Front. Comput. Sci., 2024, 18(4): 184106
https://doi.org/10.1007/s11704-023-2656-1

REVIEW ARTICLE

A survey on dynamic graph processing on GPUs: concepts, terminologies and systems

Hongru GAO[1,2], Xiaofei LIAO[1], Zhiyuan SHAO (✉)[1,2], Kexin LI[1,2], Jiajie CHEN[1], Hai JIN[1]

1 National Engineering Research Center for Big Data Technology and System/Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
2 Zhejiang Lab, Hangzhou 311121, China

Abstract Graphs that are used to model real-world entities with vertices and relationships among entities with edges, have proven to be a powerful tool for describing real-world problems in applications. In most real-world scenarios, entities and their relationships are subject to constant changes. Graphs that record such changes are called dynamic graphs. In recent years, the widespread application scenarios of dynamic graphs have stimulated extensive research on dynamic graph processing systems that continuously ingest graph updates and produce up-to-date graph analytics results. As the scale of dynamic graphs becomes larger, higher performance requirements are demanded to dynamic graph processing systems. With the massive parallel processing power and high memory bandwidth, GPUs become mainstream vehicles to accelerate dynamic graph processing tasks. GPU-based dynamic graph processing systems mainly address two challenges: maintaining the graph data when updates occur (i.e., graph updating) and producing analytics results in time (i.e., graph computing). In this paper, we survey GPU-based dynamic graph processing systems and review their methods on addressing both graph updating and graph computing. To comprehensively discuss existing dynamic graph processing systems on GPUs, we first introduce the terminologies of dynamic graph processing and then develop a taxonomy to describe the methods employed for graph updating and graph computing. In addition, we discuss the challenges and future research directions of dynamic graph processing on GPUs.

Keywords dynamic graphs, graph processing, graph algorithms, GPUs

1 Introduction

In the era of big data, graphs are widely used in representing real-world entities and the relationships among them [1]. For example, in transportation networks[1)], vertices and edges represent cities and highways that connect them, respectively [2]. Similarly, in social networks, vertices represent people, and edges represent social connections [3]. Graph algorithms such as Single-Source Shortest Path (SSSP) [4] and Community Detection [5] are applied to graph data to address real-world problems, such as finding the shortest path between two cities or identifying communities in a social network [6]. The process of executing graph algorithms on a graph dataset is referred to as *graph analytics*.

Graphs in real-world applications are generally dynamic, which means that vertices and/or edges are added into and/or deleted from graphs as entities and their relationships evolve over time [7]. For instance, in 2022, an average of 6 new user accounts are registered on Facebook every second [8], while approximately 3 new websites are created per second on the World Wide Web [9]. Furthermore, several rapidly changing scenarios also exist. For example, Twitter users generate around 10,000 tweets every second [10], while over 20,000 transactions are processed per second during peak times on Alibaba's e-commerce platform [11].

Dynamic graph analytics now play a crucial role in various applications, including fraud detection [12] and intrusion detection [13]. These applications require timely updates (with different levels of time-sensitivity) to graphs to ensure the integrity of graph data. Additionally, real-time processing of graph analytics workloads is necessary to generate up-to-date analytics results efficiently. These requirements have spurred studies on two aspects of dynamic graph processing: graph updating and graph computing. Graph updating aims to apply updates to graph data as quickly as possible when they stream in, and generate the most recent status of entities and their relationships [14–17]. On the other hand, graph computing aims to conduct graph analytics on the newest graph data and produce useful results in a timely manner [18–22].

Efficiencies in graph updating and graph computing are

Received October 31, 2022; accepted March 6, 2023

E-mail: zyshao@hust.edu.cn

1) In this paper, we use the terms "network" and "graph" interchangeably.

crucial, as will be exemplified by an e-commerce transaction graph and a fraud detection system [11]. On e-commerce platforms, fake transactions (i.e., frauds) are often carried out by buyers hired by sellers to make fake purchases, thereby increasing the popularity of specific commodities. In such transactions, payments always flow from the hired buyers to sellers after several passes. To identify fake transactions, the fraud detection system conducts a graph algorithm that detects the existence of cycle structures (i.e., paths whose source and destination are the same) on the transaction graph. If the payment activities were not applied to the transaction graph in a timely manner, fraud detection would run on an outdated graph, rendering the system incapable of detecting fake transactions promptly.

To achieve high performances on graph updating and graph computing in real-world graphs with increasing scales, GPUs are becoming mainstream vehicles due to their massive processing power and high memory bandwidth [23]. However, building efficient dynamic graph processing systems on GPU platforms still faces challenges. Regarding graph updating, given the limited capacity of GPU memory, it is challenging to design graph representations that can efficiently ingest updates while maintaining low space complexity. Additionally, with constant stream-in updates, graph representations must prevent frequent re-organization to avoid incurring prohibitive overheads on GPUs.

From the aspect of graph computing, there are two widely-used methods, i.e., recomputing and incremental computing, to perform graph algorithms on the constantly changing "current" graph. Recomputing involves conducting graph analytics on the updated graph from scratch, while incremental computing utilizes the previous analytics results and the incoming updates to incrementally update the results of the current graph. Compared to conducting computing on the entire graph data, incremental computing can avoid redundant computations, especially when a large portion of graph data remains unmodified. However, it may result in higher overheads than recomputing when the graph undergoes significant changes [24]. It is challenging to determine which approach can achieve higher computing efficiency when processing updates of different sizes on GPUs. Furthermore, both recomputing and incremental computing require balancing the load among the threads of GPUs and minimizing high atomic overheads. Alleviating the overheads caused by *branch divergence* and *uncoalesced memory accesses* on GPUs is also crucial for efficient graph computing.

This paper makes the following contributions:

- introduces the concept and graph analytics of dynamic graphs and discusses typical models used to represent dynamic graphs.
- proposes a taxonomy of GPU-based dynamic graph processing systems, surveys dynamic graph processing systems on GPUs, and classifies these systems according to the taxonomy we present.
- discusses the challenges and future research directions of dynamic graph processing on GPUs.

The remainder of this paper is organized as follows: Section 2 provides the background knowledge of graphs, graph algorithms, and the GPU architecture. Section 3 introduces the concept of dynamic graphs and various dynamic graph models adopted to achieve different objectives in real-world applications. In Section 4, we delve into two critical aspects of dynamic graph processing: graph updating and graph computing. We discuss the typical techniques employed by GPU-based dynamic graph processing systems and develop a taxonomy based on these techniques. Sections 5 and 6 review existing GPU-based dynamic graph processing systems that primarily address graph updating and graph computing, respectively, and classify them based on the taxonomy proposed in Section 4. In Section 7, we examine the related works of this paper. Finally, in Section 8, we conclude the paper by discussing the remaining challenges and future research directions of dynamic graph processing on GPUs.

2 Background

This section introduces the background knowledge of this paper, including the definition of "graph", widely-used graph representations, typical graph algorithms that are widely discussed under dynamic scenarios, and the GPU architecture.

2.1 Definition of graph

We take the classical definition of "graph"[2] from [25] and give the definition in the following. We also list the notations used in explaining the definition of "graph" in Table 1.

Definition 1 (**graph**) A graph can be denoted as $G = (V, E)$, where V is a finite set of vertices called the vertex set, and E is a binary relation on $V \times V$ that represents the set of the edges called the edge set.

A graph is either directed or undirected. In a directed graph, vertices are connected by directed edges. Each edge has a direction and is denoted as an ordered pair $(u, v) \in E$, where u is the source vertex and v is the destination vertex. On the contrary, in an undirected graph, vertices are connected by undirected edges, each of which is denoted by an unordered pair $\{u, v\}$, i.e., $\{u, v\} = \{v, u\}$. In practice, to efficiently query edges, an undirected edge is generally represented by two

Table 1 Notations for graphs

Symbol	Description
$G = (V, E)$	A graph G (directed or undirected), where V is the vertex set and E is the edge set.
$\|V\|$	The number of vertices in graph G.
$\|E\|$	The number of edges in graph G.
$N_{in}(u), N_{out}(u)$	In-neighbor set and out-neighbor set of vertex u in a directed graph.
$d_{in}(u), d_{out}(u)$	In-degree and out-degree of vertex u, denoting the number of in-neighbors and out-neighbors of u, respectively.
$N(u)$	Neighbor set of vertex u in an undirected graph.
$d(u)$	Degree of vertex u, denoting the number of neighbors of u.

[2] The "graph" we discuss in this paper refers to simple graph.

directed edges with opposite directions [26].

Moreover, if each edge of a (directed or undirected) graph is assigned a numeric label, called *edge weight*, by a weight function $f: E \to \mathbb{R}$, the graph is generally called (directed or undirected) *weighted graph*. A weighted graph is typically denoted as $G = (V, E, W)$, where W is the set of edge weights, and an edge in the graph is denoted as (u, v, w), where u, v are the two endpoints of the edge and w is the edge weight. Fig. 1 gives examples of undirected, directed, undirected weighted, and directed weighted graphs.

For a directed (unweighted or weighted) graph, the set of the in-neighbors of vertex u is denoted as $N_{in}(u) = \{v \,|\, (v,u) \in E\}$, while the set of the out-neighbors of u is denoted as $N_{out}(u) = \{v \,|\, (u,v) \in E\}$. The in-degree and out-degree of u, represented by $d_{in}(u)$ and $d_{out}(u)$, are the number of u's in-neighbors and out-neighbors, respectively. For an undirected graph, the set of the neighbors of u is represented by $N(u) = \{v \,|\, \{u,v\} \in E\}$, and the degree of u is the number of the neighbors of u, denoted as $d(u)$. We say vertex u is *adjacent* to vertex v if there is an edge connecting these two vertices, and the edge is called an *incident* edge for both vertex u and vertex v [27].

2.2 Graph representation

Graph representations are essentially data structures used to organize and store graph data in memory. In the following, we enumerate and discuss four typical graph representations [25]. Fig. 2 provides presentations of the example graph in Fig. 1(d).

- **Adjacency matrix**: The adjacency matrix, which is typically denoted by the notation A, represents a graph $G = (V, E, W)$ by using a $|V| \times |V|$ two-dimensional matrix, where the value of a_{ij} (the element of A at the i^{th} row and j^{th} column) indicates the weight of edge (i, j), and $a_{ij} = 0$ if $(i, j) \notin E$. The space complexity of the adjacency matrix is $\Theta\left(|V|^2\right)$.

Fig. 2(a) shows the adjacency matrix representation of the graph in Fig. 1(d). We can easily retrieve the weight of an edge by referencing the corresponding elements in the matrix using the two endpoint IDs of the edge. However, for sparse graphs whose $|E| \ll |V|^2$, the adjacency matrix wastes a large amount of memory in storing zero elements, leading to inefficiency in terms of memory usage.

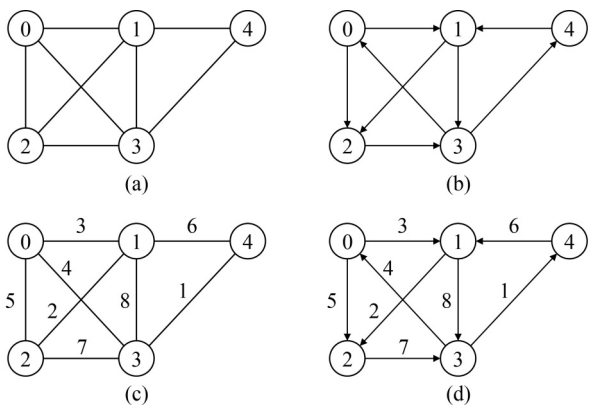

Fig. 1 Examples of undirected, directed, undirected weighted and directed weighted graphs (the number attached to an edge is the edge weight). (a) An example undirected graph; (b) an example directed graph; (c) an example undirected weighted graph; (d) an example directed weighted graph

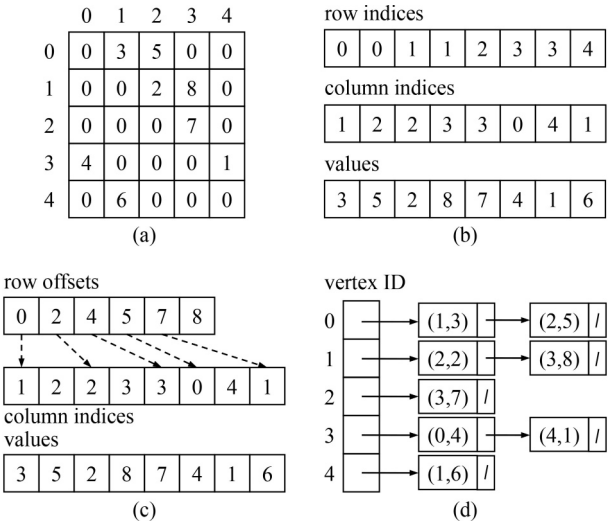

Fig. 2 Graph representations used to represent the example graph in Fig. 1(d). (a) Adjacency matrix A = (a_{ij}); (b) coordinate list; (c) compressed sparse row (dotted arrows indicate the starting offsets of the out-neighbors of vertices); (d) adjacency list (solid arrows denote pointers to objects, and the notation "/" denotes pointers that do not refer to any object)

- **Coordinate list** The *Coordinate List* (COO) is a graph representation that uses three equal-length arrays sized $|E|$, named `row indices`, `column indices`, and `values` to store the source vertices, destination vertices and weights of all edges in the graph $G = (V, E, W)$. The space complexity of COO is $\Theta(|E|)$.

To efficiently access graph data in COO, edges are generally stored in either row-major or column-major order (with respect to the adjacency matrix), depending on the access pattern of graph algorithms. The row-major COO of the graph in Fig. 1(d) is shown in Fig. 2(b). The COO representation is more memory-efficient than the adjacency matrix as it only stores non-zero elements of the adjacency matrix. However, accessing a certain vertex and its out-neighbors in COO requires traversing the `row indices` array, leading to high overheads. Besides, COO stores redundant source vertex data in the `row indices` array, as shown in Fig. 2(b).

- **Compressed sparse row** To alleviate the traversal overheads and data redundancy in COO, a *Compressed Sparse Row* (CSR) represents a graph $G = (V, E, W)$ by replacing the `row indices` array in COO with a `row offsets` array to efficiently access edges by source vertices, and leaving the `column indices` and `values` arrays unchanged. Specifically, for $i = 0, 1, 2, \ldots, |V| - 1$, the i^{th} and $(i+1)^{\text{th}}$ elements of the `row offsets` array indicate the starting and ending offsets of vertex i's out-neighbors in the `column indices` array, respectively. The space complexity of CSR is $\Theta(|E| + |V|)$, as the length of the `row offsets` array is $(|V| + 1)$ and that of the `column indices` and `values` arrays is $|E|$.

