arts.

Index Terms—Dynamic Graph, GPU, Graph Data Structure.

1 Introduction

In the era of big data, graph is used to model complex relations between entities. Massive graph processing has emerged as the de-facto standard in many relation-oriented big data analysis, such as network traffic connection, social network and communication log analysis. Different from static graph processing, high-throughput dynamic graphs propose more challenges. For example, *dynamic graphs* model continuously-updated connections between different entities rather than *isolated* items in traditional data streams [1]. The dynamic connectivity through network packets between different sites is vital for network analysis. More dynamic graph examples include social networks and real-time communication log analysis for troubleshooting in data centers.

One of major challenges in dealing with dynamic graph processing is the high dynamicity. For example, Alibaba e-commerce activity graph is being updated 20,000 edges per second at the peak and Twitter has about 100 million users login daily, with around 500 million tweets per day. Network traffic data averages to about 10^9 packets per hour per router in large data centers [1], [2], [3].

To address great challenges due to high dynamicity and complexity of graph algorithms, one way is to employ hardware assist. There is a growing interest to offload dynamic graph computation to GPU and resort to its high parallel processing ability and larger memory bandwidths compared with CPUs. Thus, in this paper, we propose an efficient GPU-oriented dynamic graph system. Updates and queries are received and buffered by the CPU part, which sends the batches of updates and queries to the GPU part periodically. The GPU part maintains a continuously updating graph and queries are conducted over the dynamic graphs in GPU.

Although GPU-based graph processing systems have been studied extensively [4], [5], [6], [7], most of existing work focus on static graphs except for some recent work [8], [9], [10], [11]. An efficient GPU-oriented dynamic graph data structure is a technical challenging issue. Due to the unpredictable node degree distribution of dynamic graphs, we cannot reserve a continuous space for the neighbor list of each node. On the other hand, graph algorithms usually require an ordered edge array for better performance. For example, to check edge existence, we can avoid scanning the whole neighbor list in an ordered edge array. Furthermore, the ordered neighbor lists benefit the list intersection in many graph algorithms, such as triangle counting [12] and subgraph matching [13]. Therefore, a desirable data structure should trade off the update efficiency and graph edge ordering.

Generally speaking, there are four categories in designing GPU-based dynamic graph data structures to address the challenges: neighbor list based, hash based, compressed sparse row (CSR) based and packed memory array (PMA) based.

- *The CSR based.* The CSR based structures [14] maintain a variable and ordered edge array to store dynamic graphs. The benefit of CSR based structures is the generality of supporting off-the-shelf GPU graph libraries (such as Gunrock[4]) and algorithms [12], [13], since most of them rely on CSR format and the ordered edge array facilitates converting a snapshot of dynamic graphs into CSR representation.
- *The neighbor list based.* The neighbor list based structures [9], [11] maintain a nodes array and a neighbor list for each node. To handle the dynamic updates, this kind of approaches is to design a variable array or list to store the dynamic neighbor lists. The neighbor list could be ordered or unordered.
- *The hash based.* The hash based structure [10] uses GPU based hash slabs to store the neighbor list for each node. Due to the hash table implantation, the hash based structure could achieve $O(1)$ time complexity for update. However, hash-based data structure cannot guarantee edge

- *All authors are from Peking University in China; E-mail: {zoulei,zhangfanau,linyinnian,yuyanpeng}@pku.edu.cn;*
- *Lei Zou is the corresponding author of this work.*

orderness in the neighbor list. Furthermore, [10] requires presetting each node's neighbor list length to reserve the hash table size, which is obviously impractical in many dynamic graphs.

- *The PMA-based.* The packed-memory array (PMA)[15], [16] is a dynamic data structure for a sorted array, which can be used to store sorted edges. Different from the traditional sorted edge array in CSR, PMA allows gaps among the elements so that only a small number of elements need to be shifted around on an insert or delete. GPMA+ [8] proposes a parallel dynamic PMA maintenance (insertion and deletion) algorithm on GPU to store dynamic graphs.

Our proposed dynamic graph data structure, called LPMA, belongs to the PMA-based category. Although CSR is the de-facto graph representation in many GPU graph systems, such as Gunrock[4] and Medusa[7], the dynamic maintenance is the bottleneck of CSR on dynamic graph applications. The main idea of GPMA+ is to maintain sorted edges in a contiguous fashion by reserving spaces to accommodate updates with a constant bounded gap ratio and propose a parallel dynamic maintenance (insertion and deletion) algorithm on GPU. Edge insertions are merged and re-distributed into the eligible segment range in a balanced fashion (called re-balance). GPMA+ has two bottlenecks related to the re-balance: unnecessary re-balance during the expansion re-balancing and redundant re-balance during the non-expansion re-balancing.

With the increasing of edge insertions, GPMA+ [8] will expand and re-allocate a double-size continuous memory space and re-balance the whole edge array. The massive unnecessary re-balances are carried out during the expansion. In order to address the unnecessary re-balance bottleneck, we propose a novel leveled data structure to maintain the sorted edge array in this paper, called *l*eveled *p*acked *m*emory *a*rray (LPMA). Instead of storing all edges in a continuous space in GPMA+, LPMA partitions the whole sorted edge array into different physical levels and stores them in a perfect binary tree[1]. Different from GPMA+, LPMA maintains the edge ordering according to in-order traversal over the binary tree and does not requires a physically continuous memory space to accommodate all edges. The leveled structures in LPMA have three benefits.

- First, during expansion, we only need to allocate a new level and append it to the current LPMA tree rather than re-allocating a double-size continuous space in GPMA+. It alleviates memory allocation cost and memory fragmentation issue in GPMA+.
- Second, after expansion, LPMA employs a *localized re-balance* strategy that reduces the unnecessary re-balances significantly. GPMA+ always performs *global re-balance*, collecting all edges and re-distributing them in the whole expanded space, but LPMA re-balances edges between some local consecutive edge segments. Obviously, our method can reduce the data movement during re-balancing. Theoretical analysis that our method can save more than 90% unnecessary re-balancing cost (Section 3.3) and experiment results also confirm that.
- Last but not least, LPMA is a GPU-friendly data structure. For example, the leveled structure fits the memory hierarchy of GPU and we can use shared memory to cache top levels to improve performance.

1. A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

Both GPMA+ and LPMA adopt the bottom-up re-balancing strategy, which leads to lots of redundancies in the high level re-balancing and cause the latency spikes in the non-expansion re-balancing. To alleviate that, we propose a top-down re-balancing method that guarantees *redundancy-free* in LPMA. To differentiate, the original LPMA with the bottom-up re-balancing strategy is called LPMA-B and the top-down one is named as LPMA-T. Although LPMA-T avoids redundancies in the non-expansion re-balancing, it bring more extra probing costs. Therefore, we propose a hybrid strategy (called LPMA-H) that balances the redundant re-balances of the bottom-up strategy and the extra cost of the top-down strategy at the same time.

Figure 1 compares the latency of insertion batches in GPMA+ and LPMA-H. First, our LPMA-H's average latency is faster than GPMA by one order of magnitude. Second, although both GPMA+ and LPMA-H have latency spikes, the frequency of performance churn of LPMA-H is less than GPMA+ and the maximum amplitude of LPMA-H is also smaller (up to $20\times$ in GPMA+, but only $15\times$ in LPMA). The reason is that LPMA-H alleviates the re-balance cost during expansion (the analysis in Section 3.3) and avoid high-level redundant re-balancing (the top-down strategy in Section 4). Deletions are the dual operation of insertions and have similar performance periodic jitter issue. We omit the discussion about deletion to reduce the repetition.

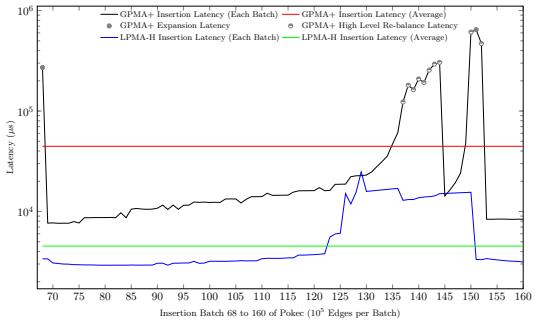


Fig. 1: Insertion Performance Churn Caused By High Level Re-balance and Expansion Process of GPMA+ and Insertion Performance Churn Flatted By LPMA-H

Besides the optimization for update-only workloads, we also propose a con-concurrent strategy (called Opera) for mixed updates and queries batch processing on GPU. We define the *streaming consistence* (Definition 7) for mixed workloads and propose the consistence-guaranteed con-concurrent processing to maximize the parallel power of GPU and accelerate the dynamic graph analysis. To summarize, we made the following contributions.

- We propose a novel leveled data structure LPMA in the context of dynamic graphs. LPMA addresses two performance problems in existing PMA-based GPU dynamic graph data structure:
  - The hierarchy organization and localized re-balance in LPMA alleviate unnecessary re-balancing cost significantly. We theoretically analyze the benefit of our approach in terms of data movement saved by LPMA. We also study some GPU-friendly designs for LPMA.
  - We propose three different re-balancing strategies in LPMA, i.e., the bottom-up, the top-down and

TABLE 1: Notations

Notation	Description
v_n	Node
σ_n	Operation received at t_n
q_n	Query received at t_n
s_n	Leaf layer segment n
B_n	Batch number n
$level$	Physical structure of the segments in LPMA. $level_n$ contains 2^{n-1} leaf segments
$layer$	Logical binary tree of the segments. $layer_n$ contains 2^n leaf segments
leaf layer	the bottom layer of the logical binary tree of the segments
root layer	the root layer of the logical binary tree of the segments
$[s_0, s_3]$	GPMA+ $layer_2$ segment that contains leaf layer segments: $\{s_0, s_1, s_2, s_3\}$
$\{s_0, s_4, s_2, s_5\}$	LPMA $layer_2$ segment that contains leaf layer segments: $\{s_0, s_4, s_2, s_5\}$

the hybrid. The hybrid one not only reduces redundant re-balances in the bottom-up strategy but also alleviate the extra cost of the top-down approach.

- We propose a con-concurrent strategy Opera for the mixed updates and queries batch to maximize the parallel power of GPU.
- Extensive experiments on large dynamic graphs confirm that LPMA outperforms state-of-the-art methods significantly and the con-concurrent framework accelerates the dynamic graph analysis.

