IEEE Transactions on Knowledge and Data Engineering (2023).
DOI: 10.1109/TKDE.2023.3336329

Large-scale Graph Label Propagation on GPUs

Chang Ye, Yuchen Li *Member*, Bingsheng He *Member*, Zhao Li *Member*, and Jianling Sun *Member*

Abstract—Graph label propagation (LP) is a core component in many downstream applications such as fraud detection, recommendation and image segmentation. In this paper, we propose GLP, a GPU-based framework to enable efficient LP processing on large-scale graphs. By investigating the data processing pipeline in a large e-commerce platform, we have identified two key challenges on integrating GPU-accelerated LP processing to the pipeline: (1) programmability for evolving application logics; (2) demand for real-time performance. Motivated by these challenges, we offer a set of expressive APIs that data engineers can customize and deploy efficient LP algorithms on GPUs with ease. To achieve better performance, we propose novel GPU-centric optimizations by leveraging the community as well as power-law properties of large graphs. Further, we significantly reduce the expensive data transfer cost between CPUs and GPUs by enabling LP processing on compressed graphs. Extensive experiments have confirmed the effectiveness of our proposed approaches over the state-of-the-art GPU methods. Furthermore, our proposed solution supports a real billion-scale graph workload for fraud detection and achieves 13.2x speedup to the current in-house solution running on a high-end multicore machine with compressed graphs.

Index Terms—Graph, Label Propagation, GPU computing

1 INTRODUCTION

GRAPH data is pervasive as people and things are digitally connected today. The information embedded in graph data brings opportunities to discover valuable insights that continuously power the development of data-driven economy [1], [2]. In many downstream applications such as fraud detection [3] and recommendation [4], a common practice for analyzing large graphs is to identify important clusters. Among the existing graph clustering approaches, Label Propagation (LP) is one of the most effective *and* efficient algorithms [5], [6]. Despite having a linear complexity theoretically [7], the workload of executing LP on large graphs is still a major bottleneck for time-critical applications, especially in major commercial enterprises.

Fraud Detection Pipeline (Figure 1). We investigated the data processing pipeline for fraud detection in TAOBAO, one of the largest e-commerce platform. The LP algorithm and its variants are used as a core component to analyze TAOBAO's transaction networks, which include purchases/click information. The networks are first processed by LP to identify suspicious clusters from known black-listed users. Subsequently, the identified clusters are then analyzed by more sophisticated algorithms, e.g., graph neural nets [8], to discover new frauds. It is worth-noting that the run-time of the LP component occupies 75% overhead of the automated detection pipeline. Fraud detection is a pressing challenge today that the number of fraud transactions is around 7.2% of the overall search volume daily. Hence, speeding up the LP component will drastically improve the detection latency and enable more responsive measures to stop financial losses.

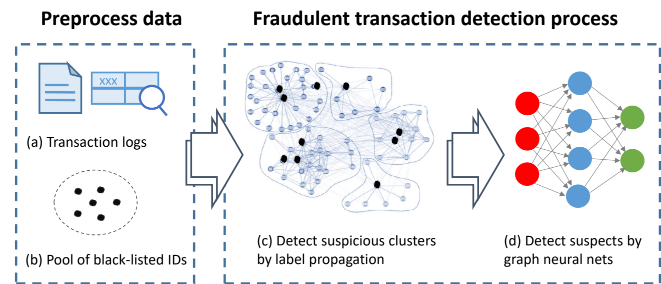

Fig. 1. Popular data pipeline for fraud detection.

GPU-accelerated Graph Processing. Recently, there are rapid growing interests in employing GPUs to accelerate a variety of graph processing workloads, e.g., graph traversal [9], [10], [11], pagerank [12] and network motif detection [13]. These existing works utilize GPU's advantage of massive parallelism for in-memory graph processing. They leverage the *associative* property of their targeted graph applications to efficiently distribute workloads. For instance, one can assign a thread to visit each individual neighbor of a vertex in graph traversal applications [9], or assign a thread to independently extend a candidate motif by adding one neighborhood vertex in network motif detection [13]. In contrast, for each iteration of the LP algorithms, every vertex u scans all its neighbors for their label values and selects the most frequent label (MFL) to update the label value of u itself. Note that computing MFLs is *not* an associative workload for each neighbor list, which renders most existing studies *infeasible* for accelerating LP efficiently.

Although there have been some pioneer studies on parallelizing LP on GPUs [14], [15], we identify three major drawbacks when developing GPU-based LP system:

- The existing works only optimize the classical LP algorithm [7] on GPUs. However, many variants of LP are used by data engineers in e-commerce platforms to

- *C.Ye and Y.Li are are with the School of Computing and Information Systems, Singapore Management University. Email: {changye.2020,yuchenli}@smu.edu.sg*
- *B.He is with the School of Computing, National University of Singapore. Email:hebs@comp.nus.edu.sg*
- *Z.Li is with Alibaba Group, Beijing, China, 100026. Email:lzjoey@gmail.com*
- *J.Sun is with the School of Computing, Zhejiang University. Email:sunjl@zju.edu.cn*

develop strategies on detecting evolving fraud patterns. A survey compares 13,834 LP variants and the results show that the variants have their niche, i.e., none is universally superior [6]. Implementing a correct and efficient GPU program is challenging in general, and even more difficult for graph applications like LP. Hence, it is not practical to train every data engineers with the knowledge of tedious programming and performance optimizations on GPUs.

- The existing works simply leverage the raw computing power and high memory bandwidth of GPUs to accelerate MFL computation. As the workload is not associative, the existing methods rely on *parallel segmented sort* of the entire neighbor lists to order the labels for label frequency evaluation, and then select the MFL [14], [15]. It requires repeated scans of the labels, which bottlenecks the MFL computation. The performance gets worse when a neighbor list does not fit into the fast shared memory, in which the existing methods must resort to the slow GPU global memory for processing the neighbor list. On the other hand, for vertices with fewer than 32 neighbors, it also wastes resources to employ a warp of 32 threads for processing vertices with tiny neighbor sets.
- The existing works incur expensive data transfer cost between CPUs and GPUs when the graph size exceeds the GPU memory capacity. Graphs that cannot fit into the GPU memory are divided into partitions for out-of-core processing on GPUs. For every LP iteration, all graph partitions must be transferred to the GPU memory. According to our experiments, data transfer costs take over 60% of overall execution time for out-of-core LP processing.