Fig. 2(c) shows the CSR representation of the graph in Fig. 1(d). In this example, the 0^{th} and 1^{st} elements of the `row offsets` array store the offset values (0 and 2

respectively), which locate vertex 0's out-neighbors (vertices 1 and 2) in the `column indices` array. Nevertheless, the CSR representation incurs extra overheads of accessing edge data by row offsets compared to the adjacency matrix.

● **Adjacency list** The adjacency list represents a graph $G=(V,E,W)$ by using an array of linked lists, where each index of the array indicates a source vertex ID, and its corresponding linked list stores all outgoing edges from that vertex, including their destination vertices and edge weights. The space complexity of the adjacency list is also $\Theta(|E|+|V|)$, as the length of the array is $|V|$ and the number of total linked list nodes for storing edge data is $|E|$. Fig. 2(d) shows the adjacency list representation of the graph in Fig. 1(d). However, the adjacency list suffers from poor locality when accessing edge data, since the edges of a vertex are generally stored non-contiguously in memory.

2.3 Graph algorithm

In the following, we briefly introduce six typical graph algorithms that are widely studied under dynamic scenarios.

● **Breadth-first search and single-source shortest path** Breadth-first search (BFS) is a graph traversal algorithm that enables the discovery of vertices reachable from a given source vertex u in a graph G. In order to execute the BFS algorithm, a *frontier queue* is typically maintained. Initially, the source vertex is enqueued, and at each subsequent step, an element is dequeued and its unvisited neighbors are added to the end of the queue. The time complexity of the BFS algorithm is $O(V+E)$. *Single-Source Shortest Path* (SSSP) algorithms are used to compute the shortest paths from a source vertex to all other vertices in a graph. Dijkstra's algorithm [25] is a well-known SSSP algorithm. In Dijkstra's algorithm, two sets of vertices V_1 and $V_2=V-V_1$ are maintained. V_1 contains the vertices whose shortest paths from the source vertex have been computed, while V_2 contains the remaining vertices. During each iteration of the algorithm, the vertex from V_2 with the minimum shortest-path value is added to V_1, and then all its out-neighbors are relaxed. This process is repeated until V_2 is empty. The time complexity of Dijkstra's algorithm is $O\left(|V|^2\right)$.

● **Triangle counting** Triangle counting algorithms are used to calculate the number of triangles in a given graph. A well-known approach for counting triangles is through the use of neighbor intersection [28]. This method involves conducting neighbor intersection on each $(u,v)\in E$, which identifies the common neighbors of u and v. These common neighbors can then be used to determine all the triangles that contain edge (u,v). Finally, the total number of triangles in the graph is computed by summing the results of neighbor intersection on all edges, while excluding repeatedly counted triangles. The time complexity of neighbor intersection is $O(|E|.d_{max})$, where d_{max} is the maximum degree among all vertices.

● **PageRank and personalized PageRank** PageRank algorithms are used to rank webpages by calculating their probabilities (denoted as *PR* values) of being accessed by users following a random walk model [29–31]. Under the random walk model, when a user is browsing a webpage, he (or she) has a probability of α, referred to as the *damping factor* [32], to browse one of the out-neighbors of the current webpage through hyperlinks. Alternatively, with a probability of $(1-\alpha)$, the user may get bored and jump to a random webpage. In general, a PageRank algorithm is essentially an iterative process [33], as shown in Eq. (1):

$$p^{(i)}=\frac{(1-\alpha)}{N}\vec{1}+\alpha W p^{(i-1)}, i=1,2,\ldots, \tag{1}$$

where N is the number of webpages (vertices), W is the transition matrix[3)], p is a N-dimensional vector indicating the *PR* values of webpages, and $\vec{1}$ is a N-dimensional vector with each element equal to 1. The algorithm starts with an initial vector p^0, where all element values are all equal to $\frac{1}{N}$. Eq. (1) is then iteratively conducted until the values of p converge to $p^{(i)}$ such that $\left|p^{(i)}-p^{(i-1)}\right|<\epsilon$, where ϵ is a predefined small value. Finally, $p^{(i)}$ is produced as the final results. The time complexity of PageRank algorithm is $O\left(|V|^2\right)$.

Personalized PageRank (PPR) algorithms [34] are a variant of PageRank algorithms. PPR algorithms are employed to measure the proximity of each webpage to a specific *source webpage* by calculating the probabilities (denoted as *PPR* value) that webpages can be accessed following a PPR random walk model. Under the PPR random walk model, a user has a probability of α to browse one of the out-neighbor webpages of the current webpage, or has a probability of $(1-\alpha)$ to return to the source webpage, instead of jumping to a webpage randomly. The iterative process for PPR is expressed in Eq. (2):

$$p_s^{(i)}=(1-\alpha)e_s+\alpha W p_s^{(i-1)}, i=1,2,\ldots, \tag{2}$$

where p_s is a N-dimensional vector that indicates the *PPR* values of webpages with respect to the source webpage s, and e_s is a N-dimensional vector with all elements set to 0, except

Algorithm 1 Vertex peeling for obtaining k-cores

Input: Graph G, Integer k, Integer $flag$
Output: all k-cores where $1\leqslant k\leqslant k_{max}$

```
1  k ← 1;
2  while G ≠ ϕ do
3      flag ← 1;
4      while flag == 1 do
5          flag ← 0;
6          foreach v ∈ V do
7              if d(v) < k then
8                  flag ← 1;
9                  remove v and its incident edges from
                     G;
10                 update the degree of v's neighbors;
11     if G ≠ ϕ then
12         Output G as k-core;
13         k ← k + 1;
```

3) A transition matrix $W=A^TD^{-1}$. A is the adjacency matrix of the graph, and $D=\left(d_{ij}\right)$ is a diagonal matrix, where d_{ii} is the out-degree of vertex i, and $d_{ij}=0$ if $i\neq j$.

for the element corresponding to the source webpage, which is set to 1. The algorithm of PPR is similar to that of PageRank. Starting with $p_s^0 = e_s$, Equation (2) is performed iteratively until p_s converges. The time complexity of PPR is also $O(|V|^2)$.

- **k-core decomposition** A k-core is the maximal[4] subgraph, where each vertex has at least k neighbors (k is a positive integer). For example, in Fig. 1(a), the entire graph is the 1-core, and also the 2-core. The subgraph constituted by vertices 0, 1, 2 and 3 and the edges connecting them, is the 3-core, and is also the k_{max}-core, where k_{max} indicates the maximum value of k to obtain a non-empty k-core. The k-core decomposition algorithm aims to find all k-core subgraphs (k ranging from 1 to k_{max}) in an input graph [35,36]. The method of *vertex peeling* [37,38] is widely adopted to conduct k-core decomposition. We list the pseudo-code of vertex peeling in Algorithm 1. Vertex peeling is an iterative algorithm that obtains all k-core subgraphs by continuously increasing k starting from 1 (Line 1). For each k, the algorithm iteratively traverses all vertices in the graph (Line 6) and removes the vertices with fewer than k neighbors and their incident edges (Lines 7–10) until no such vertices remain in the graph. After that, the remaining graph with all vertices having degrees of at least k, is output as the k-core (Line 12), and then k is added by 1 to compute $(k+1)$-core (Line 13). The time complexity of vertex peeling is $O(|V|+|E|)$ [38].

2.4 GPU architecture

A graphics processing unit (GPU) typically consists of multiple streaming multiprocessors (SMs), each of which contains hundreds or even thousands of streaming processors (SPs). Taking NVIDIA GPUs as an example, they adopt a computing architecture composed of *thread*, *warp*, *block*, and *grid* [39]. In NVIDIA GPUs, each SP is mapped to a thread, and 32 threads with consecutive IDs form a warp that executes in a Single Instruction Multiple Threads (SIMT) fashion. When the threads of a warp encounter a condition branch, only some of the threads proceed to a particular branch while the others must wait, leading to a phenomenon called branch divergence which would incur high performance penalties. NVIDIA GPUs logically organize their threads as blocks, the number of which can be set by the programmer. All threads of a block must operate within the same SM, and multiple blocks form a grid. During computation, the threads in the same grid execute the same computing kernel.

In terms of the storage architecture of GPUs, *shared memory* is the local memory of an SM, while *global memory* is the "shared" memory of all SMs of a GPU. Generally, the global memory in a GPU has a much larger capacity than the shared memories, but when accessed by a thread of an SM, it incurs substantially higher latency than the shared memory.

When conducting graph processing tasks, GPUs place specific requirements on both graph representations and algorithms. Graph representations with low space complexity are preferred due to the limited storage capacity of GPU global memory. Besides, compact graph representations that store the edges of a vertex contiguously are also preferred, as they ensure efficient memory accesses to the neighbor list of a vertex under processing. Regarding graph algorithms, specific designs are required to minimize the overheads that degrade the computing performance of graph algorithms on GPUs, including branch divergence and *uncoalesced memory access* incurred by a graph algorithm's random accesses towards the properties of a graph's vertices.

Furthermore, in dynamic graph processing, which can be regarded as a special case of graph processing, two additional challenges arise. The first challenge is to improve the efficiency when applying a large number of updates towards a graph stored in the global memory of a GPU. The second challenge is to balance the load among the threads of GPUs when the size of the neighbor lists of vertices varies significantly during graph mutations. To address these challenges, a number of dynamic graph processing systems have been proposed in recent years.

3 Graph mutation

In this section, we switch our discussions to scenarios where the graph mutates. We illustrate two types of graph modifications, based on which we present the concept of dynamic graphs. In addition, we discuss two types of analytics workloads on dynamic graphs and three typical dynamic graph models used to represent dynamic graphs. Moreover, the challenges that arise when conducting graph mutations on GPUs are also discussed.

3.1 Two types of modifications

Updates that modify a graph can be categorized into *topology updates* and *attribute updates*, also known as *structural updates* and *non-structural updates* [40]. Topology updates refer to the modifications to graph topology, including inserting vertices and edges to and deleting vertices and edges from a graph. On the other hand, attribute updates only modify graph attributes (i.e., edge weights, vertex properties[5]) and leave graph topology unchanged.

Fig. 3 gives an example of applying a sequence of updates towards the example graph in Fig. 1(d). The update operations in this example include both topology updates (the first six updates in Fig. 3) and attribute updates (the last two updates in Fig. 3). After absorbing all these updates, the "*Original graph*" transforms to the "*Updated graph*" shown in Fig. 3.

In this example, we can observe that the first six updates lead to changes in graph topology. Specifically, compared to the original graph, its updated graph contains a new vertex 5 and two new edges (1, 5, 7), (3, 5, 1), while the original edges (3, 4, 1), (4, 1, 6), and vertex 4 are removed. The last two

[4] When finding a subgraph with a specific limitation, we refer to the resulting subgraph as "maximal" if it is not contained in any larger subgraph with the same limitation.

[5] Vertex properties are the data attached to vertices. For example, in a social network, people's information, e.g., personal IDs, ages, and residential addresses, can be modeled as vertex properties. Considering that the classical definition of weighted graph given in Section 2 only involves edge weights, we introduce vertex properties in this subsection to comprehensively describe graph attributes.

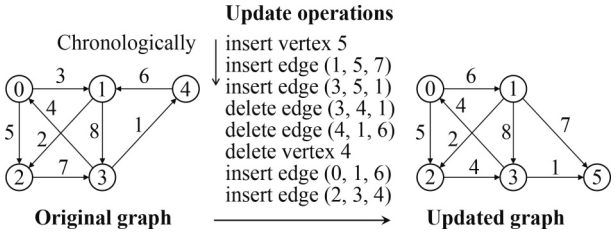

Fig. 3 Updating the example graph in Fig. 1(d) with a series of graph updates listed in chronological order

updates modify the weights[6] of edges $(0,1)$ and $(2,3)$, as these two edges have already existed in the original graph.

3.2 Dynamic graph

Various interpretations of "dynamic graph" have been developed in different works. For example, Li et al. [41] define a dynamic graph as a stream of edges, each of which indicates the arrival of an edge update, whereas Margan et al. [42] define a dynamic graph as a series of graph versions, each version representing the state of the graph at a particular time.

From our perspective, the above diverse interpretations of dynamic graph are more like "models" designed to organize dynamic graph data to tackle specific real-world problems. To be specific, Li et al. [41] design a dynamic graph model that maintains the latest graph data over graph updates, while Margan et al. [42] devise a dynamic graph model that reflects the historical evolution of the graph. A detailed discussion of these dynamic graph models will be given in Section 3.4. Nevertheless, we argue that the term "dynamic graph" should be a general concept for describing changing graph data, rather than referring to any specific model. Our definition of dynamic graph is presented in the following.

Definition 2 (**dynamic graph**) A dynamic graph can be defined as a graph whose topology and/or attributes change over time by ingesting a series of graph updates.

From the aspect of which graph data (vertex, edge, vertex property, edge weight) are updated, Harary [43] classifies dynamic graphs into four basic types: *vertex-dynamic* graphs, *edge-dynamic* graphs, *vertex weighted dynamic* graphs and *edge weighted dynamic* graphs. These four types of dynamic graphs cover the cases where only one of the vertex set, edge set, vertex property set, and edge weight set is modified. However, Zaki et al. [40] point out that, all the four sets in dynamic graphs change over time in most real-world scenarios. Such dynamic graphs are called fully dynamic graphs.

3.3 Analytics of dynamic graphs

In real-world applications, analysts perform graph analytics on dynamic graphs to mine informative results from the changing graph data. Graph analytics on dynamic graphs can be categorized into two types: *online* (real-time) analytics and *offline* (historical) analytics [24,44–46], depending on when the graph data are processed.

- **Online (real-time) analytics** With a dynamic graph continuously ingesting updates, online analytics are conducted on the most recent graph data (called *live graph data*) to produce up-to-date analytics results in real-time. It is imperative to keep the results fresh and correct over time in online analytics, since outdated results caused by changing graph data can be misleading.
- **Offline (historical) analytics** Instead of processing live graph data, offline analytics rely on historical graph data, which are archived during graph evolution, to analyze evolutionary features (e.g., mining frequently changing components) of the graph. The key challenge in offline analytics lies in how to store historical graph data and conduct graph algorithms efficiently over multiple historical graphs.