2 PRELIMINARY

In this section, we first formally define dynamic graph model and our problem; and then introduce GPMA+ [8], a dynamic PMA based data structure on GPU. Our proposed data structure LPMA also belongs to CSR-based methods and optimizes the most costly expansion and re-balance operation in GPMA+. The details of LPMA are given in Section 3. More related work discussions are given in Section 8.

2.1 Problem Definition

To define the dynamic graph model, we define the following operation stream. Table 1 lists all notations that are used throughout the paper.

Definition 1 (Operation Stream)**.** *An operation stream is a time-evolving sequence of operations $\{\sigma_1, \sigma_2, ...\sigma_x\}$, where each σ_i specifies an edge insertion or deletion or a query primitive (defined in Definition 4) at time t_i.*

In this paper, we adopt the *edge append* model to define an edge insertion and deletion uniformly.

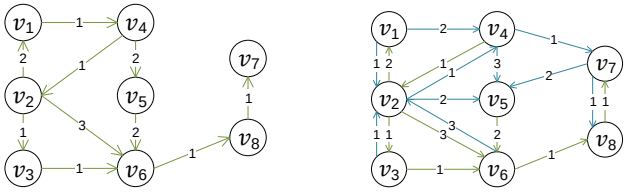
Definition 2 (Edge Update)**.** *An operation σ_i is denoted as $\sigma_i(\overrightarrow{uv}, w_i, t_i)$, i.e., appending edge $\overrightarrow{uv}$ with edge weight w_i at time t_i. Specifically,*

- *if edge $\overrightarrow{uv}$ does not exist before and weight $w_i > 0$, σ_i is an* edge insertion*;*
- *If edge $\overrightarrow{uv}$ exists before, the edge weight w_i will be accumulated to the existing one.*
  - *If the accumulated edge weight is less than 0, σ_i is an* edge deletion*;*
  - *otherwise, the corresponding edge weight is* updated*.*

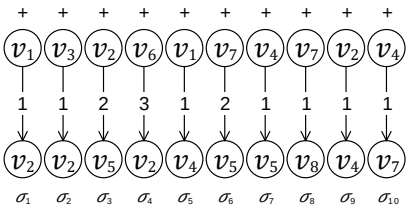
For the simplicity of presentation, edge insertion, deletion and weight update are all entitled as edge update *in this paper.*

Actually, the edge append model can also handle vertex insertion and deletion. To insert an isolated vertex v, we can insert a specific edge insertion $(\overrightarrow{v, +\infty})$. Also, to delete a vertex v, we can delete all edges adjacent to vertex v. For simplicity, we only discuss edge insertion and deletion in the following discussion.

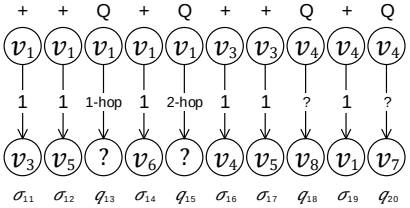
Given an original graph $\mathcal{G}$ with 10 edges in Figure 2a, after a sequence of updates $\{\sigma_1, \sigma_2, ...\}$ (Figure 2c), we can obtain the updated graph in Figure 2b.



(a) Original Graph $\mathcal{G}$ (b) Updated Graph $\mathcal{G}_{t_{10}}$



(c) Operation Stream 1



(d) Operation Stream 2 (Mixed Batch)

Fig. 2: Dynamic Graph with Operation Stream

Definition 3 (A Snapshot of a Dynamic Graph)**.** *Given a dynamic graph $\mathcal{G}$ and a time point t_i, the snapshot of $\mathcal{G}$ at t_i is a graph $\mathcal{G}_{t_i}$, which is the updated graph after performing all edge updates before time t_i in the operation stream.*

Besides edge updates, an operation stream can also include some query operations. We consider the following query primitives (Definition 4). A more complex query task is often decomposed into a series of query primitives.

Definition 4 (Query Primitives)**.** *Each query primitive is denoted as $q(C, t_i)$, where C is a query condition and t_i defines the issue timestamp of query q. We define the following query primitives over graph snapshot $\mathcal{G}_{t_i}$:*

1) ***Edge Query:*** *If $C = (u, v)$ where u and v are the node IDs, return $\{(\overrightarrow{uv}, t, w) | (\overrightarrow{uv}, t, w) \in \mathcal{G}_{t_i}\}$; If the edge doesn't exist, return ϕ.*
2) ***1-hop Successor Query:*** *If $C = (u, \rightarrow)$ where u is a node ID, return $\{(\overrightarrow{uv}, t, w) | (\overrightarrow{uv}, t, w) \in \mathcal{G}_{t_i}\}$. If the edge doesn't exist, return ϕ.*

3) ***1-hop Predecessor Query:*** *If $C = (v, \leftarrow)$ where v is a node ID, return $\{(\overrightarrow{uv}, t, w) | (\overrightarrow{uv}, t, w) \in \mathcal{G}_{t_i}\}$. If the edge doesn't exist, return ϕ.*

Due to great parallel processing ability of GPU, in this paper, we aim to design an efficient *GPU-oriented data structure* to support high throughput updates and queries over the underlying graph. Different from CPU-oriented solutions, the dynamic graph on GPUs usually adopt the *batch-based* model [8]. The system buffer module batches edge updates on CPU side and periodically sends the updating batches to update module in GPU side. The update module process all edge updates (in one batch) simultaneously. Actually, we also package issued queries with edge updates that are received in a period of time in one batch. In Section 6, we will study how to process operation primitive (including both updates and query primitives) oriented parallel strategy. Formally, a batch of operations is defined as follows.

Definition 5 (Operation Batch)**.** *A batch $B_t = \{\sigma_i\}$, where σ_i specifies an edge update or a query primitive (defined in Definition 4) at time t_i. All operations are sorted according to their associated timestamps. Furthermore, the batch size $|B_t| \leq \theta$, where θ is a tuning parameter.*

Generally, there are two reasons of adopting batch based model in GPU-oriented dynamic graph processing.

- First, due to high parallel capability of GPU, the batch model enables GPU to process multiple updates and queries at the same time. Obviously, one operation-at-a-time (sequential model) limits parallel computing of GPU.
- Second, we need to minimize the cost of PCIe transferring on designing GPU-oriented systems. The one operation-at-a-time transferring model cannot reach the full capacity of PCIe bandwidth. For example, the 16-lane PCIe could reach $64GBs$ per second [17]. Thus, the batch model can minimize the average transferring latency. Furthermore, the batch model allows overlapping data transfer with computing tasks in GPU side in a pipeline fashion, which further hides the PCIe communication cost.

2.2 GPMA+: Existing PMA-based Dynamic Graph Data Structure on GPU

Generally, existing dynamic graph structures on GPUs could be divided into four categories: CSR (compressed sparse row) based, neighbor list based, hash-based and PMA based. Since our proposed data structure (LPMA) belongs to PMA based structure, to better understand the benefit of our method, we introduce GPMA+ [8], a state-of-the-art dynamic PMA based data structure. We will review other dynamic graph structures on GPU in the related work section (Section 8).

The compressed sparse row (CSR) [14] is the de-facto graph representation in existing GPU graph systems and applications (e.g., Gunrock[4] and Medusa[7]) due to compact representation, good memory locality and friendly supporting massive-parallel processing. CSR compresses an adjacent matrix (of a sparse graph) into three arrays: row offset array (corresponding to vertices), column array (corresponding to edges) and values array (corresponding to edge weights). In CSR, entry i and $i+1$ in the row offsets array will store the starting and ending offsets for row i of the matrix, which correspond to the neighbors of the i-th vertex in graph (i.e., N_{v_i}). Figure 3 illustrates an example of CSR corresponding to the original graph $\mathcal{G}$ in Figure 2a.

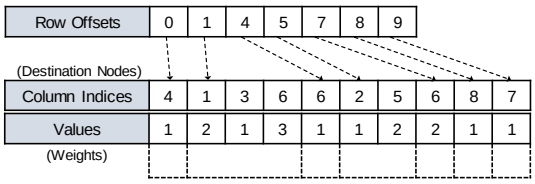


Fig. 3: Compressed Sparse Row (CSR) of $\mathcal{G}$

However, CSR is costly for dynamic updates to graphs, since all vertex's adjacent edges are stored consecutively in a sorted column array. Any edge insertion or deletion will lead to $O(|E|)$ data movements in the sorted array. GPMA+ [8] adopts Packed Memory Array (PMA) [15], [16] to maintain sorted arrays in CSR for dynamic graphs. PMA maintains a sorted array, but it leaves gaps to accommodate fast updates with a bounded gap ratio. GPMA+ extends PMA to GPU platform and use it to store sorted arrays in CSR. They propose a segment-oriented operations to parallelize edge insertions/deletions in a lock-free model. However, the expansion and re-balancing cost is still expensive when facing high throughput updates in dynamic graphs. Note that our proposed LPMA (leveled Packed Memory Array) is an optimized data structure compared with GPMA+. To better understand the benefit of our method, we briefly introduce GPMA+ by an example.

Figure 4 shows an example of GPMA+ and segment-oriented update. GPMA+ divides the edge array into fixed size chunks known as *segments*. For simplicity, we assume that the lower and upper bound density threshold in each segment is $[0, 100\%]$. Given a batch of edge insertions, GPMA+ sorts these edges by source and destination node IDs. Then for each edge insertion, GPMA+ assigns one thread to perform the binary search to locate the associated segments. Given ten edges to be inserted in Figure 4, we first locate four associated leaf layer segments s_0, s_1, s_2 and s_3.

GPMA+ adopts a *bottom-up* update strategy to perform updates and the first round is conducted over leaf layer segments. A thread or a warp (depends on the size of the segment) is called to handle one segment update. After the first round, segment s_1 is updated successfully and updates on segments s_0, s_2 and s_3 fail due to the lack of space. For example, there are four edge insertions $\{\sigma_1, \sigma_5, \sigma_9, \sigma_3\}$ that are located at segment s_0, but there are only one empty position in s_0. $\sigma_5 = \overrightarrow{v_1 v_4}$ updates the weight of the original edge $\overrightarrow{v_1, v_4}$ in segment s_0, but the other three operations are new edge insertions. Therefore, GPMA+ rolls the updates up to the above level (i.e., $layer_1$ $[s_0, s_1]$), including segments s_0 to s_1 in the second round to probe empty space. Unfortunately, the second round also fails. Note that $layer_1$, $layer_2$ and $layer_3$ are index ranges that includes consecutive segments, which are not physical storage levels. Only the leaf layer is a physical sorted array. In this example, even GPMA+ reaches the original root layer ($layer_2$) $[s_0, s_3]$ (including all segments in the ordinal array), there are no enough space to accommodate all insert edges.