In this paper, we propose a GPU-based framework to support large scale LP processing, called GLP. To ease the development of different LP variants, we offer a set of user-defined APIs in the GLP framework. These APIs provide expressive and bulk-synchronize abstractions for data engineers to quickly deploy LP variants tailored for their targeted applications, e.g., develop various algorithms to enhance the system capability of detecting new frauds in e-commerce platforms, without domain knowledge of GPUs. Furthermore, the design of GLP can seamlessly support massive graphs that do not fit into the GPU memory entirely, and GLP handles such scenarios with a CPU-GPU hybrid execution mode.

Built upon the GLP framework, we propose three novel optimizations. Our optimizations take advantages of the characteristics of LP algorithms, which are overlooked by the existing works.

First, we maximize shared memory usage even if a neighbor set does *not* fit into the shared memory. The existing works have to access the slow global memory to compute the MFL for high-degree vertices as their neighbors cannot fit into the shared memory. We leverage an important observation: two neighbors of a vertex often share the same label as they have a high chance to be in the same community [16]. To maximize shared memory processing for computing MFL, we combine two shared memory resident data structures: count-min sketch (CMS) and hash table (HT). CMS is used to effectively prune labels with low frequencies and HT is deployed to store potential MFL without accessing the global memory. We theoretically show that the global memory access only occurs with small probabilities.

Second, we optimize the MFL computation for vertices with low-degree. For most large graphs, the number of low-degree vertices are massive due to the power-law principle. The existing works either use a warp of 32 threads or a single thread to handle one low-degree vertex but both approaches suffer from low GPU resource utilization. As GPUs are very sensitive to imbalanced workloads caused by low-degree vertices, we optimize the MFL computation by employing GPU warp-centric intrinsics for grouping threads and updating the labels of multiple vertices concurrently in a warp.

Third, we propose a novel scheme to enable LP processing on compressed graphs. The reduction in memory usage achieved by compressing the neighbor lists significantly lowers the data transfer expense for out-of-core processing. Utilizing a simple approach that fully decodes the compressed representation for LP processing leads to frequent GPU memory accesses. To address this issue, we opt for a *partial* decoding of the compressed neighbor lists, generating a more compact representation. Subsequently, we develop a dynamic scheduling strategy that effectively allocates a neighboring vertex from the compact representation to a thread while maintaining workload balance.

We thereby summarize our contributions as follows:

- We propose GLP, a GPU-based framework that allows data engineers to deploy user-defined, efficient and scalable LP algorithms on GPUs for different application requirements of the data science pipeline. To our best knowledge, this is the first GPU framework that can support a range of LP algorithms.
- We devise a novel structure that combines CMS and HT to maximize shared-memory usage for MFL computation. We introduce a warp scheduling approach that handles multiple low-degree vertices concurrently for efficient in-memory processing on GPUs.
- We optimize data transfer by processing LP on compressed graphs. To avoid full decoding, we develop a compact representation where only partial decoding is sufficient for LP processing.
- Extensive experiments confirm: (1) GLP achieves significant speedups over the existing CPU and GPU approaches for LP. (2) GLP reduces the data transfer cost for LP processing on out-of-core graphs with the compression techniques on average of 88.4%. (3) GLP processes real-world fraud detection workloads on a graph of over 10 billion edges with a single GPU, while offers 13.2x speedup to the current in-house solution running on a high-end multicore machine. We show that GLP can be an efficient and cost-effective solution towards fraud detection pipelines.

This journal article expands on our earlier conference paper [17]. In this extended version, we introduce new compressed label propagation methods in Section 5, specifically designed for optimizing out-of-core GPU graph processing. Further, Section 6.4 presents new experiments that highlight an average data transfer cost reduction of 88.4% achieved using our compression techniques. Finally, the case study in Section 6.5 demonstrates that our methods amplify the

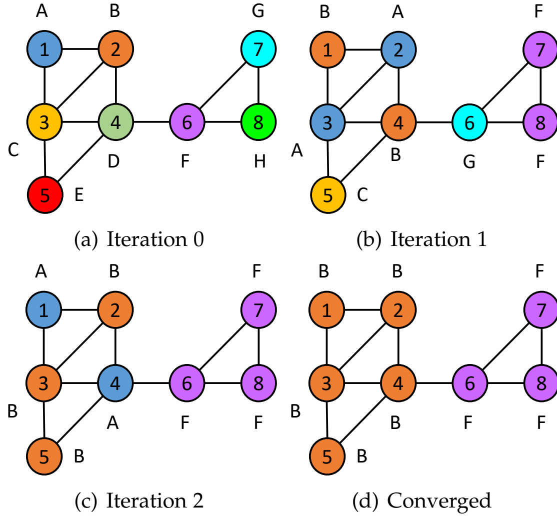

Fig. 2. An example of parallel LP execution.

speedup from 8.2x to 13.2x over TAOBAO's in-house solution when handling a real billion-scale graph workload.

In what follows, we present the preliminaries and the literature review in Section 2. The GLP framework is introduced in Section 3. The in-GPU optimizations and compressed LP processing are presented in Section 4 and Section 5, respectively. Section 6 discusses the experimental results. We conclude the paper in Section 7.

2 PRELIMINARIES

2.1 Label Propagation (LP)

Given a graph $G = (V, E)$, where V represents the vertex set and E represents the edge set. The neighbor set of a vertex v is denoted as $N(v)$. We present the classical LP [7] as a brief introduction to LP algorithms. Every vertex v_i is first initialized with a unique label L_i representing its cluster ID. Subsequently, each vertex takes the most frequent label (MFL) among its neighbors for updating its own label. Specifically, it invokes two stages for updating the labels: (1) retrieving the labels of neighbors for all vertices; (2) count the labels and extract the MFL. This process iterates until a termination condition is met. Vertices with the same label are assigned to the same cluster. The example in Figure 2 shows the first three iterations of LP. Two clusters are identified by labels B and F after LP converges at step (d).

We note that many variants of LP are proposed [18], [19], [20], [21], [22], see a survey in [6]. Nevertheless, they follow the same pattern which a vertex first loads the label value of neighbors and then use the MFL to aggregate the labels of neighbors. We hence design the GLP framework anchored on this pattern for supporting different LP variants. More details will be presented in Section 3. Before moving on, we present the frequently used notations in Table 1.

2.2 Related Studies

We review the literature by discussing four related areas: (1) LP processing on GPUs; (2) Optimizing workload balance for GPU graph processing; (3) Optimizing memory access for GPU graph processing; and (4) Out-of-core graph processing on GPUs.