The results obtained from online analytics and offline analytics can be considered as complementary, as they serve different objectives in analyzing dynamic graphs. Online analytics need to provide real-time responses to graph updates, and therefore the computing conducted by online analytics tends to have low time complexity, which varies depending on the specific application requirements. In contrast, offline analytics aim to conduct in-depth analytics on graph changes, and therefore the computing tends to be more complex. For instance, in the context of fraud detection discussed in Section 1, online analytics use polynomial time complexity algorithms such as real-time cycle detection [11] to detect fake transactions promptly in the live graph, while offline analytics employ computationally expensive algorithms such as clique enumeration [47,48] (an NP-hard problem) to identify suspicious users in historical transaction graphs. By imposing penalties (i.e., freezing their accounts) on these users, the recurrence of fake transactions among them can be effectively prevented in the live graph.

3.4 Models of dynamic graphs

In order to support online and offline analytics on dynamic graphs, various types of models [40,49] have been developed to represent dynamic graphs by organizing graph data in different ways. In this subsection, we will introduce three typical dynamic graph models: *stream graph model* [41,49], *snapshot sequence model* [42,50], and *temporal graph model*. Table 2 shows the comparisons of these three models from four aspects: the type of analytics workloads they support, the method of storing historical graph data, the pattern of ingesting updates, and the preferred application scenarios, which will be discussed in detail below.

3.4.1 Stream graph model

The stream graph model, also called "streaming graph" [51] or "graph stream" [49], represents the graph updates as a stream. An example of a stream graph model under edge insertion scenarios is illustrated in Fig. 4. The dynamic graph starts with an empty graph at t_0, and then ingests two edges $(0,1)$ and $(1,2)$ at t_1, two edges $(0,2)$ and $(0,3)$ at t_2, and one edge $(2,3)$ at t_3. These newly inserted edges are listed sequentially

[6] Since this paper only discusses simple graphs, directed graphs do not allow for multiple edges with the same direction existing between two vertices, and thus inserting an edge that already exists in the graph will only update the edge weight.

Table 2 Comparisons between the stream graph model, the snapshot sequence model, and the temporal graph model

	Stream graph	Snapshot sequence	Temporal graph
Analytics workloads	online	online & offline	offline
Historical graph data	not supported	coarse-grained (with snapshots)	fine-grained (with timestamps)
Ingesting updates	single & batch (with fixed batch size)	batch (with fixed time interval)	not mentioned
Preferred scenarios	high time-sensitivity	low time-sensitivity & long-term relation	short-term relation

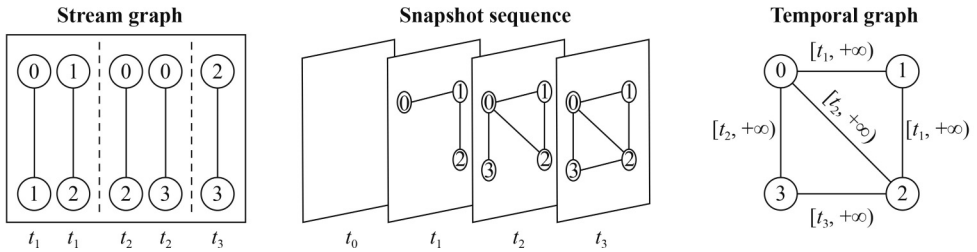

Fig. 4 Schematic figures of the stream graph, snapshot sequence, temporal graph models under edge insertions (edges will not be removed once they are inserted), where t_0, t_1, t_2, t_3 denote time points

according to their arriving time points in Fig. 4. The general stream graph model can be defined as follows:

Definition 3 (**stream graph model**) A stream graph model can be denoted as a stream of edges $\{e_1, e_2, e_3, \ldots\}$, where $e_i = (u_i, v_i, w_i, \delta_i)$, $\delta_i \in \{-1, 1\}$. $\delta_i = 1$ indicates the insertion of e_i to the graph, while $\delta_i = -1$ indicates the deletion of e_i from the graph.

The stream graph model is originally proposed in the situations where the scale of entire graph data is too large to fit into internal storage [52]. To perform graph analytics in such a graph, the graph is denoted as an *edge stream* $\{e_1, e_2, \ldots\}$, and graph algorithms (e.g., triangle counting [53]) are redesigned for the stream graph model. Under dynamic graph scenarios, the stream graph model lists edge insertions and deletions in the order of their arrival. The stream graph model typically only maintains the live graph data, and therefore it only supports online analytics.

When applying updates to a graph, [54–56] apply one update at a time, following the above edge stream model. The edge stream model is suitable for the applications where graphs evolve gradually and are highly time-sensitive[7] on analytics results. Specifically, in these applications, graphs do not ingest a large number of updates per unit time, and each update needs to be applied to the graph immediately when it occurs. A typical example of this type of application is computing shortest paths in transportation graphs[8], where each new edge (i.e., a new road between cities) must be instantaneously applied to ensure that the results of the SSSP algorithm are up-to-date. In transportation graphs, it is feasible to apply updates to graph data one at a time due to its low update rate. However, the edge stream model cannot efficiently handle real-world graphs with highly concurrent updates, e.g., a Twitter graph is updated with an average of 10,000 tweets per second [10]. Applying these large quantities of updates sequentially will result in high update latency.

To handle highly concurrent updates, a batch update method is proposed in [22,57]. In such a method, a graph is represented as a *batch stream* $\{G_1, G_2, \ldots\}$, each of which contains multiple (typically a fixed value) updates, called *batch updates*. The batch stream model aggregates a number of updates into a batch and applies these updates to the graph simultaneously. The batch stream model effectively improves the efficiency of graph updating by maximizing memory bandwidth and amortizing updating overheads, such as the re-organization overheads of data structures.

3.4.2 Snapshot sequence model

The snapshot sequence model [58,59] describes how a graph evolves by archiving a series of graph versions at different time points. In the example snapshot sequence in Fig. 4, the differences between graph versions at t_1 and t_2 reflect the new edges $(0,2)$ and $(0,3)$ applied to the graph during $(t_1, t_2]$. In the following, we will first introduce *graph snapshot*, the core concept in the definition of snapshot sequence model.

Definition 4 (**graph snapshot**) When a graph evolves, a graph snapshot $G_{t_i} = \left(V_{t_i}, E_{t_i}, W_{t_i}\right)$ indicates the state of the graph at time point t_i, with V_{t_i}, E_{t_i} and W_{t_i} representing the sets of vertices, edges, and edge weights that exist in the graph at t_i.

A graph snapshot (called snapshot for short) G_{t_i} is in essence a static graph that archives the graph topology and attributes of G at a particular past time point t_i. Based on the definition of snapshot, we define the snapshot sequence model as follows:

Definition 5 (**snapshot sequence model**) A snapshot sequence model can be denoted as $\left\{G_{t_0}, G_{t_1}, \ldots, G_{t_n}\right\}$ archived at different time points $\{t_0, t_1, \ldots, t_n\}$. The snapshot $G_{t_{i+1}}$ $(i = 0, 1, \ldots, n-1)$ is constructed by integrating the updates within $(t_i, t_{i+1}]$ into its previous version G_{t_i}. Among the time points, t_0 is the initial time, and G_{t_0} represents the earliest snapshot. t_n indicates the time point closest to the current time, and G_{t_n} denotes the most recent snapshot.

[7] We use the term "highly time-sensitive" to describe the graph application where updates need to be applied to graph data within a short time constraint.
[8] A transportation graph can be viewed as a dynamic graph with an extremely low update rate, as building a direct transportation line between two cities always takes a long time.

From the above definition, the snapshot sequence model is composed of a series of snapshots that are typically archived at fixed intervals (e.g., daily, weekly), and provides a coarse-grained description of graph changes. The transition from G_{t_i} to $G_{t_{i+1}}$ reflects the differences between the graph at t_i and t_{i+1}. The snapshot sequence model is suitable for describing the changes of "relationship-stable graphs" [46], where the edges have longer duration than the archiving interval, so that no update information is lost. Specifically, in such graphs, no vertices or edges are added to the graph and then removed within an archiving interval; otherwise, it would be challenging to describe the changes of these vertices and edges by two successive snapshots. As for the update rate, the snapshot sequence model is suitable for graphs with low update rates. In the case of fast-updating graphs, the snapshot sequence model faces the problem of selecting an appropriate archiving interval. A short interval leads to storage explosion, while a long one may cause information loss. Moreover, the existence of time intervals prevents the snapshot sequence model from being used in highly time-sensitive applications.

A graph that models co-worker relationships is a typical example suitable for the snapshot sequence model, because the relationships of co-workers generally persist for a long period, such as several years. Another example is a citation network, where edges (representing citation relationships between two papers) can be regarded to have infinite durations once they are established. Both these two graphs are generally updated at a time scale of weeks or months [46].

The offline analytics are well supported by the snapshot sequence since historical graph data are archived in a regular way and can be easily accessed. Besides, the latest snapshot can be extracted for online analytics.

3.4.3 Temporal graph model

The temporal graph model [40] records graph changing by attaching timestamps to the graph data. In the temporal graph model shown in Fig. 4, taking edge $(0,1)$ as an example, its timestamps indicate that it is inserted to the graph at the time point t_1 and will not be removed. In the following, we will give the general definition of temporal graph model:

Definition 6 (temporal graph model) A temporal graph model can be modeled as $G = (V', E', W')$ where V' is a set of tuples (v, t_s, t_e), with t_s, t_e indicate when vertex v is inserted into and removed from the graph; E' is a set of tuples (u, v, w, t_s, t_e), with t_s, t_e indicate when edge (u, v, w) is inserted into and removed from the graph. For the vertices and edges that currently exist in the graph, their t_e can be set to -1 temporarily.

Different from the snapshot sequence, the temporal graph model provides precise and fine-grained temporal descriptions of graph evolution by using timestamps [60,61]. Hence, it can be used to represent "short-term-event graphs" where edges are of short durations. The temporal graph model has no preference for the update rate and can capture graph changes at any time scale.

The temporal graph model can be further subdivided into two categories: *contact sequence* and *interval graph* [62] to describe graphs with edge duration of different granularities:

- **Contact sequence** The contact sequence model is used to represent the graphs whose edge duration is negligible or not important for graph analytics. In this model, only one timestamp is assigned to each edge, i.e., $t_s = t_e$. For instance, in an email graph[9], an edge (email) can be considered instantaneous (in essence an approximation by ignoring the transmission latency of emails), and the only timestamp of each edge indicates the time when the email is delivered.
- **Interval graph** The interval graph model represents the graphs whose edges are active during a short interval rather than at an instantaneous point. Each edge is accompanied with two time points t_s, t_e to indicate when it exists in the graph. This model is suitable for representing graphs such as phonecall graphs, where edges have different durations, and the proximity between two people can be implied by the length of the phone call.

In view of the fact that interactions between two entities may recur as time passes, e.g., two people may send multiple emails and make several phone calls, only one timestamp t_s in the contact sequence or two timestamps t_s, t_e in the interval graph cannot completely record historical changes of edge data. To address this issue, the original temporal graph model is improved by replacing the single timestamp in the contact sequence with a set of timestamps, and replacing the two timestamps in the interval graph with a set of timestamp pairs [61].

The temporal graph model is generally not used for online analytics, since live graph data are stored together with historical data as a single graph [40]. Extracting the live data from the entire graph would lead to intolerable time overheads in online analytics. For offline analytics, the temporal graph model is typically employed in the scenarios of short-term events due to its fine-grained nature. However, a historical version of the graph at a given time point may not contain enough edges to obtain informative results [63]. Considering an email network with one-second time granularity, only a handful of emails are being delivered among millions of users (vertices) at a second [64]. To address this issue, a time window method [62] is generally used to extract a historical version of the graph that includes all vertices and edges present within the time window.

3.5 Graph mutation on GPUs

In the above discussions, the concepts of graph mutation are presented, including two types of graph modifications, the definition, analytics, and models of dynamic graphs. When conducting graph mutations on GPUs, there are two crucial challenges that need to be addressed:

- **Design graph representations that efficiently ingest updates** Graph updates applied to graph representations maintained on GPUs are generally unpredictable and irregular.

[9] In an email graph, we assume that an edge (email) is active only when it is being transmitted from a sender to a receiver. The edges in a phonecall graph can be defined in a similar way.

These updates need to be ingested to graph data as fast as possible to ensure the timeliness of online analytics. Consequently, graph representations need to support efficient graph modifications to meet the demands of real-time processing.

- **Balance loads among the threads of a GPU** Load balancing is critical to the computing performance when conducting computations on dynamic graphs on GPUs. Most real-world dynamic graph structures are irregular, with the degrees of vertices generally following a power-law distribution [1]. Besides, the sizes of neighbor lists of vertices also change with graph mutations. Therefore, straightforward computing task assignments to threads (e.g., assigning a single thread to process all edges of each vertex) will inevitably lead to load imbalance. As a result, efficient load-balancing strategies need to be developed to achieve high performance.

4 Dynamic graph processing on GPUs

This section presents the workflow of dynamic graph processing on GPUs and discusses our taxonomy of existing methods for graph updating and graph computing. In the following of this paper, we focus on dynamic graph processing under the stream graph model and describe how GPUs handle real-time updates. The snapshot sequence and temporal graph models are beyond the scope of our discussions, since they need to store and process historical graph data, which is more like traditional static graph processing and does not highlight the process of merging updates to the graph.

4.1 Workflow of dynamic graph processing on GPUs

As the scales of real-world dynamic graphs increase, GPUs are becoming a mainstream vehicle to improve the efficiency of dynamic graph processing due to their rich parallel processing power and high memory bandwidth. As shown in Fig. 5, the general workflow of dynamic graph processing on GPUs consists of three phases [51,65,66]. Initially, incoming graph updates are gathered and stored in a buffer on the CPU host (Phase I). Then, when the specified condition (e.g., an upper limit of the number of gathered updates) is reached, the updates in the buffer are transmitted to the GPU device and ingested to the graph data (Phase II, *graph updating*). Generally, to exploit the massive parallel processing power and the high memory bandwidth of GPUs, batch updates are generally adopted. Finally, graph algorithms are conducted on the live graph to update the previous analytics results (Phase III, *graph computing*). The above three phases are repeated under continuously stream-in updates [57]. In the following subsections, we will discuss common methods for coping with graph updating and graph computing on GPUs.

4.2 Graph updating

To efficiently absorb continuous graph updates into the existing graph data, the adopted graph representation should be flexible enough to ingest updates without high overheads of reconstructing the data structure. Existing graph representations [67,68] of dynamic graphs are generally based on those of static graphs (e.g., CSR, adjacency list) with modifications to data structures for efficient ingestion of updates.

According to state-of-the-art studies [65,67–72], we divide the existing methods for applying updates to a graph into three types: array-based space management, CSR-based space management, and chain-based space management.