Thus, GPMA+ triggers the expansion that doubles the original sorted array. Finally, GPMA+ merges all edges in the original array and inserted edges; also re-balances these edges into eight segments in the expanded array (see Figure 4). Since the re-balance involves all segments, we call it *global re-balancing*.

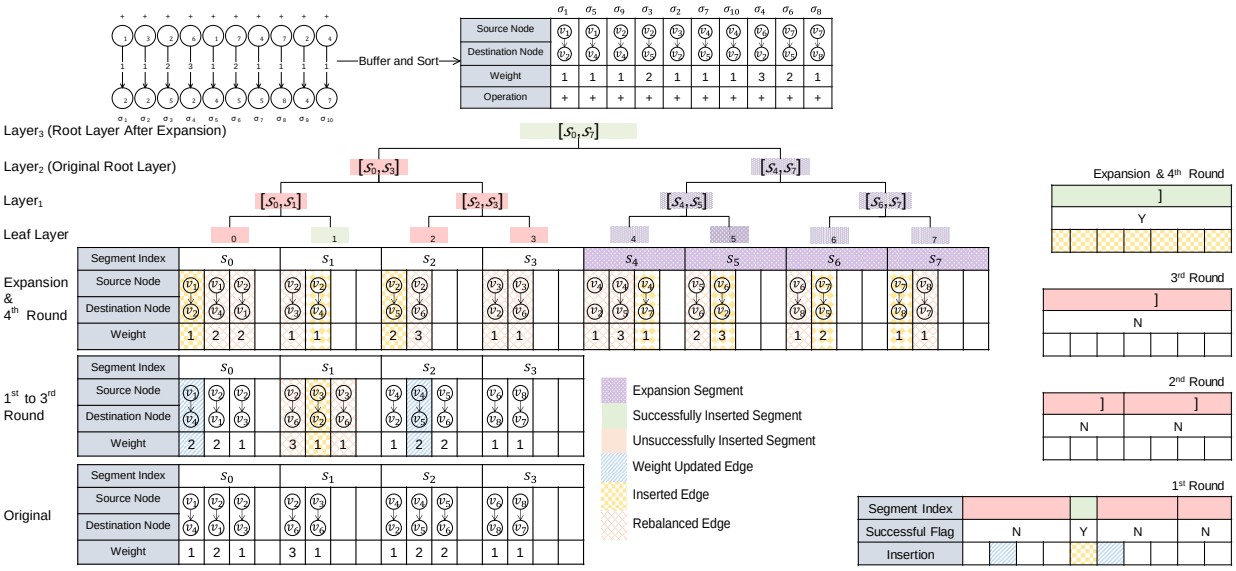


Fig. 4: GPMA+ Update and Expansion

Analogously, GPMA+ handle the deletion as a dual process of insertion.

3 Leveled Packed Memory Array

3.1 Expansion in GPMA+ and Unnecessary Re-balance

GPMA+ aims to maintain a continuous sorted array with public reserve space for future insertions. After re-balancing, new edges could be inserted into the reserve space. During the re-balancing, edges are sorted by nodes IDs. However, if new insert edges overflow the original array, GPMA+ will trigger the expansion. Let us recall Figure 4. After the third round insertion, each segment reaches the density threshold. GPMA+ doubles the sorted array from four leaf layer segments to eight leaf layer segments, and then inserts and re-balances edges to complete this update.

The expansion operation in GPMA+ is costly in two ways. Firstly, the whole array resize and re-balance requires massive memory access and reallocation. Since GPMA+ is stored in a continuous memory space, the system needs to allocate a double size memory space, re-balance the whole array, write into the new space and release the original one. The reallocation process could be costly when the edges number grows large. Furthermore, since GPMA+ always allocates new continuous memory space to hold all edges and release the original continuous memory space, this kind of allocation leads to a lot of potential memory fragmentation. Secondly, unnecessary data movements are carried out during the re-balance. In each expansion & re-balance phase, GPMA+ collects all edges in the original array and merges them with insert edges. We call it *global re-balancing*. Finally, these edges are distributed to the whole new double-size array to balance the fragment density. Obviously, this is a global operation with $O(|E|)$ complexity and many unnecessary segments also need to participate. The re-balancing operation in the unnecessary segments is defined as the *Unnecessary Re-balance*.

3.2 LPMA Structure and Update

In order to reduce the unnecessary re-balances and the expansion overhead in GPMA+, we propose a leveled structure LPMA to organize edge segments. Different from GPMA+, LPMA partitions edge segments into different levels. Except for the head level $level_0$, each x-th $level_i$ ($x = 1, ...n$) contains 2^{x-1} leaf layer segments, where segments in the same level are stored in a continuous memory space, but we do not store all levels consecutively. Logically, segments in each level except for $level_0$ (in LPMA) form a *perfect binary tree*. The dash lines in Figure 5 show the tree's structure. Since LPMA is a perfect binary tree, we do not store the dash line (parent-child relation) physically in LPMA. The in-order traversal of the binary tree (LPMA) forms a sorted edge array, which corresponds to GPMA+. Note that head level $level_0$ has a single segment that precedes all other segments. In a word, segments in LPMA form a sorted edge array through the in-order traversal, but it has different physical storage scheme from GPMA+.

Since LPMA is a perfect binary tree, it is easy to map the y-th segment in the x-th level (i.e., $level_x$) to the i-th segment in the sorted array and vice versa. We also call i as the sequential index of the segment. The following equations illustrates the process (m is the greatest level number of LPMA and d is the greatest common divisor of i and 2^m):

$$i = \begin{cases} 0 & x = 0 \\ (2y+1) \times 2^{m-x+1} & m \geq x \geq 1 \end{cases} \tag{1}$$

$$(x, y) = \begin{cases} (0, 0) & i = 0 \\ (m - \log d, \dfrac{i/d - 1}{2}) & i > 0 \end{cases} \tag{2}$$

Note that LPMA adopts the similar segment-oriented *bottom-up* update procedure as GPMA+. Specifically, when the update batch arrives, LPMA first conducts binary search to locate the associated leaf layer segments for insert edges. The system performs the segment oriented parallel insertion on the logical sorted array that is same with GPMA+. If the corresponding segment cannot accommodate insert edges, we will roll up the insertions. Figure 5 shows an update example of LPMA. Initially, LPMA has four leaf layer segments. When edge insertion batch comes, the system sorts edges according to edge endpoint IDs and conducts binary

search over LPMA to identify the associated leaf layer segments. In the first round, new edges σ_5 and σ_7 updates corresponding original edges $\overrightarrow{v_1v_4}$ and $\overrightarrow{v_4v_5}$, respectively. Edge $\sigma_2 = \overrightarrow{v_3v_2}$ is successfully inserted into leaf layer segment s_2. Other edge insertions fail due to insufficient space in the associated segments. All the updates and insertions are executed parallel. Then, we probe larger segment ranges (2^1 and 2^2 segments) in the second and third round. Finally, there are still seven edges that cannot be inserted into the whole sorted array (LPMA) and they trigger expansion in LPMA.

LPMA is designed to address the expansion performance issue in GPMA+, thus, it has different expansion process. At expansion, LPMA only needs to allocate a new level and append it to the current tree. Figure 5 illustrates an expansion example, i.e., appending new $level_3$. After expansion, LPMA performs *localized re-balance* that does not always involve all segments. Since LPMA is a perfect binary tree, appending one new level, each segment (in the original LPMA tree) has an empty successor segment (shown in purple colour in Figure 5) in the in-order traversal. Logically, after expansion, we append one empty segment after each original segment to obtain an extended LPMA. Figure 5 visualizes the sorted segments in the expanded LPMA. However, GPMA+ appends all new empty segments to the original sorted array (see Figure 4).

In Figure 5, there is one new empty segment s_4 between s_0 and s_2. Before expansion, seven edges still fail to be inserted. Edges σ_1,σ_9 and σ_3 should be inserted into segment s_0, but, there are insufficient spaces to hold three edges in segment s_0. After expansion, s_0 has a new empty successor segment s_4. Therefore, we can re-balance six edges between s_0 and s_4, including three edges in the original s_0 and three insert edges. The re-balancing only happens between two consecutive segments, thus, we call it *localized re-balance*. The same happens over segment pairs $\{s_1, s_6\}$ and $\{s_3, s_7\}$ to handle edge insertions. In this example, segments s_2 or s_5 does not participate in re-balancing, which is different from global re-balance in GPMA+. Of course, if two consecutive segments cannot accommodate insert edges, we also need to roll up the insertion and check four consecutive segments. Although in the worst case all segments of LPMA participate in the re-balance process, which degrades to global re-balance in GPMA+, we theoretically prove that the worst case rarely happens and our method (LPMA) reduces the expected number of re-balancing segments significantly (see Lemma 1 in Section 3.3). Experiment results also confirm our analysis (see Section 7.1).

Algorithm 1 (in Appendix A [2]) shows the bottom-up re-balance of LPMA. LPMA first sorts the update batch and runs the binary search to locate the leaf layer segments S for the updates (Lines 1-3 in Algorithm 1). Then LPMA filters out duplicated segments in S to get S^* (Line 6). For each segment in S^*, LPMA calculates the densities and checks if the segment has enough room (Lines 7-11). If the segment has enough room for the updates, LPMA merges the updates with the items in the segment and re-dispatch the merged items into the segment evenly (Line 11-14). We remove all successfully inserted edges from U (Line 13) and the corresponding inserted segment s from S^* (Line 14). Note that a thread or a warp (depends on the size of the segment) is called to handle one segment update and all segment updates are processed in parallel (Lines 7-14). After the first round, if $|U| = \phi$ (no insert edges left), we finish the whole process. Otherwise, we continue the next round. If the current layer is the root, it means that the current LPMA cannot accommodate edge insertions in U. So, we expand the current LPMA by appending one new layer (Line 19) and re-locate the leaf layer segments S (in the augmented LPMA) for the remaining inserted edges (Line 20). Otherwise, we can roll up the updates to one upper layer (Lines 23-25).

Query Processing. Since LPMA maintains a sorted edge array, it is easy to answer query primitives (edge query and 1-hop successor/predecessor queries in Definition 4) using the binary search. Other graph queries can be decomposed into a series of query primitives over LPMA. Furthermore, to employ existing graph analysis library, we can also convert the sorted edge array in LPMA (corresponds to a graph snapshot) into CSR representation, since most GPU graph libraries are based on CSR.