TABLE 1
Frequently used notations in this paper

Symbol	Descriptions
$G(V, E)$	a graph G with vertex set V and edge set E
(v, u)	an edge from vertex v to vertex u
l	a label of a vertex
L,L_i	the label value array (indexed by the vertex id i)
MFL	the most frequent label (among the neighbors of a vertex)
CMS, HT	the count-min sketch and the hash table in the shared memory
GHT	the hash table in the global memory
d	the number of hash functions for CMS
w	the number of buckets for each hash function for CMS
h	the number of buckets for HT

GPU-based label propagation. The existing works leverage the high memory bandwidth of GPUs to load the labels of neighbors for each vertex [14], [15]. Subsequently, a GPU-optimized segmented-sort kernel is executed to order the neighbor labels. Lastly, a count kernel scans the ordered neighbor labels to extract MFL and then updates the vertices in parallel. There are two major performance issues when the aforementioned approach is executed iteratively for LP: (1) the label values are repeatedly loaded but only a subset of them have their labels updated, which leads to unnecessary non-coalesced global memory accesses; (2) the segmented-sort kernel executed on the entire graph is an overkill for obtaining MFL and thus incurs redundant workloads. GLP addresses these issues with a novel hash table-based design.

Optimizing workload balance for GPU graph processing. Accelerating graph workloads on GPUs incurs unbalanced work distribution as the neighbor lists in large graphs follow the power-law distribution. The most common workload that is being extensively studied is graph traversal. Hong et al.[23] propose a virtual warp-centric BFS algorithm that divides an entire warp into small virtual groups to active more threads within a warp. This technique is later adopted by Medusa [24]. Merrill et al. [25] devise a block-based thread scheduling where a block/kernel is assigned to handle a high degree vertex, which is the same strategy adopted by Gunrock [26]. Liu et al. [10] introduce iBFS for processing concurrent BFS where a warp voting technique is used to balance the workloads. Many scheduling approaches have been proposed for other graph applications, see [27] for a comprehensive survey. However, there is an important assumption made in these existing works: the workloads are associative in every neighbor lists, which does not hold for the LP algorithm. Our proposed solution introduces a number of novel optimizations to handle the non-associativity: (1) combine HT and CMS for handling large degree vertices that do not fit into the shared memory and reduce workloads; (2) group small vertices for computing their label frequencies in a warp. (3) compute the label frequencies on the compressed neighbor list.

Optimizing memory access on GPU graph processing. Recent works on GPU-based graph processing propose solutions for addressing irregular memory accesses. Cusha [28] eliminates random memory access by reordering edges into G-shards, but at the cost of producing block/warp divergence. Furthermore, the window representation proposed in

Cusha is designed for associative workloads [29]. However, the task of extracting MFL in LP is naturally non-associative, which renders the infeasibility of adopting Cusha for LP. Khorasani et al. [30] introduce warp segmentation (WS) for graph processing. Instead of reducing non-coalesced memory accesses directly, they hide the latency of memory transfer by feeding GPUs with compute-intensive tasks. Nevertheless, this strategy is not suitable for I/O intensive algorithms such as LP.

Out-of-core Graph Processing on GPUs. In general, there are two strategies developed in existing systems to enable efficient out-of-core graph processing using GPUs. The first strategy replies on the Unified Memory Access (UMA) to manage large graphs [31], [32]. The second strategy decomposes large-scale graphs into smaller subgraphs that can fit into the GPU's memory [33], [34], [35], [36]. For LP workloads, the entire graph is accessed for every processing iteration. Hence, the existing strategies cannot reduce the amount of data being transferred.

We adopt an orthogonal strategy that enables LP processing on compressed graphs. Sha et al. [11] propose a GPU graph traverse approach on compressed graphs.Each neighbor list in the graph is encoded to a compact graph representation (CGR) via Variable Length Code (VLC) for compression. However, directly employing the processing strategy in [11] results in frequent global memory accesses due to complete VLC decoding. We propose a partial decoding scheme that decodes CGR to a compact representation of intervals and residuals, on which we can execute LP efficiently. In addition to CGR, Kaczmarski et al. [37] introduce a fixed-length compression technique, which is employed to efficiently represent integers in a compact and tightly constrained manner. However, it is unsuitable for scaling to large graphs containing millions of nodes since the efficacy of fixed-length encoding is limited by the range of values it can represent. Such an issue has been pointed out in [11]. Yin et al. [38] propose Chunk-wise Graph Compression format for graph sampling. This format, which uses linear estimation (LE) to approximate the entire neighbor list, is efficient for sampling purposes which visits neighbors in the adjacency list with random access. However, for LP processing, we need a full scan of the adjacency list. Furthermore, LE requires additional stores and loads for bias of each vertex before extracting the vertex ID, leading to a full compression of the adjacency list for LP processing.

3 The GLP Framework

We follow two important design goals to make our system a useful framework.

- *Programmability*. We provide a set of user-defined APIs for data engineers to develop various LP variants on GPUs with ease.
- *Efficiency*. The designed processing workflow and APIs allow efficient and scalable GPU implementations.

Figure 3 shows the data structures and processing workflow of GLP. We use two structures, *Graph* and *Attributes*, to represent the underlying graph and its attributes/labels. The compressed sparse row (CSR) format is used to store the graph structure. The users of GLP can include additional user-defined data structures for customization, but are advised to follow the structure of arrays (SoA) layout for coalesced memory accesses on GPUs. The workflow of GLP is iterative and each iteration contains three main components as follows:

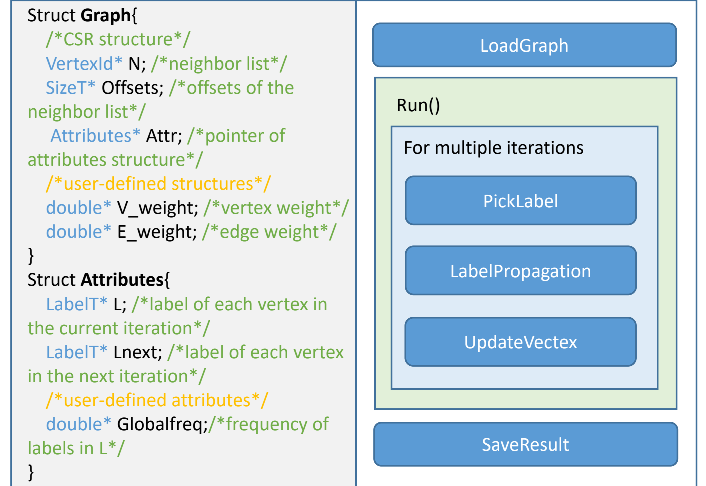

Fig. 3. Overview of GLP.