- **Array-based space management** Methods under this type employ a separate array (also called *adjacency array*) to store the edges of each vertex. The initial capacity of the adjacency array of a vertex is typically larger than the vertex's degree, with empty slots valued "null" existing at the end of the array. Fig. 6(a) shows an example insertion process of this type of graph updating method. During edge insertions, new edges are accommodated after the last non-null element of corresponding adjacency arrays. If an edge (i.e., the element 5 in Fig. 6(a)) is going to be inserted into a full array, a larger array will be allocated, and *all elements of the full array* (i.e., the elements 1, 2, 3, and 4 in Fig. 6(a)) *are copied to the larger one*, as well as modifying the pointer of the vertex to the new edge array. The method of array-based space management benefits from the compactness of graph representations. However, if the insertions to adjacency arrays occur frequently, this type of method will incur high overheads of memory allocation and data movement in GPU

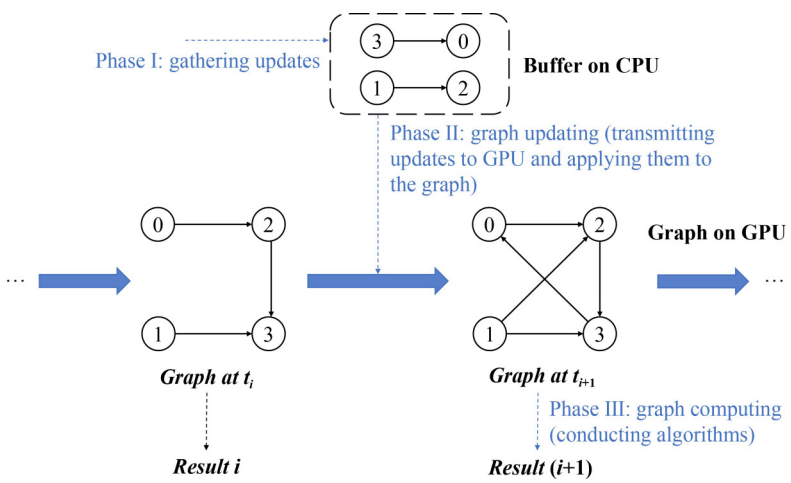

Fig. 5 General workflow of dynamic graph processing on GPUs

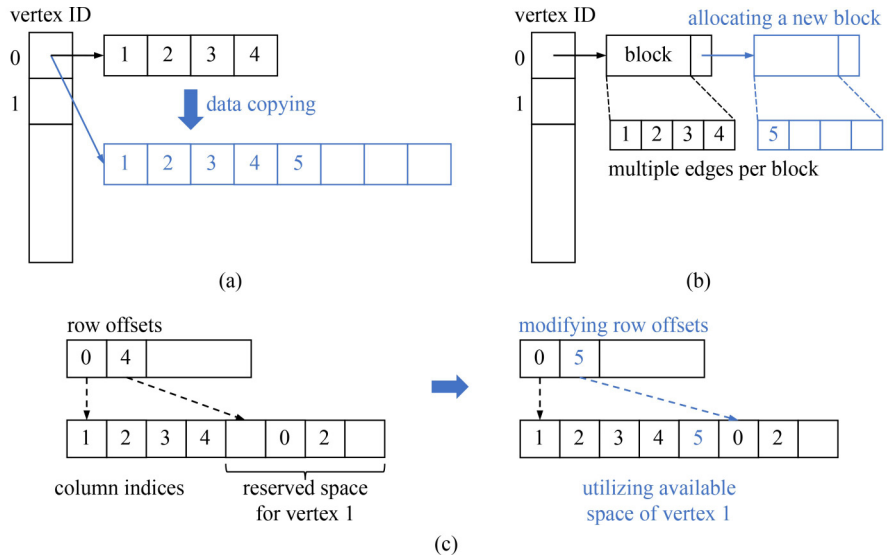

Fig. 6 Schematic figures of graph updating methods (blue parts show the process of merging graph updates). (a) Array-based space management; (b) chain-based space management; (c) CSR-based space management

memory.

● **CSR-based space management** Methods under this type use *variants of CSR* as their graph representations, which over-allocate the space of the `column indices` and `values` for each vertex. When the reserved space of a vertex is insufficient to incorporate incoming edges, this type of method first utilizes the allocated but unoccupied memory in the `column indices` and `values` arrays to store these new edges, while *minimizing the movement of existing data in these two arrays*. Then, the corresponding `row offsets` are modified to locate the new edges of the vertex. Fig. 6(c) shows an example insertion process of this method. As the reserved space of vertex 0 is exhausted, vertex 0 borrows the space of vertex 1 to incorporate the new edge (i.e., the element 5 in the `column indices` array). After that, the row offset of vertex 1 is changed from 4 to 5 to distinguish the edges of vertices 0 and 1. This type of method reduces the overheads of massive data movement in GPU memory as well as frequent memory reallocation by sharing reserved space among vertices. However, high atomic overheads are incurred when multiple vertices request the same reserved space. Furthermore, this type of method typically needs auxiliary data structures and extensive parameter optimizations to achieve high graph updating performance.

● **Chain-based space management** Methods under this type adopt a linked list of blocks [73] (also called block-based adjacency list) to accommodate the edges for each vertex. Each block is generally an array that can store multiple edges. For example, a block in Fig. 6(b) stores four edges. A block-based adjacency list yields better locality of the edges within linked list nodes than a general adjacency list. When the last block of a linked list does not have enough room to store new edges during edge insertions, a new block is allocated and linked to the end of the linked list to store the updates. It can be seen in Fig. 6(b) that a new block is allocated for vertex 0 to store the new edge (the element 5), since the existing block of vertex 0 is exhausted. This type of method prevents data movement but still incurs the overheads of GPU memory allocation when new blocks are allocated. Moreover, a trade-off exists between block capacity and efficiency, since a large one leads to wasted memory while a small one suffers from poor locality of edges and frequent allocating operations.

4.3 Graph computing

In the phase of graph computing, graph algorithms are performed on the live graph to obtain real-time analytics results. To meet the requirements for timeliness in different applications, graph algorithms need to be designed to optimize efficiency. In this paper, we classify graph algorithms for dynamic graphs into two categories: recomputing and incremental computing, which will be elaborated on below:

● **Recomputing** Graph algorithms of recomputing conduct computation on the entire live graph to derive analytics results from scratch. The approach of recomputing can be regarded as a combination of dynamic graph representations and static graph computing, since the analytics results only depend on the currently processed graph data. Previous graph data and analytics results do not need to be preserved or processed during recomputing processes. However, recomputing generally entails a considerable amount of redundant computation overlapping with the computation on the previous graph data. This redundancy poses challenges for graph computing on large graphs, where only a small portion of vertices and edges are modified.

● **Incremental computing** Incremental computing is an extensively studied method to handle graph computing in recent years [44,74,75]. Unlike recomputing, in incremental computing, computations are only conducted on the vertices and edges affected by the graph updates to obtain intermediate results. And the final analytics results of the graph algorithm are produced by combining the previous results with the intermediate results. Fig. 7 shows an example process of incremental computing. When performing graph analytics on graph G_1, it is feasible to obtain the analytics results "*Result* 1" by combining "*Result* 0" on the previous graph G_0 with the results of computing the graph data affected by the new edge

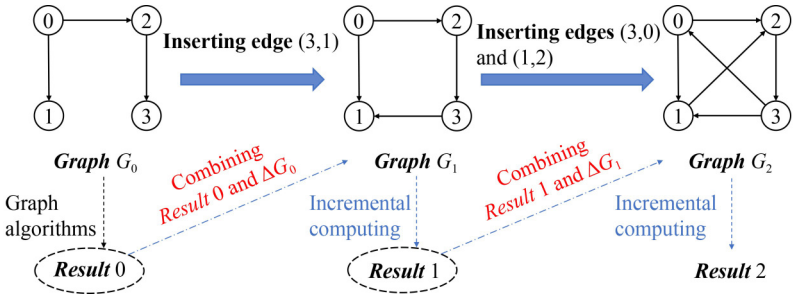

Fig. 7 An example of incremental computing processes

(3,1). The incremental computing process on graph G_2 is similar. However, in the event of significant graph changes, incremental computing may not perform better than recomputing. Besides, incremental computing incurs uncoalesced memory accesses on GPUs, as the affected vertices to be computed are non-contiguously stored in memory [76].

4.4 Taxonomy of dynamic graph processing systems on GPUs

In the foregoing discussions on graph updating and computing, we research the state-of-the-art methods for applying updates to a graph and conducting graph algorithms on the live graph. Based on the problem addressed by GPU-based dynamic graph processing systems and the method they employ, we develop a taxonomy for these systems, as shown in Fig. 8. Specifically, we first classify these systems into dynamic graph updating systems and dynamic graph computing systems. Dynamic graph updating systems can be further classified by the method used to ingest updates: array-based space management, CSR-based space management, or chain-based space management. Dynamic graph computing systems can be subdivided based on the approach employed to perform graph analytics: recomputing or incremental computing.

5 Dynamic graph updating systems

In this section, we review existing dynamic graph updating systems on GPUs. We first classify the systems we reviewed according to the taxonomy in Section 4.4. Then we introduce the example systems of different categories respectively and discuss the design differences of these systems.

5.1 Typical systems and classification

We survey typical dynamic graph updating systems on the GPU: cuSTINGER [69] and DCSR [67] in 2016, GPMA [65] and aimGraph [68] in 2017, Hornet [72], GPU-LSM [77], and faimGraph [70] in 2018, Slabhash [71] in 2020, and LPMA [78] in 2021. In Table 3, we classify these systems according to our taxonomy (Fig. 8). We categorize cuSTINGER, Hornet, and GPU-LSM into the type of array-based space management. For DCSR, GPMA, and LPMA, we classify them into the type of CSR-based space management. faimGraph (and its previous version aimGraph) and Slabhash are representatives of the type of chain-based space management. The remainder of this section follows our taxonomy and discusses in detail how each system works.

5.2 Systems using array-based space management

1) cuSTINGER [69]

cuSTINGER (an extension of CPU-based STINGER [73,79]), presented by Green et al. [69], is the first GPU-based dynamic graph updating system, which employs an array for each vertex to store its edges contiguously. cuSTINGER supports edge insertions and deletions. When processing mixed batch updates, cuSTINGER first divides them into an

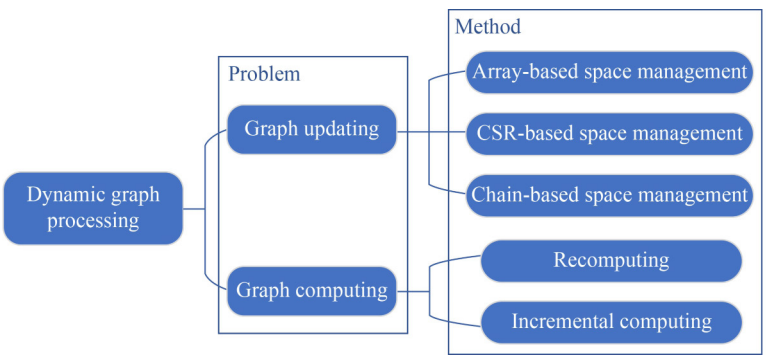

Fig. 8 A taxonomy of GPU-based dynamic graph processing systems

Table 3 A classification of GPU-based dynamic graph updating systems

Methods	Typical systems
array-based space management	cuSTINGER [69], Hornet [72], GPU-LSM [77]
CSR-based space management	DCSR [67], GPMA [65], LPMA [78]
chain-based space management	aimGraph [68], faimGraph [70], Slabhash [71]

insertion batch and a deletion batch, and then conducts these two batches separately. For edge insertions, updates are first scanned on the device for checking duplication (between batch updates and existing graph data, and in batch updates). Then, all unduplicated edges are added to the corresponding adjacency arrays. If an adjacency array is insufficient to incorporate the updates, a larger array will be allocated by the host for the vertex. After that, all edges in the old adjacency array are copied to the new one, and the uncompleted updates are inserted to the new array. For edge deletions, the edges to be deleted are first marked, and then each edge to be removed is replaced by an edge at the end of the adjacency array, which is conducted in parallel with high efficiency.

2) Hornet [72]

Hornet, proposed by Busato et al. [72], alleviates the overheads of switching to the CPU for GPU memory allocation by pre-allocating a series of *block-arrays* as a storage pool for all vertices. Besides, the *blocks* in block-arrays are recycled after the edges in blocks are copied to larger ones. As shown in Fig. 9(a), Hornet employs block-arrays to store edges of vertices and uses a vertex array to store the number of edges of each vertex and the pointer to its neighbor list. A block-array contains multiple blocks of the same size (called *blocksize*, and limited to powers of 2). The blocksizes of the three block-arrays in Fig. 9(a) are 1, 2, and 4, respectively.

Hornet supports inserting and deleting vertices and edges. Inserting vertices needs to allocate a larger vertex array and copy over the old one. Removing vertices is implemented by deleting the elements of the vertex array and reclaiming the corresponding blocks. Hornet separately handles edge insertions and deletions. For edge insertions, if a vertex requires additional space to store new edges, an empty block (traced by *Vectorized bit trees* and *B+ trees*) with double blocksize in existing block-arrays is assigned for the vertex. Then, all edges in the old block are copied to the new one. If no available block with requested blocksize can be found, a new block-array has to be allocated. Fig. 9(b) shows the process of inserting edges (0,4), (1,4) and (1,5) to the graph representation in Fig. 9(a). The block of vertex 0 has enough room to store (0,4), while the block of vertex 1 is full. Hence, a block sized 4 is assigned to vertex 1 to store (1,4) and (1,5). For edge deletions, this work only mentions that smaller blocks will be assigned for vertices if their degrees are less than half of the current blocksizes after removing edges.