3.3 Analysis

In this subsection, we analyze the memory efficiency and expansion & re-balancing cost in LPMA.

3.3.1 Memory Efficiency

Leveled memory allocation in LPMA does not affect memory access. Since GPU has smaller cache and supports parallel memory access, we can assign different warps to access different levels of LPMA in parallel. The non-continuous memory allocation is not the bottleneck as long as each segment has 128-byte continuous space for one warp access. That is different from CPU architecture due to larger cache size and serial memory access. Furthermore, as discussed before, LPMA alleviates memory allocation cost and memory fragmentation issue. Last but not least, the leveled structure enables shared memory to cache top levels to accelerate memory access.

3.3.2 Localized VS. Global re-balancing

Both LPMA and GPMA+ employ the same update strategy on the sorted edge arrays, although LPMA maintains a sorted array using a perfect binary tree while GPMA+ stores a sorted array physically. They have the exactly same condition to trigger expansion. The only difference lies in the re-balancing process. To quantify the cost of data movement during re-balancing, we introduce the following the definition.

Definition 6 (Re-balancing Size)**.** *Re-balancing size is the number of segments involved in the re-balancing process after expansion.*

Let us recall the running example in Figure 4 and 5. Given the same insertions, after expansion, all eight fragments participate in re-balancing in GPMA+. However, in LPMA, segment s_2 or s_5 is not involved in re-balancing. Actually, according to localized and global re-balancing strategies in LPMA and GPMA+, it is easy to conclude the following claim.

Claim 1. *Given the same edge insertion batch, the re-balancing size in LPMA is always smaller than that in GPMA+.*

To better quantify the benefit of LPMA, we will make the following analysis. Let s be the number of segments in the original array (before expansion). x is the number of insert edges in a batch and n is the maximum number of edges in each segment. According to LPMA insertion strategy, given an insert edge σ, we first locate the associated segment in the original array. For the simplicity of analysis, we assume that edges are randomly

2. Due to space limit, we provide all pseudo codes in Appendix of the supplementary materials.

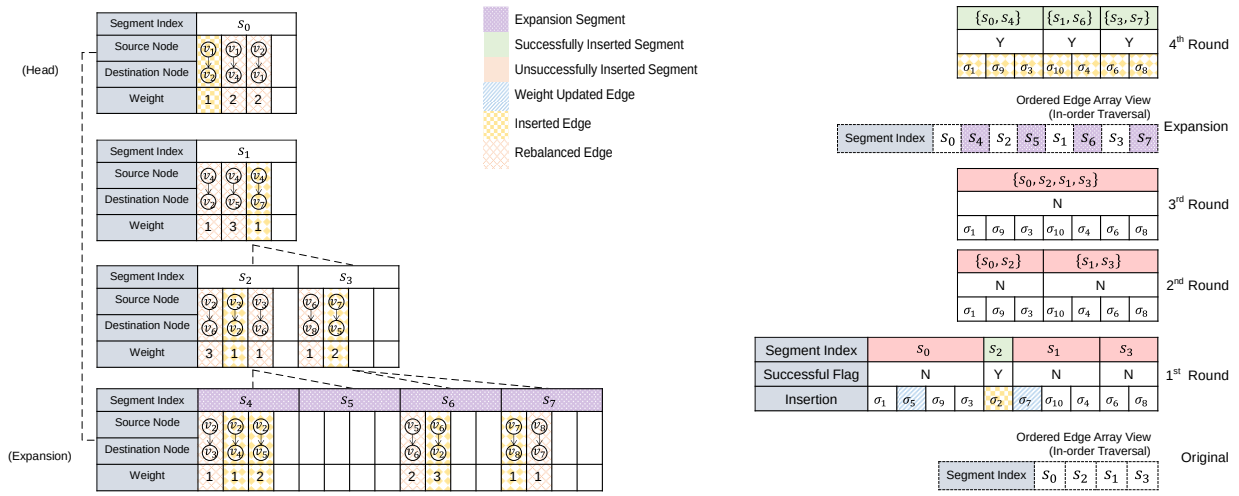


Fig. 5: LPMA Update and Expansion

inserted. We call all segments involved in the re-balancing process as *associated segments* and others are called *irrelevant segments*. The following lemma states the number of associated segment and irrelevant segments.

Lemma 1. *The expected numbers of irrelevant segments and associated segments are* $s \cdot (1 - \frac{1}{s})^x$ *and* $s - s \cdot (1 - \frac{1}{s})^x$, *respectively.* [3]

According to LPMA expansion and re-balancing process, only associated segments and their successor empty segments (in the new expanded layer of LPMA) participate in the re-balancing process. However, if an associated segment and its successor empty segment cannot accommodate the corresponding edge insertions, we may need to roll up the insertion and consider the consecutive four segments. Fortunately, the rolling up rarely happens in practice. Considering typical parameters in our experiments, there are $x = 10^5$ insert edges in a batch and $s = 10^6$ segments in the original LPMA. Each fragment can accommodate at most $n = 16$ edges, which corresponds to 128-byte continuous space for one warp access. Consider an associated segment a and its successor empty segment a'. We need to trigger rolling up process only when there are at least $(n + 1) = 17$ edges to be inserted into a; otherwise, a and a' can hold all insert edges. The number of edges to be inserted into segment a (denoted as y) can be modeled as a binomial distribution $y \sim B(x, \frac{1}{s})$. According to typical parameters in experiments, $y \sim B(10^5, 10^{-6})$. Thus, the probability of non-rolling up is calculated as a cumulative distribution $P(y \leq 17) = \sum_{y=0}^{17} C_{10^5}^{y} (\frac{1}{10^6})^y (1 - \frac{1}{10^6})^{10^5 - y} > 99\%$[4]. In other words, the rolling up rarely happens and most re-balancing is conducted between two consecutive segments (i.e., an associated segment a and its successor empty segment a').

Lemma 1 considers the edge insertion as a hashing process. The x insertions are inserted into s segments and $2s - s \cdot (1 - \frac{1}{s})^x$ segments are expected to be re-balanced with the insertions in LPMA. According to the typical parameters in experiments ($s = 10^6$ and $x = 10^5$), the proportion of irrelevant segments is $(1 - \frac{1}{s})^x \approx e^{\frac{x}{s}} \geq 90\%$. Since the rolling up rarely happens, the expected re-balancing size in LPMA is $2s(1 - \cdot(1 - \frac{1}{s})^x) \leq 2s \times 10\%$. Due to global re-balance in GPMA+, GPMA+'s re-balancing size is always $2s$. Therefore, LPMA saves more than 90% cost.

3. The proof based on probability analysis [18] is given in Appendix E.

4. For the ease of calculation, we can use a normal distribution to simulate the binomial distribution calculation when x is large enough

4 Redundancy-Free Re-balance

4.1 Motivation: Redundant Re-balance

The leveled structure of LPMA has addressed unnecessary re-balance cost during the expansion process. However, given a batch of inserted edges, some re-balances may be redundant during the non-expansion re-balancing. Let us continue the example in Figure 5. After expansion, LPMA has eight consecutive segments $[s_0, s_4, s_2, s_5, s_1, s_6, s_3, s_7]$. This is called as the top layer range. Given another batch of 6 new edge insertions (see Figure 2d), in the first round, LPMA locates new edges $\{\sigma_{11}, \sigma_{12}, \sigma_{14}\}$ into segment s_0 and the insertion fails due to lack of room. The same happen for $\{\sigma_{16}, \sigma_{17}\}$ in segment s_2. Only edge σ_{19} is inserted into segment s_5 successfully. In the first round, re-balance only happens in each segment itself. We call this leaf layer. In the second round, we consider 2 consecutive segments to re-balance, i.e., LPMA will try segments $[s_0, s_4]$ and $[s_2, s_5]$, respectively. New edges $\{\sigma_{16}, \sigma_{17}\}$ are re-balanced into the segment $[s_2, s_5]$ successfully, but edges $\{\sigma_{11}, \sigma_{12}, \sigma_{14}\}$ fail again. Thus, LPMA has to roll up the upper layer, i.e, considering size-4 segment range $[s_0, s_4, s_2, s_5]$ to re-balance. In this case, $\{\sigma_{11}, \sigma_{12}, \sigma_{14}\}$ are re-balanced into the segment $[s_0, s_4, s_2, s_5]$ successfully. In this example, the edge insertion and re-balance in segment s_5 are redundant in the first and second rounds, since all edges need to be re-balanced in segment range $[s_0, s_4, s_2, s_5]$ as a whole in the third round. Obviously, a more desirable strategy is to skip the first two rounds and save more redundant re-balance costs.

4.2 Top-Down Re-balance Strategy

The above redundant re-balance problem lies in the bottom-up probing strategy for edge insertion, while both GPMA and LPMA adopt this kind of strategy. To address that in LPMA, we propose a redundancy-free re-balance scheme, i.e., the top-down scheme (called LPMA-T). To differentiate, the original LPMA with the bottom-up re-balance strategy is named as LPMA-B. Assume that LPMA has h levels excluding the head level l_0. There are 2^{h+1} consecutive segments in total. We call this top layer range. A three level LPMA example is given in Figure 5. Thus, the top layer range has 8 consecutive segments. Given a batch of edge insertions U, LPMA-T first initiates the process from the top layer, i.e., all

consecutive segments in LPMA. If we have no enough room to accommodate all inserted edges in a batch, we need to trigger an expansion process. Otherwise, we iterate the process on both the left and the right half of the top layer, i.e., drilling down the lower layer. Both the left and the right parts have 2^h consecutive segments, respectively. Specifically, the edge insertion batch U is decomposed into two groups U_l and U_r that are located at the left and right parts of the top layer. If neither of the left nor the right part can accommodate edges in U_l or U_r, it means that we have to re-balance all edges in the top layer and all re-balances in the lower layer are redundant. If both the left and the right ranges have enough room for inserted edges in both U_l and U_r , it means that it is unnecessary to re-balance at the top layer. We can further iterate drilling down the probing process to the lowest layer that can accommodate all edges. In this way, we can avoid all redundant re-balances. Algorithm 2 shows the pseudo code of the top-down strategy.