- PickLabel. This component is responsible for deciding a label for each vertex with a user-defined strategy.
- LabelPropagation. Each vertex will pick the label which achieves the highest score value among its neighboring vertices. The users can customize the score functions of labels. We illustrate more details in Section 4.
- UpdateVertex. Given a vertex and a label picked from LabelPropagation, it is responsible for updating the status of the vertex with a user-defined strategy.

APIs. We provide APIs for developers to customize and deploy their LP algorithms on GPUs for different application requirement on the data science pipeline. We show some sample APIs in Table 2. With the help of those APIs, users can directly customize PickLabel and UpdateLabel. For LabelPropagation, we expose two APIs, namely LoadNeighbor and LabelScore, to balance between ease of customization and efficient GPU optimizations. The LabelPropagation kernel running on GPUs will invoke LoadNeighbor and LabelScore to select the MFL for each vertex. The configurations for GPU kernel functions are automatically set up, there is no requirement for users to deal with any GPU optimizations.

Examples. To showcase the ease of implementing various LP algorithms for data engineers to deploy different LP variants, we present three commonly used LP variants under the GLP framework in Figure 4.

- Classic LP. This algorithm is introduced in Section 2.1
- LLP (The layered LP algorithm [19]). The classic LP tends to provide undesirably large communities. In contrast, LLP updates its label by the following formula. For each label l currently appearing on the neighbors of a vertex, LLP computes $val = k - \gamma * (v - k)$. k is the number of neighbors having the same label with l, γ is a density parameter, and v is the overall number of vertices having the same label with l. The classic LP chooses l maximizing k, whereas LLP chooses the label maximizing val.

TABLE 2
Sample User-defined APIs in GLP

APIs/Parameters	Descriptions
PickLabel(VertexId *vid*)	Given a vertex vid, it decides vid's label and write the label to the current label array L.
LoadNeighbor(VertexId *vid*, VertexId *did*)	Given an edge (vid, did), it returns the label and the frequency for did as a neighbor of vid.
LabelScore(VertexId *vid*, LabelT *l*, double *freq*)	Given a vertex vid, a label l and l's frequency *freq* among vid's neighbors, it returns a score of l for vid.
UpdateVertex(VertexId *vid*, LabelT *l*, double *score*)	Given a vertex vid, update the status of vertex vid with label l and $score$.

```
__device__ Graph* G; /*global graph structure G*/
/*---------------------------------------- ----Classic LP---------------------------------------------*/
__device__ void PickLabel(VertexId vid){
  /*copy Lnext to L */
  G-> Attr-> L[vid] = G-> Attr-> Lnext[vid];
}
__device__ pair<LabelT, double>
LoadNeighbor (VertexId vid, VertexId sid){
  LabelT l = G-> Attr-> L[sid];
  /*accumulates frequency of l by one */
  return pair<double,LabelT>(l, 1.0);
}
__device__ double LabelScore
(VertexId vid, LabelT l, double freq){
  /*return freq as its label score*/
  return freq;
}
__device__ void UpdateVertex
(VertexId vid, LabelT l, double freq){
  return;
}
/*---------------------------------------- ---------LLP---------------------------------------------------*/
__device__ void PickLabel (VertexId vid){
  LabelT l = G-> Attr-> Lnext[vid];
  G-> attr-> L[vid]] = l;
  /*accumulate the frequency of l*/
  UpdateGlobalFreq(l);
}
__device__ pair<LabelT, double>
LoadNeighbor (VertexId vid, VertexId sid){
  LabelT l = G-> Attr-> L[sid];
  return pair<LabelT, double>(l, 1.0);
}
__device__ double LabelScore
(VertexId vid, LabelT l, double freq){
  double r = G-> Attr-> Gamma;
  /*get the frequency  of label l in L*/
  double v = GetGlobalFreq(l);
  return freq – r * ( v - freq );
}
__device__ void UpdateVertex
(VertexId vid, LabelT l, double freq){
  return;
}
/*---------------------------------------- ---------SLP---------------------------------------------------*/
__device__ void PickLabel(VertexId vid){
  G-> Attr-> L[vid] = randomPick(vid);
}
__device__ pair<LabelT, double>
LoadNeighbor (VertexId vid, VertexId did){
  EdgeT eid = GetEdgeId(vid,did);
  double w = V_weight[vid] +
E_weight[did];
  LabelT l = G-> Attr-> L[vid];
  return pair<LabelT, double>(l, w);
}
__device__ double LabelScore
(VertexId vid, LabelT l, double freq){
  return freq;
}
__device__ void UpdateVertex
(VertexId vid, LabelT l, double freq){
  LabelT l = G-> Attr-> Lnext[vid];
  /*update labels of vid in candidates*/
  AddLabelToLists(vid, l);
  RemoveLowFreqLabels(vid);
}
```

Fig. 4. User-defined APIs for LP variants.

- SLP (The speaker-listener LP algorithm [39]). Both classical LP and LLP can only assign a vertex to one community. SLP is designed for identifying overlapping communities. Each vertex may have multiple labels. In each iteration one label among the candidates is chosen to be the current label of a vertex. Each vertex then selects the MFL from its neighbors like the classical LP, and the MFL is used to update the candidate labels for each vertex. At the end of each iteration, labels whose frequency are less than a threshold will be removed from the candidates.

4 Optimize MFL Computation

In this section, we present optimizations for implementing LabelPropogation on computing MFL under the GLP framework. We focus on two types of vertices, high and low degree vertices, which are the major issues for memory access overhead and workload imbalance respectively.

4.1 Handling High Degree Vertices

The existing works either use segmented sort or global hash tables for counting the frequencies of labels. However, the existing approaches have trouble in handling high degree vertices. Implementations based on segmented sort have to gather labels into an addition array in the GPU global memory, i.e., the neighbor label array NL, and impose expensive memory overheads on GPUs as the size of NL is proportional to the total number of edges. Additionally, segmented sort degenerates to plain parallel sort for high degree vertices. In this approach, multiple scans on NL are required. Thus, the segmented sort approach incurs unnecessary workloads for obtaining the MFL. The other possible approach is to allocate a hash table for each vertex v with memory size equivalent to v's neighbors in the GPU global memory for counting the label frequencies. The hash table approach relies on the built-in caching mechanism of GPUs to reduce global memory accesses. However, when the number of neighbors exceeds the cache size, the hash table cannot avoid random accesses in the global memory.