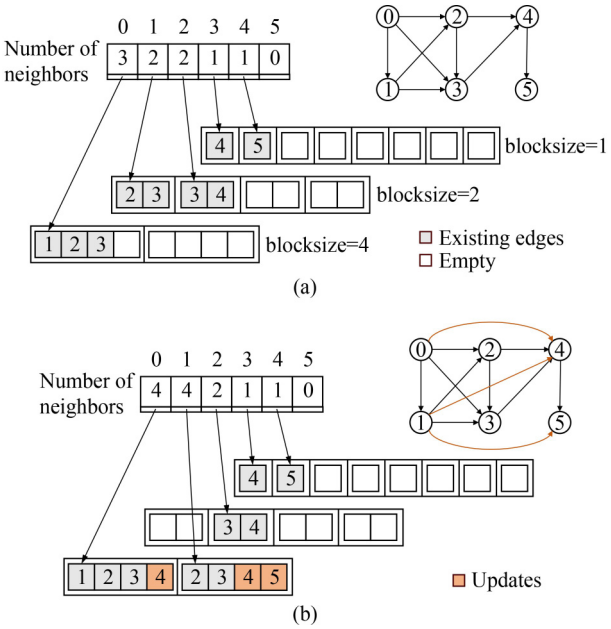

Fig. 9 An example of the graph representation used in Hornet and the process of conducting edge insertions on Hornet. (a) An example graph representation of Hornet (solid arrows indicate pointers of vertices to their edges); (b) inserting three edges (0, 4), (1, 4), (1, 5) to the Hornet in Fig. 9(a)

3) GPU-LSM [77]

GPU-LSM, proposed by Ashkiani et al. [77], maintains multi-levels of arrays, each of which contains multiple edges, as shown in Fig. 10. The array at Level i, where $i = 1, 2, \ldots$, is twice the size of that at Level $(i-1)$. GPU-LSM supports inserting and deleting edges. Initially, the size of batch updates is set to the size of the first-level array. When ingesting new edges, they are stored in the first-level array if it is unoccupied. Otherwise, they are merged with the elements at the first level and then attempt to be stored in the higher-level array. Such a process is repeatedly conducted until batch updates are applied into the graph representation. Fig. 10 shows an example of performing edge insertions on GPU-LSM. Since the array at Level 0 is filled, batch updates along with the edges at Level 0 are merged into Level 1, and then the array at Level 0 becomes empty. When inserting edges that already exist in the graph, their weights are not overwritten, which means that there may be multiple versions of edges. For an edge, only the most recently inserted one is valid and the

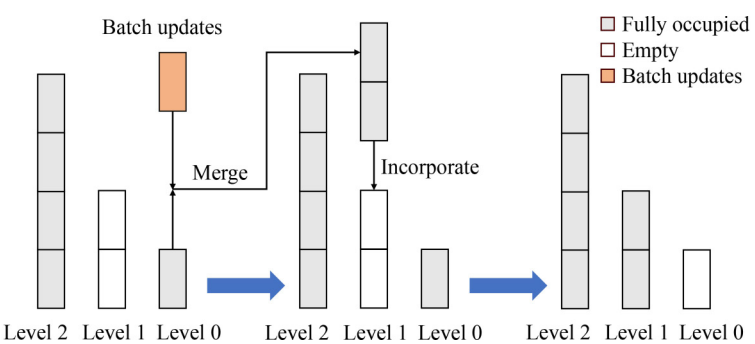

Fig. 10 The process of conducting insertions on GPU-LSM (grey parts indicate that the levels are filled, while white parts indicate empty levels)

others are all obsolete. Deleting edges is implemented by adding the edges with the weights of "null". Under such a design, edge insertions and deletions can be conducted concurrently.

5.3 Systems using CSR-based space management

1) DCSR [67]

DCSR, proposed by King et al. [67], employs a variant of CSR to store graph data. Specifically, the graph representation consists of four arrays: `row offsets`, `column indices`, `values`, and `row sizes`. Fig. 11(a) shows the graph representation of the example graph in Fig. 1(d). The `column indices` and `values` arrays are over-allocated and are separated into multiple *segments* to store edges. The over-allocated memory of these two arrays is maintained as a *memory buffer*. Each vertex in the `row offsets` array contains a fixed number of offset pairs to locate the (typically non-contiguous) segments assigned to each vertex. The elements of the `row sizes` array indicate the number of edges of each vertex. For example, in Fig. 11(a), each vertex is assigned an initial segment of length 3. The offset pair of vertex 0 stores the values (0 and 3) that locate its initial segment.

DCSR only supports edge insertions. When conducting edge insertions to a vertex, its last segment is checked. If the unoccupied space of the last segment is sufficient to incorporate new edges, they will be contiguously stored after existing edges. Otherwise, an empty segment is assigned for the vertex from the memory buffer, and the offsets of this segment are added to the `row offsets` array. Fig. 11(b) shows the process of inserting edges $(0,3,7)$ and $(0,4,5)$ to the DCSR's graph representation in Fig. 11(a). The initial segment of vertex 0 has only one empty slot, and thus cannot store these two edges. Therefore, a new segment is assigned to vertex 0, and the starting and ending offsets (15 and 18) of the segment are added to the offset pairs of vertex 0.

2) GPMA [65] and LPMA [78]

GPMA, proposed by Sha et al. [65], employs PMA [80] (Packed Memory Array, using an implicit binary tree to index array elements) to maintain the sorted arrays in CSR to store graph data. The "*original*" array in Fig. 12(a) shows an example of the PMA structure. PMA separates the array into multiple equal-length *leaf segments* (e.g., $[0,3]$), and *non-leaf segments* (e.g., $[0,7]$) are composed of their descendant segments. PMA leaves empty slots between array elements based on user-defined density thresholds (we assume that the upper density threshold of leaf and level 1 in Fig. 12(a) is 80%). Inserting an entry 27 to the PMA starts with checking the associated leaf segment (i.e., $[12,15]$), but it cannot store the new entry due to the density threshold of leaf. Then, its parent segment $[8,15]$ is inspected and is able to merge this entry while meeting the density threshold of level 1. Subsequently, the elements in $[8,15]$ are evenly re-balanced, leading to the "*balanced result*" array in Fig. 12(a). During the insertion process, it is possible to check upwards to the root segment of the PMA but still fail to store the updates. In such a situation, a double-length array has to be allocated, and the array elements are copied into the new array and re-balanced. In Fig. 12(b), the `column indices` array of CSR is maintained as a PMA, and sentinels (denoted as "S") whose offset values are stored in the `row offsets` array are used to locate the edges of each vertex.

On the basis of PMA, GPMA implements efficient parallel update operations on the GPU. GPMA designs a *lock-free* approach to alleviate the overheads of atomic operations and branch divergences in its previous *lock-based* version. In the lock-free implementation, new edges are first sorted by source and destination vertex IDs and located to the corresponding leaf segments. Then, updates are performed in a *segment-oriented* and *bottom-up* manner. Specifically, the insertion process in GPMA is carried out upwards level by level, and updates to the segments of the same level are conducted in

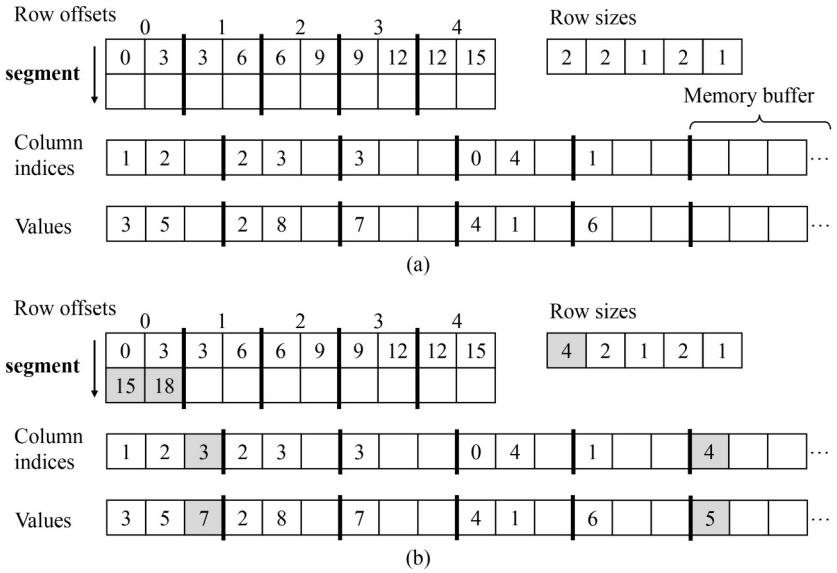

Fig. 11 The graph representation of DCSR and edge insertions to DCSR. (a) An initial DCSR's graph representation of the example graph in Fig. 1(d) (the segments are all of length 3, separated by bold lines); (b) inserting two edges (0, 3, 7) and (0, 4, 5) to the DCSR in Fig. 11(a) (modifications to the initial DCSR are highlighted with grey)

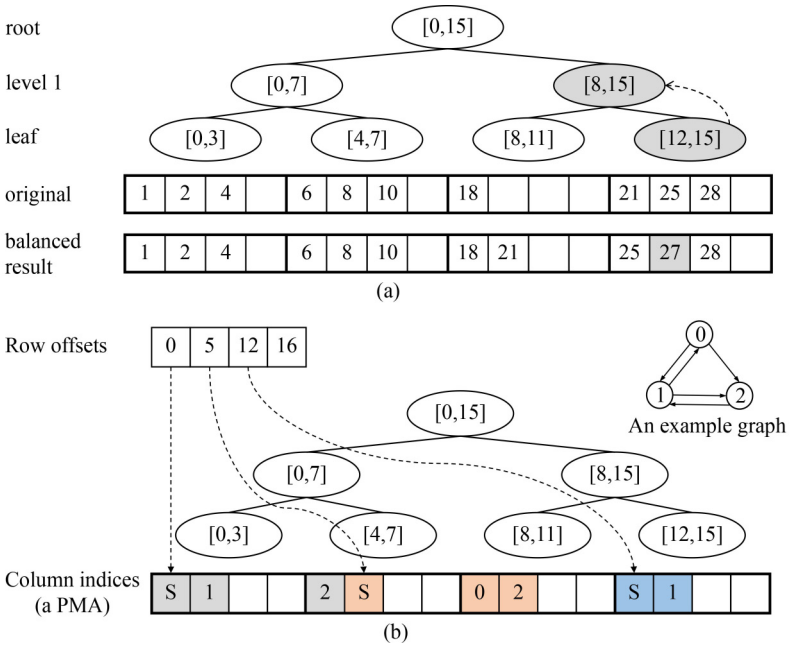

Fig. 12 An example of the basic PMA structure and a graph representation combining PMA with CSR. (a) Inserting an entry 27 to the PMA structure (the interval values in a tree node are used to locate the corresponding segment in the array; the grey part shows the process of re-balancing the PMA); (b) a graph representation that combines PMA with CSR (diverse colors indicate the neighbors and sentinels of different vertices)

parallel. Depending on the length of processed segments, GPMA designs warp-based, block-based and device-based strategies to handle update tasks.

LPMA, proposed by Zhang et al. [78], is an improvement to GPMA. LPMA separates the edge array into multi-level arrays, each of which contains a number of segments of GPMA. The length of the first-level array is equal to that of a segment of GPMA. The array at the next level is twice the size of the array at the previous level. Such a graph representation can also be regarded as a binary tree formed by the segments in multi-level arrays. According to the in-order traversal to the binary tree, the arrays of LPMA form a "linear array" so that updates can be merged by using the same strategy as GPMA. LPMA reduces the overheads of reconstructing GPMA when the length of the edge array is doubled, by only allocating a new level of segments for the tree and conducting local re-balancing among these segments rather than the entire tree.

5.4 Systems using chain-based space management

1) aimGraph [68] and faimGraph [70]

aimGraph, developed by Winter et al. [68], uses a block-based adjacency list to store graph data. The core idea of aimGraph is to allocate a large initial memory block for graph data, and then the memory management is autonomously performed on the GPU, requiring no CPU intervention unless the initial memory is exhausted. Nevertheless, aimGraph only supports edge insertions and deletions and does not propose an efficient strategy for recycling edge blocks.

faimGraph [70] is an improved version of aimGraph that adds support for modifying vertex data and uses two queue structures for memory reclamation. The memory layout of faimGraph is shown in Fig. 13. In faimGraph, the edges of each vertex are stored in a *page-based* (a page contains multiple edges, linked by page indices) linked list. Each vertex is stored in a *vertex data block* with several vertex properties. *vertex queue* and *page queue* that store vertex and page indices respectively, are used to recycle freed vertex data blocks and pages. *memory manager* is employed to keep track of the unused GPU memory. To add a vertex, faimGraph first queries the vertex and page queues to obtain a vertex index and an initial page index. If either of these queues is empty, the memory manager provides a new vertex data block or page. When deleting a vertex, all pages of the vertex should be freed and their indices are pushed to the page queue, and the vertex index is also pushed to the vertex queue. faimGraph separately conducts edge insertions and deletions. For edge insertions, if the last page of the vertex is inadequate to merge updates, the page queue is first queried to obtain an available page index. If the queue is empty, a new page has to be supplied by the memory manager. Edge deletions are conducted explicitly by replacing each edge to be deleted with an edge at the end of the linked list.

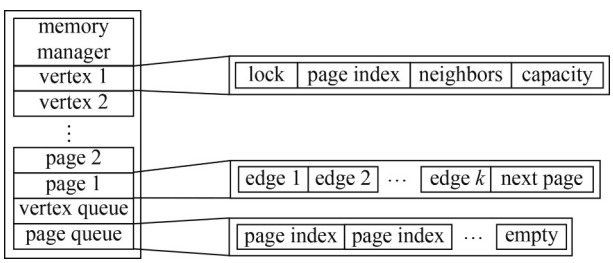

Fig. 13 Data structures used in faimGraph, including vertex data blocks, pages, a vertex queue, a page queue, and a memory manager

2) Slabhash [71]

Slabhash, proposed by Awad et al. [71], employs a hash

table [81] to store the edges of each vertex, and vertices are implicitly stored in a vertex array with pointers to their hash tables, as shown in Fig. 14. The vertex array is typically over-allocated to efficiently accommodate new vertices. A hash table is initialized with multiple *buckets* (i.e., edge blocks) based on the initial graph scale, and edges are stored in the corresponding buckets according to the hash function. When hash collisions occur and the buckets are filled, *slabs* (i.e., edge blocks that typically vary in size from buckets) are allocated and linked to buckets, forming *slab lists*.

Slabhash supports inserting and deleting vertices and edges. During vertex insertions, when the capacity of the vertex array is exhausted, a new array is allocated and the pointers to hash tables are shallow copied to the new array. When removing vertices, the hash tables of the vertices need to be reclaimed. For edge insertions, new edges are located in the corresponding bucket according to the hash function. If the slab list of the bucket is insufficient to merge updates, a new slab will be allocated and linked to the end of the slab list. Edge deletions are performed implicitly by only marking the edges to be deleted, i.e., replacing their edge weights with "null". These marked edges need to be regularly removed from the data structure to minimize memory footprint.

5.5 Discussion

In the above discussion, for the surveyed systems, we introduced their graph representations and strategies for applying updates towards a graph stored in a GPU. In the following, we summarize the design differences between these systems in each category.