We use the same example in Figure 2d. When the system receives the update batch, the system first runs the density check and determines if the expansion is needed (Lines 3-4 in Algorithm 2). If the expansion is needed, we append one new layer to LPMA (Line 6). Then, we perform the top-down re-balance from the root by calling function `TopDownRebalance` recursively (Line 7-8). Initially, the top-down strategy starts from the root layer and splits the root layer into left part s_l ($s_l : [s_0, s_4, s_2, s_5]$ in the example in Figure 2d) and right part s_r($s_r : [s_1, s_6, s_3, s_7]$) (Lines 14-15). Then, the system decomposes the updates into the left part $U_l : \{\sigma_{11}, \sigma_{12}, \sigma_{14}, \sigma_{16}, \sigma_{17}, \sigma_{19}\}$ and right part $U_r : \{\emptyset\}$ and count the number of items in both left and right parts (Lines 16-19). We check if at least one of two parts fail to hold inserted edges (Line 20). If so, we re-balance these inserted edges in this layer (Line 21). If both parts can accommodate inserted edges, we iterate the above process on these two parts (Lines 24-27) until leaf segments (Lines 11-13).

In this layer, both parts have enough room. If $|U_l| > 0$ and s_l is in the non-leaf layer, LPMA-T will iterate the above process over s_l using updates U_l (Lines 24-25). The same applies for the right counterparts U_r and s_r (Lines 26-27). In this example, since U_l is not empty and s_l is not leaf, the function `TopDownRebalance` is called over U_l and s_l into the next round. The right part update is finished in this round since U_r is empty. In the second round, we run the same procedure until the available room check. This time the left part $s_l : [s_0, s_4]$ has 2 available cells but $U_l : \{\sigma_{11}, \sigma_{12}, \sigma_{14}\}$ has 3 insertions. The function `Rebalance` is called in this round. The six insertions are merged into $[s_0, s_4, s_2, s_5]$ and the update is finished. Compare to the bottom-up strategy, the re-balancing of each segment is only carried out once and no redundant re-balance occurs.

5 Hybrid Strategy

The top-down strategy (LPMA-T) guarantees redundancy-free re-balances during the update. However, if the density of structure is low and most re-balances are proceeded on the lower layers, the top-down strategy reduces few redundant re-balances but brings extra cost. For example, if all the re-blances are carried out on the leaf layer, there is no redundant re-balance to be saved. However, the top-down strategy needs to probe all the layers. The overhead of rolling down from root to leaf layer could be larger than the redundant re-balances saved by the top-down strategy.

Apparently, if most re-balances are carried out on low layers, the bottom-up strategy is more efficient; if most re-balances happen on the high layers, the top-down strategy is more efficient. As the density increases, the updates need to access high layers to find enough room. Figure 6 shows the latency curves of the bottom-up (LPMA-B) and top-down strategy (LPMA-T) on one real life dataset (Pokec, more details are given in Section 7). When the density reaches a certain point, the performance of top-down strategy starts to surpass the bottom-up one. Therefore, we introduce a density threshold π. When the new edges arrive, LPMA-H (the hybrid strategy) first checks the density of the structure. If the density is 1, it means that LPMA is full and needs expansion. If the density is higher than π, it means that density of LPMA is high and the top-down strategy is activated. If the density is lower or equal to π, it means that density of LPMA is low and the bottom to top re-balancing is activated.

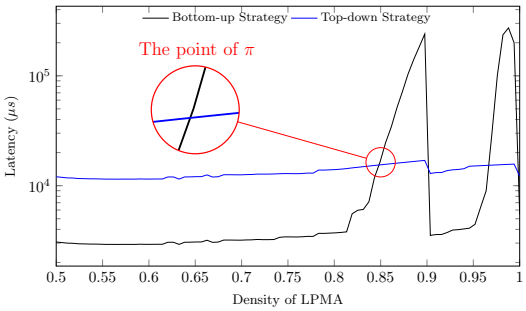


Fig. 6: Latency Curves of Bottom-up Strategy and Top-down Strategy

However, it is difficult to predict the value of π for dynamic graphs. In practice, we propose a *self-adaptive strategy* to find the desirable density threshold π. We define the time period between the two expansions as one cycle. After the last expansion, the density is low. The bottom-up strategy is more desirable. Therefore, we set π be a small empiric value (=0.6) and start to run the top-down strategy. When the density is larger than π, we perform the top-down strategy and the bottom-up strategy in a round-robin manner. When the system probes that the performance of top-down strategy starts to surpass the bottom-up strategy, the π is set to the current density of LPMA and the top-down strategy is carried out until the end of this cycle. Algorithm 3 [5] gives the pseudo codes.

6 Operation Primitive Oriented Parallel Strategy

In real world applications, the updates and queries are received and packed in one batch on the CPU side and send to the GPU side via PCIe. Figure 7 shows the mixed batch model adopted in this paper. Two mixed batches of updates and queries are given in Figure 2d. For example, Operation Stream 2 (Figure 2d) contains 6 edge updates and 4 queries. q_{13} is the 1-hop successor query of v_1; q_{15} is the 2-hop successor query of v_1; q_{18} is the edge query of $\overrightarrow{v_4v_8}$; q_{20} is the edge query of $\overrightarrow{v_4v_7}$.

All queries and updates in a batch are sorted by the corresponding timestamps. Obviously, a naive solution is to perform these queries and updates chronologically in a serial processing model; but that cannot make fully use of the massively parallel processing capability of GPU. However, the batch-oriented parallel processing can enhance the parallel power, but it may cause the time

5. The pseudo codes are given in Appendix A.

inconsistency problem. For example, in Figure 2d, if we execute Operation Stream 2 serially, the answer of q_{13} is $\{v_2, v_3, v_4, v_5\}$. The edge σ_{14} is received after t_{13} and should not be included in the result of q_{13}. If we split Operation Stream 2 into the update group $(\sigma_{11}, \sigma_{12}, \sigma_{14}, \sigma_{16}, \sigma_{17}, \sigma_{19})$ and the query group $(q_{13}, q_{15}, q_{18}, q_{20})$, we could process the two groups parallel and have better performance. However, if we run the update group first, the answer of q_{13} is $\{v_2, v_3, v_4, v_5, v_6\}$; if we run the query group first, the answer of q_{13} is $\{v_2, v_4\}$. Both are incorrect.

In the context of streaming graphs, we propose the following *streaming consistence*. Based on that, we propose our parallel solution for batch-oriented parallel processing in Section 6.1.

Definition 7 (Streaming Consistence)**.** *Given a time-evolving sequence of operations* $O = \{\sigma_1, \sigma_2, ...\sigma_x\}$[6] *over streaming graph* $\mathcal{G}$*, the* streaming consistency *requires the results of parallel processing of* O *are the same as the results formed by executing updates and queries in the chronological order.*

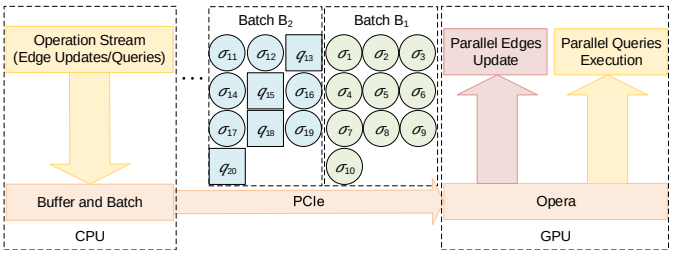


Fig. 7: Mixed Batch of the Updates and Queries

6.1 Operation Primitive Oriented Parallel Strategy

To have the better parallel performance and meet the *streaming consistence* requirement, we propose the *O*peration *P*rimitive Ori*e*nted Par*a*llel Str*a*tegy (abbreviated as *Opera*). Given a batch of operations B, we decompose B into two groups:B_U includes all edge insertions/deletions and B_Q consists all queries, and then process B_U followed by B_Q.

Processing edge updates in B_U. Although we propose a series of parallel solutions to conduct all edge insertions/deletions in Sections 3-Section 5, considering the streaming consistence in mixed workloads, we propose the following modification. Let us recall Figure 5 and all edges are sorted by the source and destination nodes. Now, we introduce another *timestamp* dimension. Assume that the earliest query timestamp in batch B_Q is t_{min}. Given an edge insertion/deletion σ_i in B_U, if the timestamp of σ_i ($t(\sigma_i) < t_{min}$), we have the same solution with Sections 3-Section 5. Otherwise, if $t(\sigma_i) > t_{min}$ and the inserted/deleted edge has been in LPMA before, we do not merge the inserted/deleted edges with the one already in LPMA and append it as a new element to the old ones with identical source and destination nodes. We will merge them after finishing the mixed batch.

Processing queries in B_Q. For simplicity, we assume that all queries in B_Q are primitives, i.e., edge queries and 1-hop successor/predecessor queries. As we know, segments in LPMA form a sorted edge array through the in-order traversal. Thus, we can run binary search to locate the query results for edge query and 1-hop successor queries[7] in parallel. However, some results may be invalid if considering *streaming consistence*. Thus, each query q_i will filter out unreasonable answers based on timestamps and merge edges with the identical source and destination nodes.

If q_i is not a query primitive (such as 2-hop query q_{15} in Figure 2d), we need to decompose it into a series of query primitives. For example, q_{15} could be decomposed into 2 rounds of successor query. In the first round, q_{15} is a 1-hop successor query, which can be combined with other query primitives for parallel processing. The second round of q_{15} is also a set of 1-hop successor queries based on the answers of the first round. They can also be evaluated in parallel. Note that the parallel query processing not only leverages the massively parallel processing capability of GPU but also leads to coalesce memory accessing.

Merge Updates. When finishing the whole mixed batch B, we launch a compaction process to merge edges with identical source and destination nodes but different timestamps. The timestamp of the merged edge is set the last one among all edges to be merged.

6. The operations may include both edge insertion/deletions and query primitives, see Definition 7

7. Predecessor query is the same operation as successor query on LPMA of in-neighbors.

TABLE 2: Datesets

Dataset	Nodes	Edges	Average Degree
Orkut	3,072,441	117,185,083	38.14
Graph500	1,048,576	125,829,120	120
LiveJournal	4,847,571	68,993,773	14.23
Pokec	1,632,803	30,622,564	18.75
cit-Patents	3,774,768	16,518,948	4.38
Road	1,379,917	1,921,660	1.39
Stack	2,601,977	36,233,450	13.93
Wiki	1,140,149	3,309,592	2.9

7 Experimental Evaluation

Datasets: Table 2 gives statistics of datasets in our experiments, which are from SNAP [19].