In this work, we propose a shared memory approach that handles high degree vertices even when the neighbors of a vertex exceed the shared-memory size. This is possible due to the important observation that, as more iterations are executed, neighbors of a vertex often share similar labels since they are likely to be assigned in the same cluster. Hence, the number of *distinct* labels among a vertex v's neighbors could be drastically smaller than the degree of v. The observation enables opportunities for handling the frequency calculation in the shared-memory alone. Nevertheless, the number of unique labels cannot be determined before accessing all neighbors of a vertex. To avoid unnecessary global memory accesses, we combine a Count-Min Sketch (CMS) and a Hash Table (HT) in the shared memory for estimating the label frequencies of a high degree vertex. CMS [40] is an effective approach for estimating frequencies in the data stream scenario. For each arriving label l, CMS hashes l to d independent hash functions and increment the counts in the corresponding buckets. CMS only overestimates the frequency of a label and has a probabilistic guarantee on the upper bound of the frequency value.

Our approach takes only one scan of the neighbor labels for any vertex v. One thread block is assigned to v and each thread is assigned to handle one neighbor u for v

Procedure SharedMemBigNodes

input : threadid tid, vertex v, shared memory structures HT,CMS, global memory hash table GHT, neighbor array N, $offset[v]$ indicates the starting index of neighbor list for v in N.

output: updated label array $Lnext[]$

1. $u := N\,[offsets\,[v] + tid]$
2. $(l, weight) :=$ **LoadNeighbor** (v, u)
3. $ht_score := INT_MIN$
4. $cm_score := INT_MIN$
5. $freq := atomicAdd\,(\mathtt{HT}, l, weight)$
6. **if** *unsuccessful insertion* **then**
7. $\quad freq := atomicAdd\,(\mathtt{CMS}, l, weight)$
8. $\quad cm_score :=$ **LabelScore**$(v, l, freq)$
9. **else**
10. $\quad ht_score :=$ **LabelScore**$(v, l, freq)$
11. $s(\mathtt{HT}) := blockReduce\,(ht_score, max\,())$
12. $s(\mathtt{CMS}) := blockReduce\,(cm_score, max\,())$
13. **if** $s(\mathtt{HT}) \geq s(\mathtt{CMS})$ **then**
14. $\quad$ **if** $s(HT) == ht_score$ **then**
15. $\quad\quad Lnext\,[v] := l$
16. **else**
17. $\quad$ **if** $l \notin \mathtt{HT}$ **then**
18. $\quad\quad freq := globalInsert(\mathtt{GHT}, l, weight)$
19. $\quad\quad gt_score :=$ **LabelScore**$(v, l, freq)$
20. $\quad$ **else**
21. $\quad\quad gt_score := ht_score$
22. $\quad s(\mathtt{GHT}) := blockReduce\,(gt_score, max\,())$
23. $\quad$ **if** $s(\mathtt{GHT}) == gt_score$ **then**
24. $\quad\quad Lnext\,[v] := l$

and its label l. A CMS and a HT are allocated in the shared memory. When scanning, we insert l into HT and increment the label frequency $\mathtt{HT}(l)$ in the hash table if the insertion is successful. Otherwise the hash table is full, we add l to CMS, followed by computing a score based on the label and its frequency. After processing all neighbors, the thread block synchronizes to find the maximum score in HT as $s(\mathtt{HT})$ and the maximum score estimated by CMS as $s(\mathtt{CMS})$. If $s(\mathtt{HT}) \geq s(\mathtt{CMS})$, we can safely update the vertex by using the MFL in HT. The approach maximizes the chances for shared memory processing.

We present the aforementioned approach in Procedure SharedMemBigNodes. One thread with ID tid loads the label of one neighbor u of v to l as well as l's weight (Line 2). For the ease of presentation, a thread only processes one neighbor of v but one thread will process multiple labels in the actual implementation.

We try to insert the label to the shared memory structures HT and CMS with the *atomicAdd* primitives (Lines 5-7). The variables ht_score and cm_score store the scores of label l in HT and CMS after the insertion, respectively. Upon insertions are complete, two *blockReduce* primitives [1] are invoked to get the maximum scores in HT and CMS from all threads in the block. We can safely update $Lnext[v]$ with the MFL in HT if $s(\mathtt{HT}) \geq s(\mathtt{CMS})$ (Lines 13-15). Otherwise, we insert l into the

1. *blockReduce* is a GPU block-wise reduction that uses a binary max operator to compute a single aggregate from a list of input elements.

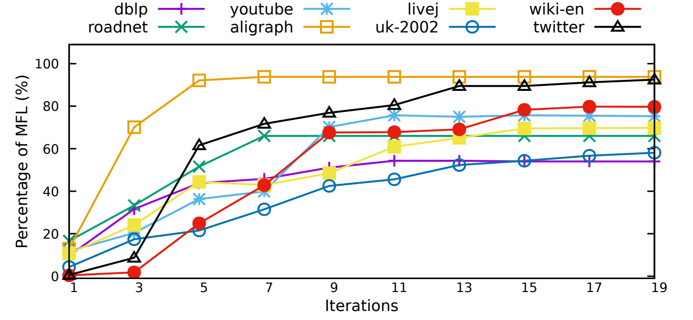

Fig. 5. The average percentage of MFL in the high-degree (larger than 128) vertices across all datasets.

global hash table GHT and retrieve the MFL from both GHT and HT to update $Lnext[v]$ (Lines 16-24).

Special Note. The proposed approach for combining CMS and HT is *not* an approximated solution for MFL computation. Instead, it is a pruning strategy that takes advantages of the label distribution in the neighborhood. In the worst case, we still need to access the global memory to count label frequencies. Then we show that this strategy can effectively reduce the global memory accesses with a high probability.

Theoretical Analysis. We discuss the theoretical guarantee for our proposed shared memory approach. In particular, we study the probability that global memory accesses are needed for processing any given vertex v in a classic LP algorithm. To simplify the analysis, we assume that all labels except the MFL appear only once in the neighbor list. This assumption is grounded in an observation of the label distribution during label propagation for high-degree nodes, where the MFL predominantly prevails among the majority of neighbors. We measure the average percentage of average MFL changes across high-degree vertices at each iteration of label propagation, as depicted in Figure 5. It reveals that the percentage of MFL experiences a swift escalation in the initial iterations, followed by a more gradual increase in subsequent iterations.