1) **Systems using array-based space management** cuSTINGER [69], Hornet [72], and GPU-LSM [77] rely on array-based space management to absorb incoming updates. This type of method, however, leads to frequent movements of graph data between CPUs and GPUs, as well as frequent GPU memory allocation. In cuSTINGER, we note that there are high context-switching overheads incurred by the frequent adjustments of GPU memory from the CPU side. In addition, an empty array is not reclaimed after copying all edges to the newly allocated one [69], leading to the waste of limited GPU memory and even system crashes.

Hornet (illustrated in Fig. 9) reduces the context-switching overheads of the GPU memory allocation in cuSTINGER by introducing two auxiliary data structures (*Vectorized bit tree* and *B+ tree*) that manage a series of block-arrays with different lengths, and recycles these blocks on-the-fly on the GPU side. However, the uses of these two auxiliary data structures also incur the costs of management.

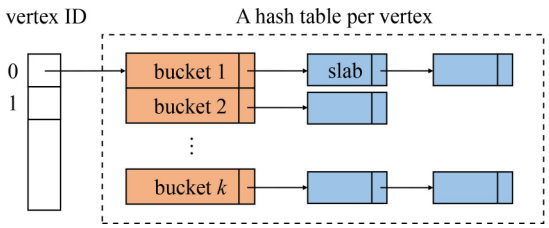

Fig. 14 An example of the graph representation in Slabhash (buckets and slabs are edge blocks of different capacities)

As for GPU-LSM (shown in Fig. 10), although it does not employ a general adjacency array representation, we classify it into this category, since insertions on GPU-LSM are implemented by copying graph data across multi-levels of arrays, which is similar to the process (data copying) in the method of array-based space management. However, GPU-LSM has more limitations and weaknesses than cuSTINGER and Hornet. Firstly, compared with linear arrays used in cuSTINGER and Hornet, GPU-LSM organizes the neighbor list of a vertex by using multi-level arrays, resulting in high overheads when accessing the graph data. Moreover, when absorbing a new update, the insertion may happen in the deep-level array, which leads to extra costs during insertion. Secondly, to accelerate the ingestion of updates, GPU-LSM simply marks an obsolete edge with a flag rather than actually removing it, which incurs high periodic cleanup overheads.

2) **Systems using CSR-based space management** DCSR [67], GPMA [65], and LPMA [78] reserve space at each neighbor list of the CSR representation (shown in Fig. 2(c)) to ingest incoming updates. Except for the reserved space at each neighbor list, DCSR (shown in Fig. 11) also reserves a large chunk of space at the end of the edge array of CSR. When absorbing updates towards a vertex, DCSR first attempts to store them in the neighbor list of the vertex. If the attempt fails, DCSR stores them in the extra space at the end of the edge array. However, when updates to this extra space occur concurrently, high atomic overheads are incurred due to mutual exclusion. Moreover, since the edges of a vertex are not contiguously stored in the edge array, extra indices are required to access these edges, which incurs overheads.

Different from DCSR, GPMA (shown in Fig. 12) reserves space at the neighbor lists of CSR according to the PMA algorithm [80], where the size of reserved space to a vertex is approximately proportional to its degree. When inserting a new neighbor to a vertex, it is stored in the reserved space of the vertex's neighbor list. However, if there is no sufficient room for the insertion, the space reserved for the vertices with close IDs of the vertex will be used. GPMA divides the edge array into multiple regions, and the neighbor lists as well as the reserved space of vertices of the same region are located in a contiguous GPU memory chunk. By this method, GPMA uses fewer indices to manage the graph data than DCSR. Nevertheless, as the spaces reserved for vertices are finite, they will be exhausted as time passes. In that case, the whole edge array needs to be reconstructed with more space for each vertex, which incurs prohibitive re-organization overheads.

LPMA employs the same space-reserving strategy as GPMA, while it separates the edge array in GPMA into multi-level arrays. LPMA reduces the re-organization and memory allocation overheads in GPMA when the edge array needs to be expanded by only allocating a new level of the array and redistributing the reserved space with this new array. However, compared to GPMA, the edges of a vertex are scattered in the multi-level arrays, which leads to inefficient memory accesses to these edges. Moreover, a specific traversal strategy to these arrays is needed to locate the edges of a vertex.

3) **Systems using chain-based space management**

aimGraph [68], faimGraph [70], and Slabhash [71] allocate new edge blocks for the insertions, where each edge block stores a pre-defined number of edges. The edge blocks are organized in GPU memory as a linked list. However, this type of method would incur frequent GPU memory allocation for edge blocks. To reduce the frequency of GPU memory allocation, aimGraph allocates a large initial chunk of GPU memory which is divided into small memory blocks for storing edges. When ingesting new edges to a vertex, empty edge blocks of the large memory chunk are assigned to the vertex to store these updates. Nevertheless, aimGraph has three issues. Firstly, it does not support modifications to vertex data. Secondly, it lacks strategies for recycling empty edge blocks after edge deletions. Thirdly, the appropriate capacity of the initial memory chunk of aimGraph is hard to predetermine. faimGraph (illustrated in Fig. 13) is an improved version of aimGraph. The improvement lies in that faimGraph addresses the first two issues in aimGraph. However, the third issue is still not solved.

Slabhash (shown in Fig. 14) improves the efficiency of querying edges (caused by the sequential search of linked lists) on faimGraph by storing the edges of a vertex in a hash table. When absorbing updates towards a vertex, edge blocks are allocated and linked to the chains of the hash table following the user-defined hash function. However, if some of the chains are excessively long, rehashing operations are necessary but incur high re-organization overheads. Besides, Slabhash is not suited to conduct graph updating from scratch, since the hash table of a vertex would degenerate to a single chain if the vertex does not have any initial neighbors [71].

6 Dynamic graph computing systems

In this section, we review general dynamic graph computing systems and systems for conducting specific graph algorithms on GPUs. We first classify our reviewed systems based on our taxonomy in Section 4.4. Then we present the typical systems of recomputing and incremental computing, respectively. Moreover, we discuss the design differences of the systems in each category.

6.1 Typical systems and classification

We survey typical dynamic graph computing systems on the GPU, including Evograph [44], the system of Makkar et al. [74], the system of Zhang et al. [82] and the system of Guo et al. [75] in 2017, the system of Tripathy et al. [83] in 2018, the system of Tödling et al. [84] in 2019, HyPR [85] in 2020, and Khanda et al. [86] in 2022. In Table 4, we categorize these systems according to our taxonomy (Fig. 8). We classify the systems of Tripathy et al. and Tödling et al. into the type of recomputing. And the rest of the systems are categorized into the type of incremental computing. The following subsections discuss the working principle of each system.

Table 4 A classification of GPU-based dynamic graph computing systems

Methods	Typical systems
recomputing	Tripathy et al. [83], Tödling et al. [84]
incremental computing	Evograph [44], Makkar et al. [74], Zhang et al. [82], HyPR [85], Guo et al. [75], Khanda et al. [86]

6.2 Systems using recomputing methods

1) Tripathy et al. [83]

Tripathy et al. [83] presented a recomputing graph computing system to conduct the k-core decomposition algorithm on the GPU. This system utilizes the graph representation used in Hornet [72] to store graph data for efficiently removing vertices and edges from the graph. This system proposes two methods to perform k-core decomposition in the graph, and both are based on the vertex peeling algorithm introduced in Section 2. The first approach employs a queue to store the peeled vertices in each iteration for a specific value of k, and an auxiliary graph structure to accommodate the removed vertices and edges. With such a graph structure, which k-core an edge belongs to can be easily identified. During the vertex peeling process, vertices and edges are removed by using the efficient update operations implemented in Hornet. In the second approach, a flag array is used to mark whether vertices have been deleted without explicitly removing vertices from the graph. During the process of vertex peeling, an edge deletion is implemented by reducing the degrees of the two endpoints of the edge.

2) Tödling et al. [84]

Tödling et al. [84] designed a recomputing graph computing system for performing the BFS algorithm. This system is combined with faimGraph [70] to store graph data and ingest updates. Compared to the naive implementation discussed in Section 2, this system optimizes two aspects of the parallel computing of the BFS algorithm on the GPU: work efficiency and workload distribution. For the former aspect, this system uses an explicit frontier queue to facilitate identifying the vertices that need to be examined in each iteration. Exclusive prefix-sum operations are used to calculate the offsets of the vertices handled by each thread at the frontier queue, and then the vertices can be added to the frontier queue with fewer atomic operations. Atomic compare-and-swap operations are used to check the state of vertices to avoid repeatedly discovering the same vertex. For the latter aspect, to ensure the load balance among threads, the frontier vertices to be processed are first classified and then assigned different numbers of threads according to their degrees. To this end, two strategies are designed for classification: deferred classification and immediate classification, which are conducted in a separate pre-processing step, or immediately when new vertices are discovered. The latter strategy proves to have better performance.

6.3 Systems using incremental computing methods

1) Evograph [44]

Evograph proposed by Sengupta et al. [44] is a general incremental graph computing system on GPUs. Evograph employs CSR to store previous graph data and uses edge lists to store graph updates. Evograph develops *I-GAS* (Incremental-Gather-Apply-Scatter) model to process the batch updates and incrementally update the analytics results of the live graph. I-GAS model identifies inconsistent vertices that are affected by updates, and generates a frontier for these inconsistent vertices. The vertices in the frontier are the only

graph data that need to be computed in one iteration. After conducting graph computing, the updates and the previous graph are merged. This paper presents the execution process of Evograph on three graph algorithms: BFS, triangle counting, and connected components.

2) Makkar et al. [74]

Makkar et al. [74] proposed an incremental graph computing system for conducting the triangle counting algorithm. This system uses the sorted version of the adjacency array implemented in cuSTINGER [69] to store graph data and uses CSR to store batch updates. The approach of neighbor intersection is used to count triangles in the graph, and the sorted adjacency array ensures efficient intersection operations. The triangle counting algorithm implemented in this system only recounts new triangles created by batch updates and obtains the exact number of triangles by using the inclusion-exclusion formula, avoiding performing neighbor intersection on all edges of affected vertices.

This system identifies three types of new triangles generated by batch updates: the triangle formed by three new edges (denoted as Δ_3), two new edges and one original edge (denoted as Δ_2), or one new edge and two original edges (denoted as Δ_1). These three types of triangles can be obtained by intersecting the neighbors (in different graph data) of two vertices of a new edge. Specifically, the neighbor intersection for Δ_3 is conducted on the neighbors of two vertices of a new edge in the graph constituted by batch updates (called update-graph). To obtain Δ_2, the neighbor intersection is performed on the neighbors of one vertex in the update-graph and those of another vertex in the live graph. The neighbors of two vertices of a new edge in the live graph are intersected to get Δ_1. Given that a triangle is identified multiple times in this method, the exact number of new triangles is computed by using an inclusion-exclusion formula for Δ_1, Δ_2, and Δ_3. And the number of triangles in the live graph is equal to the sum of the previous result and the result of the inclusion-exclusion formula.

3) Zhang et al. [82] and HyPR [85]

Zhang et al. [82] designed an incremental PageRank computing system. It is a hybrid system that utilizes GPU and CPU computing resources. The incremental PageRank algorithm starts with computing iterations on the affected graph data and produces intermediate results. The final results are then computed by combining the previous PR values and intermediate results. The PageRank algorithm is conducted in a *UGAS* (Update-Gather-Apply-Scatter) model, where the update phase extracts a list of affected vertices for subsequent GAS computations. The CPU and GPU are synchronized by using the BSP model. And the strategies of load balancing and message communication between CPU and GPU derive from the previous work [87].

HyPR, developed by Giri et al. [85], is also a hybrid CPU-GPU system for incrementally conducting the PageRank algorithm. The algorithm in this system follows [88], which identifies three types of vertices in the live graph: *new vertices*, *old vertices*, and *border vertices*, representing the new vertices in batch updates, unaffected vertices, and affected vertices in the original graph. In HyPR, when batch updates arrive, the CPU host partitions the live graph into three subgraphs that contain new, old, and border vertices, and then these three subgraphs are transferred to GPU memory for computation. According to [88], to incrementally update the results on the live graph, scaling operations (i.e., scaling PR values by a factor proportional to the expansion of vertices) are performed on the old and border vertices in parallel, followed by computing new PR values for the new and border vertices in parallel.

4) Guo et al. [75]

Guo et al. [75] designed an incremental computing system to conduct the Personalized PageRank (PPR) algorithm on the GPU. The PPR algorithm in this system adopts the method of parallel local update scheme [55] and obtains approximate PPR values of vertices. The scheme maintains two values for each vertex v: $P_s(v)$ denoting the estimate PPR value, and $R_s(v)$ denoting the estimate bias (called *residual*, with an upper bound of ϵ) to the accurate value.

Parallel local update scheme consists of two steps: *restore invariants* and *local pushes*. After applying batch updates to graph data, the current residual of each affected vertex is recomputed according to the invariant equation in local update scheme. If any recomputed residuals exceed ϵ, the vertices are pushed into a frontier queue, and then the process of local pushes is invoked to update their estimate values. Local pushes are an iterative process, including two parallel steps: *self-update* and *neighbor-propagation*, which update estimate values of the frontier vertices with a portion of their residuals in parallel and propagate the remaining residuals to their neighbors by using atomic operations. If any residuals still exceed ϵ, the vertices are added to the frontier queue and repeat local pushes until the queue is empty. After that, the estimate PPR values of vertices are output as the result of the live graph.

Given that the results on some vertices require extra local pushes to converge, caused by ignoring the residuals pushed to themselves before propagating residuals to their neighbors, this system optimizes the process of local pushes. Specifically, it reverses the order of self-update and neighbor-propagation by first performing residual propagation to the neighbors of frontier vertices and then updating their estimate values. Besides, a comparison of the before-value and after-value of residuals during residual propagation prevents adding the same vertex to the frontier queue multiple times.

5) Khanda et al. [86]

Khanda et al. [86] designed an incremental graph computing system to conduct the SSSP algorithm under edge insertions and edges deletions on GPUs. The dynamic SSSP algorithm consists of two steps. In the first step, the vertices (called affected vertices) affected by edge updates are identified. Edge insertions will lead to changes to the parent of affected vertices, while edge deletions will result in some subtrees to be disconnected from the SSSP tree. Only the edge deletions that affect the structure of the SSSP tree will be further processed. In the second step, incremental computation is conducted on the affected vertices to update the SSSP tree. At

first, all descendants of the vertices affected by deletions in the SSSP tree are also marked as "affected" and reset their shortest-path values to infinity. Then, the shortest-path values of all affected vertices are updated by comparing the shortest-path value of an affected vertex with that of its neighbors. Specifically, if the shortest-path value of an affected vertex is reduced by passing one of its neighbors, its value will be updated. If the shortest-path value of a neighbor is reduced by passing an affect vertex, the neighbor's shortest-path value will also be updated. In the implementation, this system designs a strategy named *Vertex-marking functional block* to reduce the atomic overheads of concurrently updating the shortest-path values of vertices.