- **Orkut** is a free on-line social network where users form friendship between each other.
- **Graph500** is a synthetic dataset generated by Graph500 RMAT [20] to synthesize a power law graph.
- **LiveJournal** is a free on-line community with about 10 million members, which contains the activities among the members.
- **Pokec** is the most popular on-line social network in Slovakia. The dataset contains 1.6 million users and their friendship connection.
- **cit-Patents** is a U.S. patent citation network includes all citations made by patents granted between 1975 and 1999, totaling 16,522,438 citations.
- **Road** is a road network of Texas, in which intersections and endpoints are represented by nodes, and roads connecting these intersections or endpoints are represented by undirected edges.
- **Stack** is a temporal network of interactions on the stack exchange web site Stack Overflow. The graph has 63,497,050 temporal edges associated with timestamps and 36,233,450 unique edges in total.
- **Wiki** is a temporal network representing Wikipedia users editing each other's Talk page. The graph has 7,833,140 temporal edges associated with timestamps and 3,309,592 unique edges in total.

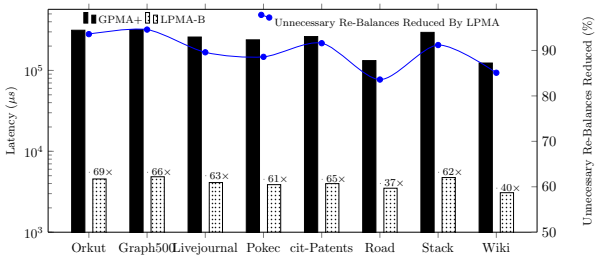
For the first six graph datasets, we shuffle edges in a random order to assign the timestamps to simulate dynamic graphs. Both Stack and Wiki use the original associated edge timestamps. In

experiments, we load these edges incrementally in batches to build dynamic graph structures.

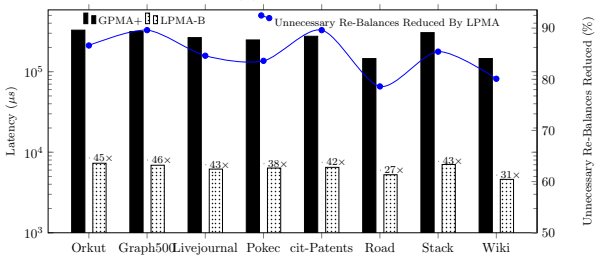
Setup: We experimentally compare LPMA with state-of-art GPU dynamic graph data structures, including GPMA+ [8], faimGraph (sort version) [11], Hornet [9] and HashBased [10]. We obtain their codes from downloadable URL[8]. All experiments are implemented with CUDA 7.5 and GCC 4.8.5 and run on Red Hat 4.8.5 server that has Intel(R) Xeon(R) E5-2640 (6-cores, 2.60GHz) with 128GB main memory and NVIDIA Tesla P100 GPU with 16GB main memory.

7.1 LPMA-B VS. GPMA+ in Expansion Performance

LPMA is designed to address the expansion cost in GPMA+. So, we first evaluate the expansion cost in both LPMA and GPMA+. In this subsection, we adopt the bottom-up update strategy in LPMA (Section 3.2), denoted as LPMA-B. During the insertion, some batches would trigger the expansion and we denote them as *expansion batches*. We have the same parameter settings for both LPMA-B and GPMA+, including the same density thresholds and the same segment sizes. Thus, both structures have the same time point to trigger the expansion. In this experiment, we compare LPMA-B with GPMA+ in *expansion batches*. We evaluate the average update time of one expansion batch in both LPMA-B and GPMA+ in Figure 8.



(a) 10^4 Batch



(b) 10^5 Batch

Fig. 8: Expansion Performance and Unnecessary Re-balances Reduced by LPMA (only Expansion Batch)

When a batch contains 10^4 edge insertions, LPMA-B has 69-61× speedups over GPMA+ on the first four datasets, 37× speedups on Road, 62× speedups on Stack and 40× speedups on Wiki. As we analyzed in Section 3.3, GPMA+ always needs to re-balance the whole array during the expansion. The expansion cost depends on the size of the array. The array size grows quite large in the last few rounds on Pokec and cit-Patents, thus the expansion costs increase rapidly. However, LPMA-B performs the localized re-balance. The quantitative analysis in Section 3.3 states that LPMA-B can reduce most re-balance cost in GPMA+. Figure 8 also confirms that more than 80% re-balance costs can be saved in LPMA-B in 10^4 batch.

When the batch size increases to 10^5 edges, the LPMA has 46-38–speedups over GPMA+ on the first four datasets, 27× speedups on Road, 43× speedups on Stack and 31× speedups on Wiki in Figure 8. Note that Figure 8 only considers *expansion batches*. The advantage of LPMA weakens with the increment of batch size during expansion batches, since the expected numbers of non-associated segments (Lemma 1) is reduced. Actually, the overall update performance of LPMA increases with regard to larger batch sizes in Figure 11, as analyzed in Section 7.2.

7.2 Evaluating three Re-balance Strategies

In this subsection, we evaluate our proposed three re-balance strategies of LPMA: bottom-up (LPMA-B), top-down (LPMA-T) and hybrid (LPMA-H). Figure 9 shows the latency curves of the three strategies on Pokec. With the arrivals of batches, the density of LPMA grows until the expansion. LPMA-B has the good performance when the density is low but has up to 83× latency spikes when the density is high. LPMA-T has a more flat latency curve and the spikes are flattened because the redundant re-balances are reduced by the top-down strategy, but it has extra costs to roll down from the root layer to the leaf layer when the density is low. In most batches, LPMA-H follows the better performance between LPMA-B and LPMA-T except the probe rounds. When the density reaches π, where π is an empiric value (60%), the probe rounds bring some fluctuations to the latency curve of LPMA-H, but Figure 10 shows that LPMA-H has better average update performance than LPMA-B and LPMA-T.

When a batch contains 10^4 edge insertions, LPMA-T saves 21–27% re-balances on LPMA-B (see Figure 10). The reduced re-balances are considered as redundant re-balances that are analyzed in Section 4. However, the extra costs to roll down from the root layer to the leaf layer bring down the average performance of LPMA-T. In the eight datasets, LPMA-B has around 1.3× faster than LPMA-T. LPMA-H saves 15–20% redundant re-balances which are fewer than LPMA-T. Generally, LPMA-H follows the better performance between LPMA-B and LPMA-T in most batches and has the best update performances. In the eight datasets, LPMA-H has around 2× speedups over LPMA-T and 1.4× speedups over LPMA-B. When a batch contains 10^5 edge insertions, LPMA-T could save 33–37% re-balances on LPMA-B because the larger batch size causes the larger increment speed of density. In the eight datasets, LPMA-B has around 1.5× speedups over LPMA-T. LPMA-H saves 24–29% redundant re-balances and has around 2.5× speedups over LPMA-B.

7.3 Overall Edge Update Performance

We compare the average latency for all edge update batches (including expansion batches) among LPMA-H, GPMA+ [8], faimGraph (sort version) [11], Hornet [9] and HashBased [10] in Figure 13. Note that we also experimentally evaluate node updates in Appendix B.

LPMA-H with different batch sizes: Figure 11 shows that the throughput of LPMA-H increases significantly varying the batch size from 10^3 to 10^6. For every 10× the batch size, the throughput increases by 5–9×. The acceleration effect caused by the larger batch size is due to two reasons: the lower average transfer cost and the higher parallel power. The larger batch size makes full

8. GPMA+:https://github.com/desert0616/gpma_demo
Hornet:https://github.com/hornet-gt/hornet
faimGraph:https://github.com/GPUPeople/faimGraph
HashBased:https://github.com/gunrock/gunrock/tree/dynamic-graph

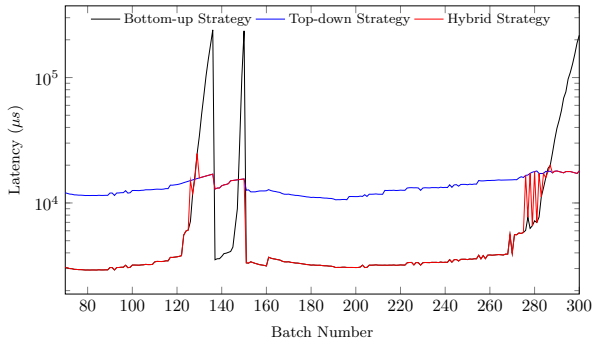
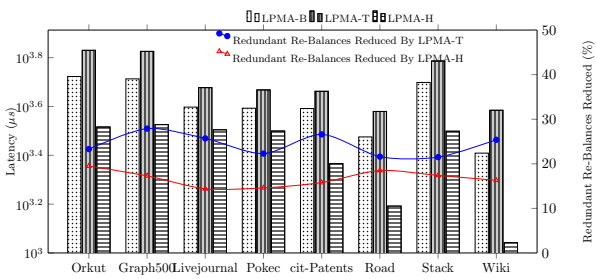
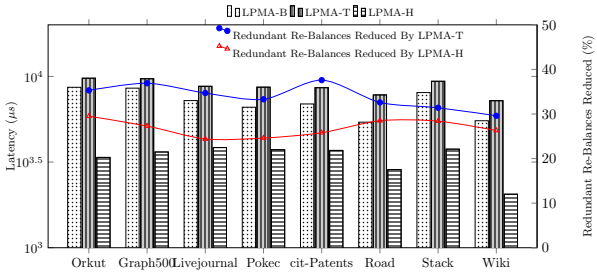


Fig. 9: Latency Curves of 3 Strategies on Pokec



(a) 10^4 Batch



(b) 10^5 Batch

Fig. 10: Insertion Performances and Redundant Re-balances Reduced by LPMA-T and LPMA-H

use of PCIe bandwidth and the higher parallel power of GPU. We also show the data transfer and synchronization time in Figure 12, which show that the data transfer and synchronization spends 22–38% of total update time. On the other hand, the larger batch size causes longer latency due to more buffer time on the CPU side. Therefore, we choose 10^4 and 10^5 batch size in our experiments.

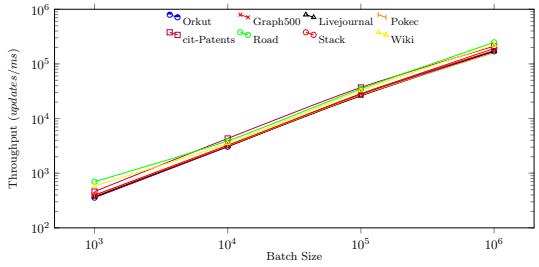


Fig. 11: Insertion Performance of LPMA-H in Different Batch Sizes

Comparing with GPMA+: As discussed in Section 3.3 and 4, LPMA-H reduces unnecessary re-balances in the expansion batches and redundant re-balances in the non-expansion batches.