Let m be the number of distinct labels in $N(v)$, h be the number of buckets in the HT, d and w be the number of independent hash functions and the number of buckets for each hash function in the CMS, respectively. For any label l, $f(l)$ denotes the frequency count in $N(v)$ and $f_{\mathtt{min}} \leq f\,(l) \leq f_{\mathtt{max}}$. We first show the following lemma to study the probability that the label with the maximum frequency in $N(v)$ is *not* in the HT after inserting all labels into the HT and the CMS in a random order. To simplify the analysis, we assume that all labels except the MFL appear only once in the neighbor list.

Lemma 1. Let l^* be the label where $f(l^*) = f_{\mathtt{max}}$. Then $\mathbb{P}[l^* \notin \mathtt{HT}] \leq (1 - \frac{h}{m+k})^{2k}$ where $k = \frac{f_{\mathtt{max}}-1}{2}$.

Proof 1. We only discuss the case where $m > h$ since all unique labels will be presented in HT if $m \leq h$. We study the following random process: all distinct labels except l^* are randomly permuted as a sequence s and we insert l^* into s for $f_{\mathtt{max}}$ times at random positions. Hence, $l^* \notin \mathtt{HT}$ if and only if l^* does not appear in the first h positions

of s. It then follows that:

$$\mathbb{P}[l^* \notin \texttt{HT}] = \frac{m-h}{m} \cdot \frac{m+1-h}{m+1} \cdot \ldots \cdot \frac{m+f_{\texttt{max}}-1-h}{m+f_{\texttt{max}}-1} \leq \Big(\frac{m+(f_{\texttt{max}}-1)/2-h}{m+(f_{\texttt{max}}-1)/2}\Big)^{(f_{\texttt{max}}-1)}$$

The ith factor of the equation presents the probability that the MFL is captured by the HT in the ith position. Further, the inequality holds since $\frac{m+i-h}{m+i} \cdot \frac{m+n-i-h}{m+n-i} \leq (\frac{m+n/2-h}{m+n/2})^2 \ \forall i \in [0,n]$. Simply substitute $k = \frac{f_{\texttt{max}}-1}{2}$ derives the lemma.

From Lemma 1, we can infer that l^* has a low probability not present in the HT when dealing with practical scenarios. Suppose $m \leq k$ (i.e., the number of unique values is small compared with the maximum frequency count in $N(v)$), $\mathbb{P}[l^* \notin \texttt{HT}] \to e^{-h}$ for large $f_{\texttt{max}}$, where the probability decreases exponentially with h.

Next, we analyze the scenario where the maximum frequency estimated by CMS is *larger* than the maximum frequency in HT when $l^* \in \texttt{HT}$. Such a scenario implies global random accesses are needed. To establish the theoretical result, we denote $f(\texttt{HT})$ as the sum of frequency counts of labels inserted into the HT and $g(l)$ to denote the frequency count estimated by the CMS for label l.

Lemma 2. $\mathbb{P}[\max_l g(l) > f_{\texttt{max}}] \leq m\delta$ where $\delta = 2^{-d}$.

Proof 2. As the hash table stores $f(\texttt{HT})$ counts, the number of insertions to the CMS is $s = (|N(v)| - f(\texttt{HT}))$. As all frequency counts are integers, we have the followings:

$$\begin{aligned} \mathbb{P}[g(l) > f_{\texttt{max}}] &= \mathbb{P}[g(l) \geq f(l^*)+1] \quad (1)\\ &\leq \mathbb{P}[g(l) \geq f(l)+1] \\ &\leq \mathbb{P}[g(l) \geq f(l) + \frac{1}{s}\cdot s] = 2^{-d} \quad (2) \end{aligned}$$

Equation 2 holds due to the properties of the CMS [40] when w is set to $2s$. Hence, $\mathbb{P}[\max_L g(l) \geq f_{\texttt{max}}] \leq m\delta$ by the union bound of m distinct labels.

Now we are ready to estimate the probability of having global memory accesses.

Theorem 1. The probability of global memory accesses is bounded by $(m\delta + e^{-h})$ for any vertex v as $f_{\texttt{max}} \to \infty$ and $m \leq \frac{f_{\texttt{max}}-1}{2}$.

The proof naturally follows by combing Lemma 1 and Lemma 2. In practice, the maximum frequency $f_{\texttt{max}}$ becomes large and the number of distinct label m becomes small after a few iterations of LP for high degree vertices since communities form when labels are merged. Hence, $(m\delta + e^{-h})$ is small and it renders our proposed approach effective in reducing global memory accesses.

4.2 Handling Low Degree Vertices

The existing works either assign a single thread or one warp of 32 threads to handle one low-degree vertex. Both strategies are far from optimal. One-thread-one-vertex strategy has the workload imbalance issue when two threads are assigned to two vertices with different number of neighbors. Further, the threads in a warp access different neighbor lists concurrently, and have frequent uncoalesced memory accesses for the warp. One-warp-one-vertex strategy handles a vertex with 32 threads and all threads in the warp access the same neighbor list to avoid imbalance workload and uncoalesced memory accesses. However, vertices with less than 16 neighbors are common in power-law graphs [41]. One-warp-one-vertex strategy will result in many idle threads and under-utilize the computing resources of GPUs.

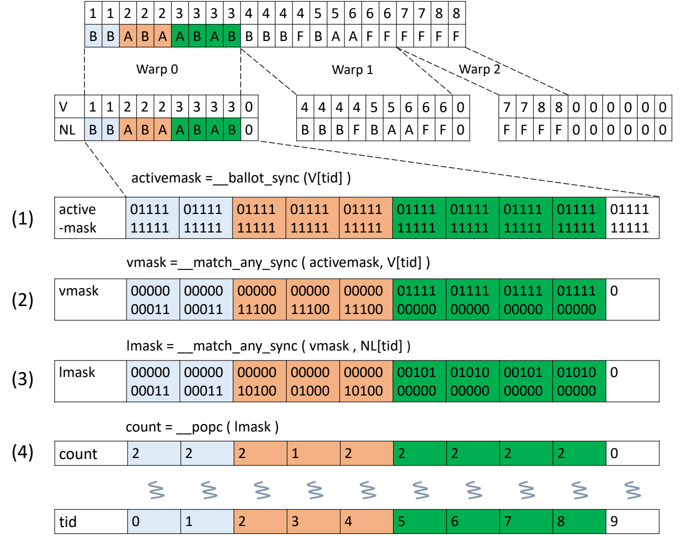

Fig. 6. An example of the warp-centric approach. Each number in V represents a node ID v and the number in NL below v represents a label from a neighbor of v. Different colors represent different vertices handled by the threads in warp 0 respectively. The bit mask for thread tid represents all threads in warp 0 having the same workload as tid.