6.4 Discussion

In the above introduction, for the reviewed systems, we introduce their designs of dynamic graph algorithms. In the following, we summarize the design differences between these systems in each category.

1) **Systems using recomputing methods** Tripathy et al. [83] and Tödling et al. [84] conduct graph algorithms from scratch on the live graph data, without the need to process previous graph data. These two systems are both built on existing dynamic graph updating systems (Hornet and faimGraph respectively) to ensure the efficiency of the graph updating process on GPUs. Based on vertex peeling [37], Tripathy et al. [83] propose two improved k-core decomposition algorithms for dynamic graphs. However, the system that implements these two k-core decomposition algorithms lacks the strategies for balancing the load among threads on GPUs and reducing atomic overheads when conducting vertex peeling in parallel. Tödling et al. [84] design a BFS algorithm for dynamic graphs. This work not only designs load-balancing strategies but also reduces the atomic overheads when pushing newly discovered vertices to the frontier queue.

2) **Systems using incremental computing methods** Evograph [44], Makkar et al. [74], Zhang et al. [82], HyPR [85], Guo et al. [75], Khanda et al. [86] conduct incremental computing on the live graph data by combining the previous analytics results with the intermediate results computed from the subgraph induced by the vertices (called affected vertices) affected by the updates. In these systems, the key challenge lies in reducing the effort of computing by reusing the previous results to produce the most recent results.

Among the systems we reviewed in this category, Evograph [44] is currently the only general GPU-based graph computing system that proposes an incremental computing model and supports multiple graph algorithms, including BFS, triangle counting, and connected components. Conversely, other systems all design incremental computing methods for a specific algorithm. Makkar et al. [74] design an incremental computing system to conduct the triangle counting algorithm. Different from other systems that use general static graph representations (e.g., CSR) to store graph data, this system employs the graph representation designed in an existing graph updating system (cuSTINGER). Moreover, this system reduces the computation of neighbor intersections in the previous system [89] by only conducting neighbor intersections on the newly inserted edges, while this system does not propose any load-balancing strategy. Zhang et al. [82], HyPR [85] both design PageRank algorithms conducted on hybrid CPU-GPU systems. However, the first system conducts computing tasks on both CPU and GPU with a series of strategies on workload distribution, message communication, and synchronization, while the second system only partitions the live graph on the CPU, and all subgraphs are computed on the GPU. Guo et al. [75] design an incremental computing system to perform the Personalized PageRank algorithm. This system reduces the number of iterations to converge the Personalized PageRank algorithm by redesigning the two key steps in the algorithm: self-update and neighbor-propagation, while this system also lacks the strategy for load balancing. Khanda et al. [86] design a system to incrementally perform the SSSP algorithm on dynamic graphs. This system first identifies the vertices affected by updates, and then updates the results of the neighbors with the result of an affected vertex or updates the result of the affected vertex with the result of one of its neighbors. This system designs a strategy of Vertex-marking functional block to reduce atomic overheads and improve its computing performance on GPUs.

7 Related work

To the best of our knowledge, this paper is the first survey to discuss dynamic graph processing on GPUs. To differentiate our study from previous reviews, we extensively review related surveys on dynamic graphs.

Currently, there is a great research interest in reviewing algorithms designed for dynamic graphs. Hanauer et al. [90] surveyed various classical graph algorithms, e.g., connected components and shortest paths, while Fournier-Viger et al. [91] specifically reviewed pattern mining algorithms on dynamic graphs. Rossetti and Cazabet [62] described community discovery problems on dynamic graphs and proposed a taxonomy of dynamic community discovery algorithms. Aggarwal and Subbian [46] reviewed graph algorithms of both online and offline analytics, and related applications. Another direction of research is reviewing the models for representing dynamic graphs. Zaki et al. [40] theoretically summarized the models for storing historical graph data, including snapshot sequences, temporal graphs, and log files, and presented typical systems of these models. Holme and Saramäki [63,64] introduced the models, analytics, and various applications of temporal graphs. McGregor [49], Zhang [52], and O'Connell [92] reviewed the graph algorithms under the stream graph model, which are designed for the cases where memory is insufficient to store the whole graph data. Besides, there are also several surveys on extended studies of dynamic graphs. Skarding et al. [93] focused on the recent studies on dynamic graph neural networks. Kazemi et al. [94] surveyed the research advance of representation learning on dynamic graphs. Their study presents various models of representation learning from an encoder-decoder aspect and classifies encoders and decoders.

Besta et al. [95] developed the only survey of online dynamic graph processing systems. However, their study mainly reviews the works on CPUs, lacking a discussion of

the dynamic graph processing workflow on GPUs and a detailed introduction to dynamic graph processing systems on GPUs. Our survey fills this gap by presenting the research advance of GPU-based dynamic graph processing systems.

8 Conclusion and future research direction

With their massive parallel processing power, GPUs have emerged as a viable option for conducting dynamic graph processing tasks, particularly on large-scale dynamic graphs. This paper presents a comprehensive survey of existing works on dynamic graph processing on GPUs. We introduce key concepts and terminologies of dynamic graph processing, including dynamic graph analytics and models, and present the workflow of dynamic graph processing on GPUs. Furthermore, we review typical GPU-based graph updating and computing systems and propose a taxonomy of these systems based on the methods they adopt to ingest updates and conduct graph analytics. According to this taxonomy, we classify these systems and discuss their design differences.

During our discussion on dynamic graph processing systems on GPUs, we note that several existing problems need to be addressed by today's dynamic graph processing systems:

● **Not reserving appropriate space for different vertices according to update frequencies** From the aspect of graph updating, the three types of methods all reserve space for the neighbor lists of vertices to facilitate edge insertions, effectively reducing the overheads of data structure reconstruction and memory reallocation. However, existing graph updating systems on GPUs do not reserve appropriate space for edges based on their update frequency, which brings about two issues. Firstly, reserving excessive space for infrequently updated vertices will lead to the waste of limited GPU memory. Secondly, reserving insufficient space for vertices with high update frequency will result in the overheads of frequent memory reallocation.

● **Lacking criteria for choosing recomputing or incremental computing** As discussed in Section 4, the methods of recomputing and incremental computing have inherent advantages and disadvantages. An ideal dynamic graph computing system should adaptively select one of these two methods in response to different sizes of incoming updates to achieve high computing performance for specific graph algorithms. However, existing graph computing systems on GPUs design either recomputing or incremental computing graph algorithms, lacking explicit criteria for choosing one of them under different update situations.

● **Facing a trade-off between the efficiencies of graph updating and computing** To ensure the update efficiency, existing dynamic graph updating systems generally do not sort edges by source and destination vertex IDs. However, dynamic graph computing systems require sorted graph representations to efficiently access edges when conducting graph algorithms. This creates a trade-off between update efficiency and computing efficiency that need to be carefully considered to meet practical demands.

There are also a number of promising future research directions in the area of dynamic graph processing on GPUs. We briefly enumerate and discuss some of them below:

● **Migrating CPU-based dynamic graph processing systems to the GPU platform** Efficient CPU-based systems can be extended to conduct the workloads of dynamic graph processing on GPUs. cuSTINGER and GPMA are representatives that migrate CPU-based systems STINGER and PMA to the GPU platform. Such migrations involve optimizations for the computational and memory access features of GPUs.

● **Applying GPU-oriented efficient data structures to the field of dynamic graphs** Various advanced data structures have been developed for GPUs, such as hash tables [96,97] and B-trees (B+trees) [98,99]. These data structures can be employed to design graph representations for dynamic graphs. In order to meet the demands of graph updating and graph computing on dynamic graphs, these data structures should support efficient modifications, including insertions and deletions, as well as queries.

● **Conducting offline analytics workloads of dynamic graphs on GPUs** Existing GPU-based dynamic graph processing systems focus on accelerating online analytics tasks by using GPUs. However, limited by the high time costs of transferring large amounts of historical graph data between the CPU and GPU, it is challenging to conduct efficient offline analytics on GPUs. In our investigation, the system proposed by Zhang et al. [100] in 2022 is the only one that addresses this issue by exploiting the similarity of different snapshots to reduce the overheads of CPU-GPU data transfer and improve the efficiency of offline analytics on GPUs. It is of great research value to study offline processing systems on GPUs in the future.

● **Dynamic graph processing systems in CPU-GPU hybrid computing environments** Among the existing dynamic graph processing systems we reviewed, only Zhang et al. [82] and HyPR [85] use CPUs to conduct graph computing workloads. In contrast, other systems only employ CPUs to control the process of dynamic graph processing, with their primary function being to transfer graph data to GPU devices and receive analytics results generated by GPUs. Consequently, the computing power of CPUs is under-utilized. In the future, it is worthwhile to study hybrid computing systems for CPU-GPU dynamic graph co-processing. Key challenges in this direction lie in computing task assignment and workload balancing between CPUs and GPUs.

● **Multi-GPU dynamic graph processing systems** Existing GPU-based dynamic graph processing systems are designed only on a single GPU, with little work on multiple GPUs. However, due to the limited GPU memory, single-GPU systems may struggle to efficiently store and process large-scale graphs that exceed GPU memory limits. Hence, designing multi-GPU systems is a promising research direction, as they can provide higher processing power and larger storage space than single-GPU systems. Techniques involved in this design include efficient graph partitioning and strategies for reducing communication and synchronization overheads among multiple GPUs.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61972444, 61825202, 62072195, and 61832006). This work was also supported by Zhejiang Lab (2022P10AC02).

References

1. Shi X, Zheng Z, Zhou Y, Jin H, He L, Liu B, Hua Q S. Graph processing on GPUs: a survey. ACM Computing Surveys, 2018, 50(6): 81
2. Li B, Gao S, Liang Y, Kang Y, Prestby T, Gao Y, Xiao R. Estimation of regional economic development indicator from transportation network analytics. Scientific Reports, 2020, 10(1): 2647
3. Alkhamees M, Alsaleem S, Al-Qurishi M, Al-Rubaian M, Hussain A. User trustworthiness in online social networks: a systematic review. Applied Soft Computing, 2021, 103: 107159
4. Karamati S, Young J, Vuduc R. An energy-efficient single-source shortest path algorithm. In: Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium. 2018, 1080−1089
5. Yang J, McAuley J, Leskovec J. Community detection in networks with node attributes. In: Proceedings of the 13th International Conference on Data Mining. 2013, 1151−1156
6. Zhong J, He B. Medusa: simplified graph processing on GPUs. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(6): 1543–1552
7. Ammar K. Techniques and systems for large dynamic graphs. In: Proceedings of 2016 on SIGMOD'16 PhD Symposium. 2016, 7−11
8. Brailovskaia J, Margraf J. The relationship between active and passive Facebook use, Facebook flow, depression symptoms and Facebook addiction: a three-month investigation. Journal of Affective Disorders Reports, 2022, 10: 100374
9. Muin M A, Kapti K, Yusnanto T. Campus website security vulnerability analysis using Nessus. International Journal of Computer and Information System, 2022, 3(2): 79–82
10. Gowda S R S, King R, Kumar M R P. Real-time tweets streaming and comparison using naïve Bayes classifier. In: Proceedings of the 3rd International Conference on Data Science, Machine Learning and Applications. 2023, 103−110
11. Qiu X, Cen W, Qian Z, Peng Y, Zhang Y, Lin X, Zhou J. Real-time constrained cycle detection in large dynamic graphs. Proceedings of the VLDB Endowment, 2018, 11(12): 1876–1888
12. Ye C, Li Y, He B, Li Z, Sun J. GPU-accelerated graph label propagation for real-time fraud detection. In: Proceedings of 2021 International Conference on Management of Data. 2021, 2348−2356
13. Kent A D, Liebrock L M, Neil J C. Authentication graphs: analyzing user behavior within an enterprise network. Computers & Security, 2015, 48: 150–166
14. Wheatman B, Xu H. Packed compressed sparse row: a dynamic graph representation. In: Proceedings of 2018 IEEE High Performance Extreme Computing Conference. 2018, 1−7
15. Kumar P, Huang H H. GraphOne: a data store for real-time analytics on evolving graphs. ACM Transactions on Storage, 2019, 15(4): 29
16. Zhu X, Feng G, Serafini M, Ma X, Yu J, Xie L, Aboulnaga A, Chen W. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. Proceedings of the VLDB Endowment, 2020, 13(7): 1020–1034
17. De Leo D, Boncz P. Teseo and the analysis of structural dynamic graphs. Proceedings of the VLDB Endowment, 2021, 14(6): 1053–1066
18. Cheng R, Hong J, Kyrola A, Miao Y, Weng X, Wu M, Yang F, Zhou L, Zhao F, Chen E. Kineograph: taking the pulse of a fast-changing and connected world. In: Proceedings of the 7th ACM European Conference on Computer Systems. 2012, 85−98
19. Shi X, Cui B, Shao Y, Tong Y. Tornado: a system for real-time iterative analysis over evolving data. In: Proceedings of 2016 International Conference on Management of Data. 2016, 417−430
20. Vora K, Gupta R, Xu G. KickStarter: fast and accurate computations on streaming graphs via trimmed approximations. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems. 2017, 237−251
21. Sheng F, Cao Q, Cai H, Yao J, Xie C. GraPU: accelerate streaming graph analysis through preprocessing buffered updates. In: Proceedings of the ACM Symposium on Cloud Computing. 2018, 301−312
22. Mariappan M, Vora K. GraphBolt: dependency-driven synchronous processing of streaming graphs. In: Proceedings of the 14th EuroSys Conference 2019. 2019, 25
23. Shi X, Luo X, Liang J, Zhao P, Di S, He B, Jin H. Frog: asynchronous graph processing on GPU with hybrid coloring model. IEEE Transactions on Knowledge and Data Engineering, 2018, 30(1): 29–42
24. Sengupta D, Sundaram N, Zhu X, Willke T L, Young J, Wolf M, Schwan K. GraphIn: an online high performance incremental graph processing framework. In: Proceedings of the 22nd International Conference on Parallel and Distributed Computing. 2016, 319−333
25. Cormen T H, Leiserson C E, Rivest R L, Stein C. Introduction to Algorithms. 3rd ed. Cambridge: MIT Press, 2009
26. Shao Z, Li R, Hu D, Liao X, Jin H. Improving performance of graph processing on FPGA-DRAM platform by two-level vertex caching. In: Proceedings of 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2019, 320−329
27. Goodrich M T, Tamassia R. Algorithm Design and Applications. Hoboken: Wiley Hoboken, 2015
28. Green O, Yalamanchili P, Munguía L M. Fast triangle counting on the GPU. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms. 2014, 1−8
29. Park S, Lee W, Choe B, Lee S G. A survey on personalized PageRank computation algorithms. IEEE Access, 2019, 7: 163049–163062
30. Boldi P, Santini M, Vigna S. PageRank as a function of the damping factor. In: Proceedings of the 14th International Conference on World Wide Web. 2005, 557−566
31. Ohsaka N, Maehara T, Kawarabayashi K I. Efficient PageRank tracking in evolving networks. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2015, 875−884
32. Brin S, Page L. The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, 1998, 30(1-7): 107–117
33. Kamvar S D, Haveliwala T H, Manning C D, Golub G H. Extrapolation methods for accelerating PageRank computations. In: Proceedings of the 12th International Conference on World Wide Web. 2003, 261−270
34. Hou G, Chen X, Wang S, Wei Z. Massively parallel algorithms for personalized PageRank. Proceedings of the VLDB Endowment, 2021, 14(9): 1668–1680
35. Mandal A, Al Hasan M. A distributed k-core decomposition algorithm on spark. In: Proceedings of 2017 IEEE International Conference on Big Data. 2017, 976−981
36. Victor F, Akcora C G, Gel Y R, Kantarcioglu M. Alphacore: data depth based core decomposition. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. 2021, 1625−1633
37. Esfandiari H, Lattanzi S, Mirrokni V S. Parallel and streaming algorithms for K-core decomposition. In: Proceedings of the 35th International Conference on Machine Learning. 2018, 1396−1405
38. Alvarez-Hamelin J I, Dall'Asta L, Barrat A, Vespignani A. Large scale

networks fingerprinting and visualization using the k-core decomposition. In: Proceedings of the 18th International Conference on Neural Information Processing Systems. 2005, 41−50