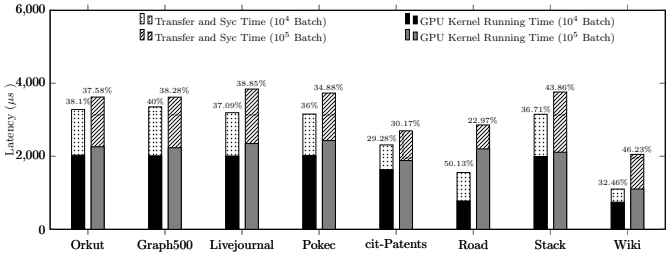
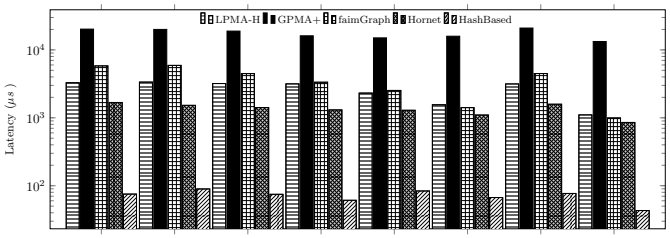
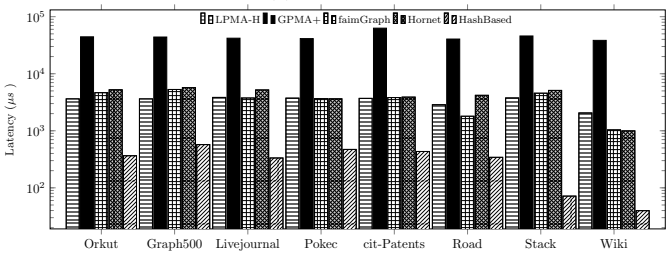


Fig. 12: Data Transfer and Synchronization Cost and GPU Kernel Running Time



(a) 10^4 Batch



(b) 10^5 Batch

Fig. 13: Overall Update Performance (All Update Batches)

The two optimizations give LPMA-H a huge advantage on updating. When edges arrive in 10^4 batches, LPMA-H has 6–9× speedups over GPMA+. In 10^5 batches, LPMA-H has 10–20× speedups over GPMA+. Although the advantage of LPMA weakens in large batch sizes during expansion batches, the expansion happens more frequently in larger batches. Therefore, consider the overall update performance, LPMA-H has more speedups over GPMA + in larger size batches.

Comparing with faimGraph: In the 10^4 batch sizes, LPMA-H has 1.4–1.7× speedups over faimGraph on the large graphs: Orkut, Livejournal, Pokec and cit-Patents. On the small graph with even degree distribution, faimGraph has 1.1× speedups over LPMA-H. In the 10^5 batch sizes, LPMA-H has 1.3–1.5× speedups on Orkut and Graph500. On Livejournal, Pokec and cit-Patents, LPMA-H and faimGrah have simlar performanc. On Road, faimGraph has 1.6× speed up.

The performance diversities on different datasets are caused by the different insertion method. In faimGraph, the insertion batches are partitioned by the source nodes and merged into the associated neighbor lists. Each neighbor list is assigned a thread or a warp to process the insertion. In skewed degree distribution graphs (such as Orkut and Graph500), faimGraph has worse performance due to unbalanced workloads. LPMA-H inherits the segment oriented parallel strategy from GPMA+ and the workloads among the threads are always balanced. Thus, LPMA-H has better performance in this case. However, in Road graphs with small and even

degrees, faimGraph is better than LPMA, especially in large batch sizes.

Comparing with Hornet: Since Hornet does not sort the neighbor lists, Hornet is faster than LPMA-H, GPMA+ and faimGraph in 10^4 batch. However, Hornet dose not maintain the sorted graph and has worse performance on graph queries (see Section 7.4). In the 10^5 batch sizes, the performance of LPMA-H surpasses Hornet because the scalability of Hornet is worse than LPMA-H. Hornet assigns a continuous array to the neighbor list of each node. If the array is full, Hornet assigns a double size new array to the neighbor list and move the whole list to the new space. The expansion operation of Hornet happens much more frequently than GPMA+ and LPMA-H. The larger batch size brings even more frequent expansion and the massive data movements effect the update performance significantly.

Comparing with HashBased: For each update, all the five dynamic graph system needs to perform the edge existence check to eliminate the duplicated edge records. The hash based neighbor list of HashBased structure accelerates the edge existence check and HashBased outperforms LPMA, Hornet and faimGraph significantly, as shown in Figure 13. However, HashBased does not maintain the order of neighbor lists as LPMA does. Thus, LPMA has much better performance than HashBased on some graph queries, such as sorted CSR converting and triangle counting than HashBased. More details are given in Section 7.4.

7.4 Query Performance

In this subsection, we evaluate the query performance using query-only batches. We consider different query primitives over five structures: LPMA, GPMA+, faimGraph, Hornet and Hashbased as well as the time of converting these dynamic data structures into CSR formats. Since three re-balance strategies of LPMA only effect the update processing and the query performance keep the same, thus, we only denote LPMA in the following experiments.

Query Primitives: Since query primitives are fundamental of many other graph algorithms like BFS and n-hop neighbor, which are executed by performing query primitives iteratively. Thus, we compare LPMA with GPMA+, faimGraph and Hornet in these query primitives: edge query and sorted 1-hop neighbor query. We random generate four query workload batches: 10^4 edge queries, 10^5 edge queries, 10^4 neighbor queries and 10^5 neighbor queries. Figure 17 shows the results of the query primitives performance[9]. Since GPMA+ and LPMA has the same logical structure, they have similar query performance.

For the edge query, LPMA has 12× and 6× speedups over faimGraph in 10^4 and 10^5 batches. To run the edge query parallel, faimGraph assigns one thread for each query edge and runs binary search on the associated neighbor list. LPMA assigns one thread for each query edge and run the binary search on the whole edge array. Although LPMA has larger searching space for each query edge, the shared memory prefetch could reduce the main memory access and speed up the parallel query evidently. For the sorted neighbor list query, LPMA has 1.5× and 1.2× speedups over faimGraph in 10^4 and 10^5 batches. faimGraph assigns one thread for each query node to read the associated neighbor list and the uneven lengths of neighbor lists could cause unbalanced workload. LPMA first uses node oriented strategy to read the offset and locate the associated segments. Then LPMA uses segment oriented strategy to read the neighbor lists parallel that avoid the unbalanced workload.

Since Hornet does not maintain the sorted edges, thus LPMA has 5× speedups on the edge query and 3× speedups on the sorted neighbor query. The experiment results show that edge query on the hash table is faster than the binary search on the sorted edge array. The hash based neighbor list accelerates the update and edge query performance significantly and HashBased has 10× speedups over LPMA on edge query. However LPMA has 5-10× speedups on the sorted neighbor query since LPMA maintains the sorted edge array.

CSR Converting Performance: As we discussed in Section 2.2, CSR is a de facto structure for many existing graph algorithms. To enable use existing graph analysis libraries, we need to convert a dynamic graph structure to CSR format. Thus, we compare the CSR converting time cost for these four data structures.

In experiments, we load eight datasets and convert the dynamic graph structure into the CSR format. GPMA+ and LPMA maintain a sorted edge array. By compacting the empty cells in the array, both GPMA+ and LPMA could be converted into CSR within a small cost. Similar as sorted neighbor list query, faimGraph uses the node oriented strategy to extract CSR and the performance is limited by the unbalanced workload. Hornet does not maintain the ordered graph and needs to be sorted first before the converting. Figure 14 shows that GPMA+ and LPMA has the same performance, 30–7× speedups over Hornet and around 1.5× speedups over faimGraph on most of the datasets. The only exception is that faimGraph has better CSR converting performance than LPMA on the Road dataset, since the road graph has evener degree distribution and the average degree is small, which favors faimGraph that assigns one thread for each query node to read the associated neighbor list.

In codes of HashBased [10], the hash neighbor neighbor list only be converted into *unsorted* CSR on GPU side, i.e., all neighbors of each node are not sorted by the destination node IDs. We show the CSR converting performance of different approaches in Figure 15. Obviously, the unsorted version of CSR converting has the fastest time; but LPMA is faster than converting HashBased to sorted CSR no matter sorting on CPU or GPU. Specifically, the data movement and the sort on CPU side cause 11–20× latency for the sorted CSR converting. LPMA still has 3–8.5× speedups over HashBased for the sorted CSR converting on GPU side.

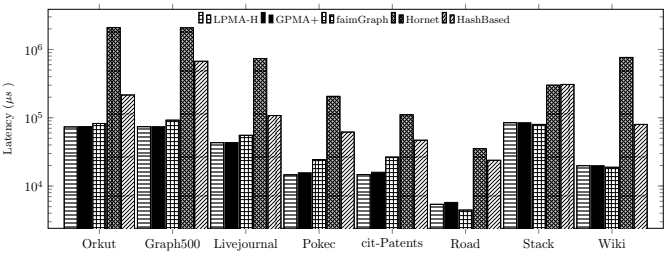


Fig. 14: CSR Converting Performance

Graph Algorithms Performance: We conduct Triangle counting (Figure 18), PageRank (Figure 19) and BFS (Figure 20) algorithms on the five data structures.[10].

LPMA and faimGraph (sorted version) have better performance than Hornet and HashBased on triangle counting algorithm. LPMA has 2.4–5× speedups over Hornet and 4–7× speedups

9. Due to space limit, Figures 17 is given in Appendix D.

10. Due to space limit, Figures 18, 19, 20 are given in Appendix E.

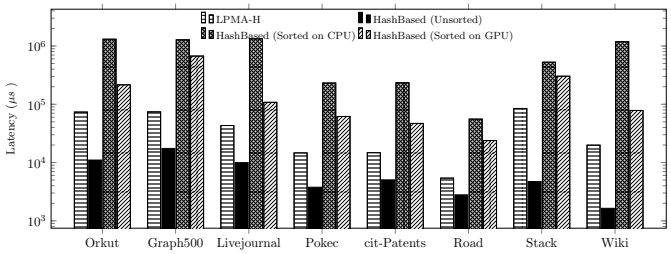


Fig. 15: CSR Converting Performance (LPMA and HashBased)

over HashBased. As the set intersection takes a large proportion in triangle counting [21], the sorted neighbor lists in LPMA and faimGraph (sorted version) can significantly improve the set intersection performance significantly. Therefore LPMA and faimGraph have much better performance on triangle counting algorithm than Hornet and HashBased.