Motivated by the above drawbacks, we propose to employ a warp of 32 threads to compute the label frequencies for multiple low-degree vertices concurrently, i.e., one-warp-multi-vertices approach. In this way, we make full utilization of GPU threads compared with the one-warp-one-thread approach. As the threads in a warp execute the instructions in a lock-step manner, the major challenge is how to efficiently identify the set of peer threads in the warp working on (1) the same neighbor list; (2) the same label from the same neighbor list. To solve the challenge, we combine a number of warp-centric intrinsics to quickly identify peer threads.

In the following, we give an illustrative example for the classic LP algorithm and demonstrate how to handle multiple low-degree vertices concurrently in a warp. For ease of presentation, we assume the size of a warp is 10 and warp 0 handles vertices 1, 2 and 3. The execution sketch is illustrated in Figure 6.

1) We select those active threads having a valid label in the warp. by calling the **__ballot_sync** intrinsic [2]. The returned $activemask$ tracks which threads are assigned with a valid label to work on. For warp 0, all threads have the same $activemask$ and the most significant digit is 0 indicating that thread 9 will be idle.
2) We group the active threads according to their assigned vertex by calling the **__match_any_sync** intrinsic [3]. The returned $vmask$ for each thread t indicates the set of

2. **__ballot_sync** returns the bit mask of all active threads where the input parameter is non-zero.

3. **__match_any_sync** returns the bit mask of all active threads (indicated by the activemask) that have the same value of the input parameter.

peer threads assigned with the same vertex as t. In our example, both thread 0 and thread 1 are assigned with vertex 1, thus the position 0 and 1 of $vmask$ are set to 1

3) We employ **__match_any_sync** again for grouping threads to compute the label frequency. The returned $lmask$ for each thread tid indicates the set of peer threads assigned with the same label of the same vertex. In our example, thread 2 holds label A from vertex 2. Among all three threads that are assigned to vertex 2, only thread 4 is assigned with label A, thus both threads 2 and 4 hold the same $lmask$ where the corresponding positions are set to 1.
4) The count of label frequency is the number of ones in $lmask$, which are computed by simply calling the **__popc** intrinsic [4].

It is noted that our approach handles multiple vertices for updating the labels with warp-level intrinsics. As warp-level intrinsics are extensively optimized in the throughput-oriented architectures, the approach can efficiently process low degree vertices. The atomic operations are replaced by efficient bit manipulation with the help of intrinsics. Each intrinsic operation can be executed much more efficiently within a few clock cycles on GPUs.

5 GLP ON COMPRESSED GRAPHS

For large graphs that do not fit into the GPU memory, the existing works on out-of-core GPU graph processing divide the graphs into smaller partitions [33], [34], [35], [36]. The partitions are transferred to GPU via the PCIe bus based on the computational demand. However, since LP requires to process all adjacent lists, all partitions must be unavoidably transferred to GPUs in each LP iteration. In effect, the partition-based approach does not reduce the amount of transferred data. Although the existing works take advantage of asynchronous transfer to overlap the PCIe overhead with in-GPU processing [34], [36], they assume that the PCIe bus is dedicated to a single application to maximize its usage. In modern data centers, the PCIe bus is shared by multiple applications on a diverse set of peripheral devices, such as network adapters [42], SSDs [43] and FPGAs [44]. Reducing the amount of transferred data in a shared PCIe environment is essential for achieving high-performance data processing in practice.

In this paper, we develop techniques for enabling LP processing on compressed graphs. The compression results in reduced data to be transferred via PCIe. In what follows, we first introduce the compressed graph representation (CGR) and discuss the technical challenges for enabling LP processing on CGR. Subsequently, we propose a novel partial decoding scheme that processes LP on a compact representation without full CGR decoding.

5.1 Compressed Graph Format

Recently, Sha et al. [11] propose a novel CGR such that larger graphs can fit into the GPUs for graph traversal. The CGR compresses each vertex's neighbor list via Variable Length Encoding (VLC). As depicted in Figure 7, the

4. **__popc** counts the number of bits that are set to 1 of the input parameter.

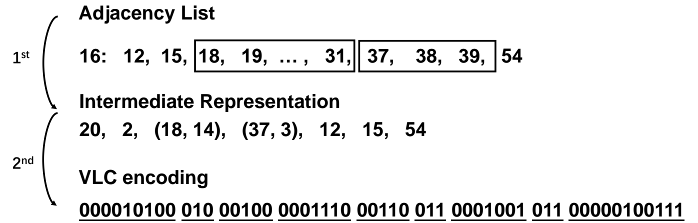

Fig. 7. The CGR of an adjacent list. The adjacency list contains 20 neighbors of vertex 16 (the first neighbor is vertex 12). After transformed to the compact representation, the list is represented as a combination of intervals and residuals. The number of neighbors is stored (i.e., 20) followed by the number of intervals (i.e., 2). Vertex 18 in the first interval (18,4) is encoded as 2(00100), which is the gap between vertex 18 and vertex 16. We encode a positive gap x as $2x$, otherwise, a negative gap as $2x+1$ for the first interval and residual. Thus, the gaps 2 is encoded as 00100. The length of the intervals 14 is encoding as 0001110. The starting vertex of second interval (37,3) is encoded as 6(00110), which is the gap between the starting vertex of the second interval 37 and the ending vertex of the first interval 31. The length of second interval is encoded as 3(011). The first residual 12 is encoded as the gap between the vertex 12 and vertex 16, which is -4(0001001). The second and final residuals, 15 and 54, are encoded based on the gap between their respective values and the value of the preceding residual, which are 3(011) and 39(00000100111).

compression scheme first transforms a neighbor list to a compact representation of intervals and residuals. The interval contains nodes with continuous vertex IDs and the remaining vertices are left as residuals. For every sequence of intervals and residuals, we convert it into a differential sequence based on the original sequence. This involves representing each element as the difference between itself and the element that precedes it. Afterwards, VLC encoding is used to compress the intervals and residuals to a bit array.