39. Zeng L, Zou L, Özsu M T, Hu L, Zhang F. GSI: GPU-friendly subgraph isomorphism. In: Proceedings of the 36th International Conference on Data Engineering. 2020, 1249−1260
40. Zaki A, Attia M, Hegazy D, Amin S. Comprehensive survey on dynamic graph models. International Journal of Advanced Computer Science and Applications, 2016, 7(2): 573–582
41. Li D, Li W, Chen Y, Lin M, Lu S. Learning-based dynamic graph stream sketch. In: Proceedings of the 25th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. 2021, 383−394
42. Margan D, Pietzuch P. Large-scale stream graph processing: doctoral symposium. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. 2017, 378−381
43. Harary F, Gupta G. Dynamic graph models. Mathematical and Computer Modelling, 1997, 25(7): 79–87
44. Sengupta D, Song S L. EvoGraph: on-the-fly efficient mining of evolving graphs on GPU. In: Proceedings of the 32nd International Conference on High Performance Computing. 2017, 97−119
45. Iyer A P, Pu Q, Patel K, Gonzalez J E, Stoica I. TEGRA: efficient Ad-Hoc analytics on evolving graphs. In: Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation. 2021, 337−355
46. Aggarwal C, Subbian K. Evolutionary network analysis: a survey. ACM Computing Surveys, 2014, 47(1): 10
47. Van Vlasselaer V, Akoglu L, Eliassi-Rad T, Snoeck M, Baesens B. Guilt-by-constellation: fraud detection by suspicious clique memberships. In: Proceedings of the 48th Hawaii International Conference on System Sciences. 2015, 918−927
48. Xu S, Liao X, Shao Z, Hua Q, Jin H. Maximal clique enumeration problem on graphs: status and challenges. SCIENTIA SINICA Informationis, 2022, 52(5): 784–803
49. McGregor A. Graph stream algorithms: a survey. ACM SIGMOD Record, 2014, 43(1): 9–20
50. Vora K, Gupta R, Xu G. Synergistic analysis of evolving graphs. ACM Transactions on Architecture and Code Optimization, 2016, 13(4): 32
51. Sheng F, Cao Q, Yao J. Exploiting buffered updates for fast streaming graph analysis. IEEE Transactions on Computers, 2021, 70(2): 255–269
52. Zhang J. A survey on streaming algorithms for massive graphs. In: Aggarwal C C, Wang H X, eds. Managing and Mining Graph Data. New York: Springer, 2010, 393−420
53. Bar-Yossef Z, Kumar R, Sivakumar D. Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms. 2002, 623−632
54. Zhao P, Aggarwal C C, Wang M. gSketch: on query estimation in graph streams. Proceedings of the VLDB Endowment, 2011, 5(3): 193–204
55. Zhang H, Lofgren P, Goel A. Approximate personalized PageRank on dynamic graphs. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016, 1315−1324
56. Shin K, Oh S, Kim J, Hooi B, Faloutsos C. Fast, accurate and provable triangle counting in fully dynamic graph streams. ACM Transactions on Knowledge Discovery from Data, 2020, 14(2): 12
57. Basak A, Lin J, Lorica R, Xie X, Chishti Z, Alameldeen A, Xie Y. SAGA-bench: software and hardware characterization of streaming graph analytics workloads. In: Proceedings of 2020 IEEE International Symposium on Performance Analysis of Systems and Software. 2020, 12−23
58. Ren C, Lo E, Kao B, Zhu X, Cheng R. On querying historical evolving graph sequences. Proceedings of the VLDB Endowment, 2011, 4(11): 726–737
59. Khurana U, Deshpande A. Efficient snapshot retrieval over historical graph data. In: Proceedings of the 29th International Conference on Data Engineering. 2013, 997−1008
60. Han W, Miao Y, Li K, Wu M, Yang F, Zhou L, Prabhakaran V, Chen W, Chen E. Chronos: a graph engine for temporal graph analysis. In: Proceedings of the 9th European Conference on Computer Systems. 2014, 1
61. Steer B, Cuadrado F, Clegg R. Raphtory: streaming analysis of distributed temporal graphs. Future Generation Computer Systems, 2020, 102: 453–464
62. Rossetti G, Cazabet R. Community discovery in dynamic networks: a survey. ACM Computing Surveys, 2019, 51(2): 35
63. Holme P. Modern temporal network theory: a colloquium. The European Physical Journal B, 2015, 88(9): 234
64. Holme P, Saramäki J. Temporal networks. Physics Reports, 2012, 519(3): 97−125
65. Sha M, Li Y, He B, Tan K L. Accelerating dynamic graph analytics on GPUs. Proceedings of the VLDB Endowment, 2017, 11(1): 107–120
66. Mariappan M, Che J, Vora K. DZiG: sparsity-aware incremental processing of streaming graphs. In: Proceedings of the 16th European Conference on Computer Systems. 2021, 83−98
67. King J, Gilray T, Kirby R M, Might M. Dynamic sparse-matrix allocation on GPUs. In: Proceedings of the 31st International Conference on High Performance Computing. 2016, 61−80
68. Winter M, Zayer R, Steinberger M. Autonomous, independent management of dynamic graphs on GPUs. In: Proceedings of 2017 IEEE High Performance Extreme Computing Conference. 2017, 1−7
69. Green O, Bader D A. cuSTINGER: supporting dynamic graph algorithms for GPUs. In: Proceedings of 2016 IEEE High Performance Extreme Computing Conference. 2016, 1−6
70. Winter M, Mlakar D, Zayer R, Seidel H P, Steinberger M. faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In: Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. 2018, 754−766
71. Awad M A, Ashkiani S, Porumbescu S D, Owens J D. Dynamic graphs on the GPU. In: Proceedings of 2020 IEEE International Parallel and Distributed Processing Symposium. 2020, 739−748
72. Busato F, Green O, Bombieri N, Bader D A. Hornet: an efficient data structure for dynamic sparse graphs and matrices on GPUs. In: Proceedings of 2018 IEEE High Performance extreme Computing Conference. 2018, 1−7
73. Ediger D, McColl R, Riedy J, Bader D A. STINGER: high performance data structure for streaming graphs. In: Proceedings of 2012 IEEE Conference on High Performance Extreme Computing. 2012, 1−5
74. Makkar D, Bader D A, Green O. Exact and parallel triangle counting in dynamic graphs. In: Proceedings of the 24th International Conference on High Performance Computing. 2017, 2−12
75. Guo W, Li Y, Sha M, Tan K L. Parallel personalized PageRank on dynamic graphs. Proceedings of the VLDB Endowment, 2017, 11(1): 93–106
76. Jaiyeoba W, Skadron K. GraphTinker: a high performance data structure for dynamic graph processing. In: Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium. 2019, 1030−1041
77. Ashkiani S, Li S, Farach-Colton M, Amenta N, Owens J D. GPU LSM: a dynamic dictionary data structure for the GPU. In: Proceedings of 2018 IEEE International Parallel and Distributed Processing

Symposium. 2018, 430−440
78. Zhang F, Zou L, Yu Y. LPMA - an efficient data structure for dynamic graph on GPUs. In: Proceedings of the 22nd International Conference on Web Information Systems Engineering 2021. 2021, 469−484
79. Ediger D, Riedy J, Bader D A, Meyerhenke H. Computational graph analytics for massive streaming data. In: Sarbazi-Azad H, Zomaya A Y, eds. Large Scale Network-Centric Distributed Systems. Hoboken: John Wiley & Sons, Inc., 2013, 619−648
80. Bender M A, Hu H. An adaptive packed-memory array. ACM Transactions on Database Systems, 2007, 32(4): 26–es
81. Ashkiani S, Farach-Colton M, Owens J D. A dynamic hash table for the GPU. In: Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium. 2018, 419−429
82. Zhang T. Efficient incremental PageRank of evolving graphs on GPU. In: Proceedings of 2017 International Conference on Computer Systems, Electronics and Control. 2017, 1232−1236
83. Tripathy A, Hohman F, Chau D H, Green O. Scalable K-core decomposition for static graphs using a dynamic graph data structure. In: Proceedings of 2018 IEEE International Conference on Big Data. 2018, 1134−1141
84. Tödling D, Winter M, Steinberger M. Breadth-first search on dynamic graphs using dynamic parallelism on the GPU. In: Proceedings of 2019 IEEE High Performance Extreme Computing Conference. 2019, 1−7
85. Giri H K, Haque M, Banerjee D S. HyPR: hybrid page ranking on evolving graphs. In: Proceedings of the 27th International Conference on High Performance Computing, Data, and Analytics. 2020, 62−71
86. Khanda A, Srinivasan S, Bhowmick S, Norris B, Das S K. A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(4): 929–940
87. Zhang T, Zhang J, Shu W, Wu M Y, Liang X. Efficient graph computation on hybrid CPU and GPU systems. The Journal of Supercomputing, 2015, 71(4): 1563–1586
88. Desikan P, Pathak N, Srivastava J, Kumar V. Incremental page rank computation on evolving graphs. In: Proceedings of the Special Interest Tracks and Posters of the 14th International Conference on World Wide Web. 2005, 1094−1095
89. Ediger D, Jiang K, Riedy J, Bader D A. Massive streaming data analytics: a case study with clustering coefficients. In: Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum. 2010, 1−8
90. Hanauer K, Henzinger M, Schulz C. Recent advances in fully dynamic graph algorithms. In: Proceedings of the 1st Symposium on Algorithmic Foundations of Dynamic Networks. 2022, 1.11
91. Fournier-Viger P, He G, Cheng C, Li J, Zhou M, Lin J C W, Yun U. A survey of pattern mining in dynamic graphs. WIREs Data Mining and Knowledge Discovery, 2020, 10(6): e1372
92. O'Connell T C. A survey of graph algorithms under extended streaming models of computation. In: Ravi S S, Shukla S K, eds. Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz. Dordrecht: Springer, 2009, 455−476
93. Skarding J, Gabrys B, Musial K. Foundations and modeling of dynamic networks using dynamic graph neural networks: a survey. IEEE Access, 2021, 9: 79143–79168
94. Kazemi S M, Goel R, Jain K, Kobyzev I, Sethi A, Forsyth P, Poupart P. Representation learning for dynamic graphs: a survey. The Journal of Machine Learning Research, 2020, 21(1): 70
95. Besta M, Fischer M, Kalavri V, Kapralov M, Hoefler T. Practice of streaming processing of dynamic graphs: concepts, models, and systems. IEEE Transactions on Parallel and Distributed Systems, 2021
96. Ren Z, Gu Y, Li C, Li F, Yu G. GPU-based dynamic hyperspace hash with full concurrency. Data Science and Engineering, 2021, 6(3): 265–279
97. Green O. HashGraph-scalable hash tables using a sparse graph data structure. ACM Transactions on Parallel Computing, 2021, 8(2): 11
98. Awad M A, Ashkiani S, Johnson R, Farach-Colton M, Owens J D. Engineering a high-performance GPU B-tree. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. 2019, 145−157
99. Yan Z, Lin Y, Peng L, Zhang W. Harmonia: a high throughput B+tree for GPUs. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. 2019, 133−144
100. Zhang Y, Liang Y, Zhao J, Mao F, Gu L, Liao X, Jin H, Liu H, Guo S, Zeng Y, Hu H, Li C, Zhang J, Wang B. EGraph: efficient concurrent GPU-based dynamic graph processing. IEEE Transactions on Knowledge and Data Engineering, 2022

Hongru Gao received the BS degree in software engineering from Huazhong University of Science and Technology (HUST), China in 2020. He is currently working toward the PhD degree. His research interests include graph computing and graph neural networks.

Xiaofei Liao received the PhD degree in computer science and technology from Huazhong University of Science and Technology (HUST), China in 2005. He is now a professor and PhD supervisor at National Engineering Research Center for Big Data Technology and System, HUST, China. His research interests are in the areas of memory computing, runtime systems, and graph computing.

Zhiyuan Shao received the PhD degree in computer science and technology from Huazhong University of Science and Technology (HUST), China in 2005. He is now a professor at National Engineering Research Center for Big Data Technology and System, HUST, China. His research interests are in the areas of graph computing, big-data processing, and computing systems.

Kexin Li received the BS degree in computer science and technology from Huazhong University of Science and Technology (HUST), China in 2020. She is currently working toward the MS degree. Her research interests include graph computing and memory-access optimization.

Jiajie Chen received the BS degree in computer science and technology from Huazhong University of Science and Technology (HUST), China in 2020. He is currently working toward the MS degree. His research interests include graph computing and network processing.

Hai Jin is a Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST), China. Jin received his PhD in computer engineering from HUST, China in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, Germany. Jin worked at The University of Hong Kong, China between 1998 and 2000, and as a visiting scholar at the University of Southern California, USA between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a Fellow of IEEE, Fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big-data processing, data storage, and system security.