For PageRank, LPMA has 0.9–1.1× speedups over faimGraph, 1.1–1.3× speedups over Hornet and 0.88–1.2× speedups over HashBased. For BFS, LPMA has 0.9–0.92× speedups over faimGraph, 1.1–1.2× speedups over Hornet and 1.1–1.2× speedups over HashBased. Since LPMA and GPMA+ maintain exactly the same sorted edge array used in the algorithms, the algorithms have same time costs on the two structures. Hornet, faimGraph and HashBased use the node-oriented parallel strategy to read the the neighbor lists. LPMA, on the other hand, first computes the segments where the neighbor lists are located and perform the segment-oriented parallel strategy to read the the neighbor lists. Therefore, for graph queries with uneven point degree distribution, LPMA has an advantage. If the degree distribution of points is more even, this advantage is no longer obvious. Hornet has the worst performance since Hornet doesn't have the workload balance optimization.

7.5 Performance of Query-Update Mixed Batches

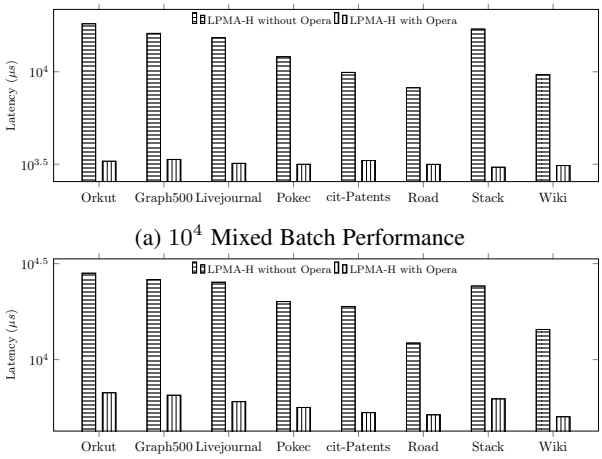
To evaluate our operation primitive oriented parallel strategy (i.e., Opera) over query-update mixed batches, we generate 3% of batch size queries which include edge queries, 1-hop neighbor queries for each batch and mixed with the update edges randomly. We load the datasets in 10^4 batches and 10^5 batches. Each edge and query is assigned a timestamp by the time of buffering by the CPU side. To the best of our knowledge, we are the first to consider parallel processing for mixed batches. Thus, we only compare LPMA-H without Opera with one with Opera.

When we load the batches into LPMA-H without Opera, the update edges are cut into many small groups and the system has to execute the small groups serially. The parallel power of GPU is not fully used without Opera. In LPMA-H with Opera system, the batch is only split into two groups and each group could be processed in parallel. Figure 16 shows the performance of LPMA without Opera and LPMA with Opera. In 10^4 Mixed Batch, Opera has 2–6× speed up. In 10^5 Mixed Batch, Opera has 4–11× speedups.

7.6 Discussion

We summarize experimental results of different dynamic GPU graph data structures and separate the discussion from two perspectives: *graph updates* and *graph query/computing algorithms*.

Graph updates: For edge updates, HashBased has the best performance, since updated edges need to be searched and duplicated



(a) 10^4 Mixed Batch Performance

(b) 10^5 Mixed Batch Performance

Fig. 16: Mixed Batch Performance

check, and the HashBased has better performance than that using binary search. In skewed degree distribution graphs, faimGraph has worse performance than LPMA due to unbalanced workloads. In the graphs with small and even degrees, faimGraph is better than LPMA, especially in large batch sizes.

Considering node updates, Hornet and HashBased use the fixed size array to store vertices and do not support node updates. Instead, LPMA and faimGraph support dynamic node updates. For node insertion, LPMA and faimGraph have similar performance. For node deletion, LPMA has better performance. This is due to the fact that when deleting nodes, the edges associated with the nodes need to be deleted in batches, and LPMA has better performance for node deletions in batches. More details are given in Appendix B of the supplementary material.

Graph query/computing algorithms: For edge query, since HashBased uses the hash table to check edge existence, it has the best performance. For the neighbor query that does not require sorting, LPMA does not have advantage. However, since both LPMA and faimGraph can maintain an sorted list of neighbors, LPMA and faimGraph have better performance for sorted neighbor query. The sorted neighbor list can speed up neighbor list intersection, which in turn accelerates triangle counting and subgraph matching that need neighbor list intersection.

In practice, it is valuable to export a snapshot of a dynamic graph at a given timestamp into the CSR structure for further graph analysis. LPMA has the best performance on the sorted CSR converting since LPMA has the similar structure as CSR.

Generally, for the graph algorithms that do not require neighbor list intersection, such as PageRank and BFS, these structures have similar performances. Due to different parallel strategies, there are slight differences in performance under different data distributions. For graph queries with uneven vertex degree distribution, LPMA has a clear advantage. Hornet has the worst performance since Hornet does not employ the workload balance optimization. For the graph algorithms requiring neighbor list intersection, like triangle counting, LPMA and faimGraph have much better performance on triangle counting algorithm than Hornet and HashBased due to the sorted neighbor list.

8 Related Work

Generally, we classify dynamic graph structures on GPUs into four categories.

CSR-Based Structure: As an early effort, the dynamic compressed sparse row (DCSR) [14] is devised to handle the dynamic changes. By reserving empty cells in the column array, the insertions could be added into the data structure with a small cost. However if the insertions for row i over-range the reserve space, the offset of row i would be divided into discontinuous space in the column array. The defragmentation operation has to be conducted after the insertion to maintain the good locality of the data structure, thus, DCSR is not efficient for dynamic graphs due to high throughput insertions. DCSR does not support deletion and the time complexities of graph queries are high on DCSR [8]. Therefore, we do not consider DCSR as the comparable structure in our experiments.

Neighbor List Based Structure: The neighbor list based structures maintain a variable array or list to store the neighbor lists. STINGER [22] data structure is rst introduced as a dynamic graph structure for both temporal and spatial graphs with metadata for multi-core architectures, while cuSTINGER [23] extends STINGER to the GPUs. Hornet [9] designs a dynamic array management system to store the neighbor list of each node. First, Hornet uses block-arrays as store unit for edges. The system assigns each node a default number of the cells for each neighbor list. Once cells are full, the system reallocates the double size of the previous space. Hornet has a Vectorized Bit Tree for each Block-array to locate the empty cells and B^+ Trees of block-arrays to manage the Block-arrays. Hornet places the neighbor lists in block sizes that are powers of two which sets the upper bound for the memory allocated for the entire graph evolution: $2|E|$.

faimGraph [11] stores the adjacency lists in congurable size memory pages. The pages of one node's neighbor list are connected by the pointers. faimGraph uses a single memory pool on GPU to manage the memory pages. It also offers both structure-of-arrays (SoA) and array-of-structures (AoS) representations to store edge data.

Hash Based Structure: A hash table approach is made in Dynamic Graphs on the GPU [10]. Instead of array, this work uses hash table for the adjacency lists storage. The GPU memory is divided into fixed size chunks and each chunk presents a bucket of the hash table. This work deals with the hash collisions with linked list achieves $O(1)$ time complexity for single edge update.

PMA-Based Structure: GPMA+ [8] adopts Packed Memory Array (PMA) [15], [16] to maintain sorted arrays in CSR for dynamic graphs. PMA maintains a sorted array, but leaves gaps to accommodate fast updates with a bounded gap ratio. GPMA+ extends PMA to GPU and use it to store the sorted arrays in CSR. They propose a segment-oriented operations to parallelize edge insertions/deletions in a lock-free model. However, the *global balance* strategy in GPMA+ lead to more unnecessary re-balancing cost, which is the motivation of our proposed LPMA in this work.

9 Conclusion

In this paper, we propose a multi-level array LPMA to replace the traditional contiguous array structure in PMA and GPMA+ to process dynamic graphs on GPU, thus reducing the unnecessary re-balance during the expansion phase and effectively improving the efficiency of the extension phase. Theoretical analysis and extensive experiments demonstrate the superiority of LPMA in reducing data movement. Also, we propose a hybrid strategy can self-adaptively choose the more efficient strategy to execute between the top-down and bottom-up strategies according to the data storage density.

Acknowledgments

This work was supported by NSFC under grant 61932001 and U20A20174.

References

[1] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[2] S. Guha and A. McGregor. (2012) Graph synopses, sketches, and streams: A survey.

[3] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[4] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016, pp. 1–12.

[5] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on GPUs: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–35, 2018.

[6] H.-N. Tran and E. Cambria, "A survey of graph processing on graphics processing units," *The Journal of Supercomputing*, vol. 74, no. 5, pp. 2086–2115, 2018.

[7] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2013.

[8] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, 2017.

[9] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[10] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic graphs on the GPU," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 739–748.

[11] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 754–766.

[12] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on GPUs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 171–182.

[13] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: GPU-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 1249–1260.

[14] J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic sparse-matrix allocation on GPUs," in *International Conference on High Performance Computing*. Springer, 2016, pp. 61–80.

[15] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 399–409.

[16] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 4, pp. 26–es, 2007.

[17] C. McGinnis, "Pci-sig® fast tracks evolution to 32gt/s with pci express 5.0 architecture," *News Release, June*, vol. 7, 2017.

[18] K. Bogart and C. Stein, "Discrete math in computer science," *Department of Computer Mathematics and Department of Computer Science. Dartmouth College, Hanover, NH*, 2002.

[19] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

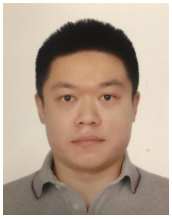
[20] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[21] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using SIMD instructions," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018*. ACM, 2018, pp. 1587–1602.

[22] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[23] O. Green and D. A. Bader, "custinger: Supporting dynamic graph algorithms for GPUs," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–6.

Lei Zou is a professor in Wangxuan Institute of Computer Technology of Peking University. He is also a faculty member in National Engineering Laboratory for Big Data Analysis and Applications (Peking University) and the Center for Data Science of Peking University. His research interests include graph databases and software/hardware co-design for graph computing.



Fan Zhang got his doctoral degree from Peking University in 2022. His research interests include graph stream processing and high performance graph computing .

Yinnian Lin is currently a Phd student at School of Intelligence Science and Technology, Peking University. His research interests include heterogeneous graph processing and high performance computing.

Yanpeng Yu was a undergraduate student in Peking University and he is now a PhD candidate at the department of computer science, Yale University. This work was done when he was in Peking University. Now, his research interests include memory disaggregation, programmable networks and operating systems.