With the CGR, the neighbor lists of large graphs are compressed into bit arrays. However, it is not feasible to execute LP on the bit array directly. A natural approach is to adapt the decoding method proposed in [11] and recover the neighbor lists completely. The recovered neighbors are then processed with the LP logic as discussed in Section 4. Nonetheless, this approach faces two major challenges. First, storing the decoded neighbors is memory-intensive, which can lead to expensive memory accesses on GPUs. Given that LP workloads are memory-bound, it is critical to minimize memory accesses. Second, power-law graphs often contain massive low-degree vertices, which require a sequential decoding process. Decoding all these vertices in parallel requires substantial thread resources and incur expensive synchronization overhead. In what follows, we discuss techniques that enable efficient LP processing without completely decoding the neighbor lists.

5.2 Label Propagation on CGR

Partial Decoding. To avoid decoding the adjacency lists completely, we partially decode the bit strings to the compact representation of intervals and residuals and process LP directly on the compact representation. To clarify, our process does not involve decoding a segment of the bit string. Instead, we decode the entire bit string to reconstruct the intermediate representation in Figure 7.

However, the intervals and residuals make it challenging to evenly distribute the LP processing workload among threads. We propose a dynamic scheduling strategy which

Procedure compressedLP

input : lane ID lid, vertex v, CGR $bitStr$
output: updated label array $Lnext[]$

1 $degNum := decode(bitStr)$
2 $itvNum := decode(bitStr)$
3 $itvIdx := 0$
4 $curLen := 0$
5 **while** $degreeNum$-- **do**
6 $\quad curNode := decode(bitStr)$
7 $\quad curLen := (itvNum\text{--} > 0)\ ?\ decode(bitStr) : 1$
8 $\quad$ **while** $itvIdx + curLen \geq warpsize$ **do**
9 $\quad\quad$ **if** $lid \geq itvIdx$ **then**
10 $\quad\quad\quad u := lid - itvIdx + curNode$
11 $\quad\quad Lnext[] := LabelPropagation(v, u)$
12 $\quad\quad curNode := curNode + warpsize - itvIdx$
13 $\quad\quad curLen := curLen - warpsize + itvIdx$
14 $\quad\quad itvIdx := 0$
15 $\quad$ **if** $(itvIdx \leq lid < itvIdx + curLen)$ **then**
16 $\quad\quad u = lid - itvIdx + curNode$
17 $\quad itvIdx := itvIdx + curLen$
18 **if** $lid \leq itvIdx$ **then**
19 $\quad Lnext[] := LabelPropagation(v, u)$

efficiently assigns a neighbor vertex to a corresponding thread and balances the workload among threads. Once a thread is assigned to a neighbor vertex, it gets the label information of the neighbor and proceeds with the MFL computation discussed in Section 4.

Procedure compressedLP presents our dynamic scheduling strategy using the compact representation. Each warp is assigned to a CGR segment of vertex v. Every thread receives the bit string $bitStr$ of the segment and its lane ID lid. At first, the threads within the warp decode the number of neighbors and the number of intervals from $bitStr$ (Lines 1-2). Following this, the threads decode intervals and residuals from $bitStr$ one at a time. We employ two variables, $itvIdx$ and $curLen$, to maintain a sliding window for each interval. $itvIdx$ serves as the starting pointer of the sliding window, while $curLen$ represents the length of the sliding window. If $curLen$ exceeds the warp size, we assign each thread in the warp to a neighbor vertex u (Line 10). Once all threads are assigned, we execute label propagation for vertex v and its neighbor u (Line 11). The remaining assigned neighbors are processed similarly in subsequent iterations by adjusting $itvIdx$ and $curLen$ (Lines 12-14). If $curLen$ is smaller than the warp size, we only assign neighbors to those threads within the sliding window (Line 16). We continue to move the sliding window for the assignment of subsequent intervals until all threads have been assigned neighbors or no more neighbors remain.

Figure 8 shows an example for the process of partial decoding, which does not incur memory overhead on storing decoded vertices in the GPU global memory.

1) First, we decode the number of degrees and intervals from the CGR bit string. Then, all the threads within the warp decode the first interval (18,14). We create a window starting from index 0 with a width of 14. As 14 is larger than the assumed warp size of 10, we split the interval into two intervals: (18,10) and (28,4). In the first round, all threads in the warp work on interval (18,10), and we perform label propagation for that interval.
2) In the second round, we address the remaining interval (28,4) by resizing the window size to 4 and setting the starting index of the window to 0. In this way, the first 4 threads in the warp are assigned neighbors 28, 29, 30, and 31. Then, we decode the next interval (37,3) and assign the corresponding neighbors to lanes 4-7 by moving the window four spaces to the right.
3) After all intervals are processed, we treat residuals as special intervals with a length of 1. The residuals 12, 15, and 54 are assigned to the last three threads by moving the window one space at a time. As all intervals and residuals are assigned, we run label propagation on all threads in the second round.

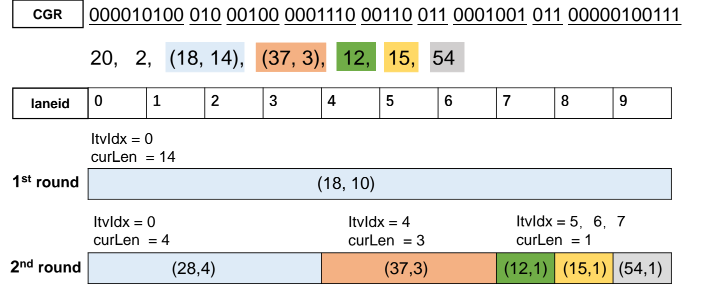

Fig. 8. An example of compressed label propagation.

Addressing Low-Degree Vertices. The decoding process is inherently sequential by design. A significant number of low-degree vertices exist, and decoding them results in inefficient use of thread resources, as their workloads are insufficient to fully leverage the available parallelism. To avoid such issue, we choose not to compress low-degree vertices. Nonetheless, this approach could potentially result in increased memory consumption, thereby negatively impacting performance since greater memory usage can lead to higher costs associated with memory transfers. Hence, we measure the compression ratio when low-degree vertices are not compressed to provide a clear insight into the actual memory usage. Table 3 illustrates the variations in compression ratios when nodes with degrees less than 0, 2, 4, 8, 16, and 32 are not compressed for all datasets. `aligraph` achieves the highest compression ratio, approximately 15 times, for all tested thresholds, due to its notably high average degree of 3991.8. `uk-2002` attains a compression ratio of 9.9 times when all vertices are compressed, but it still reaches a ratio of 8.4 times when only nodes with degrees larger than 32 are compressed. All datasets, with the exception of `roadNet`, maintain a stable compression ratio for the examined thresholds. The underlying reason is that `roadNet` represents a road network, where the degree distribution does not conform to the power law, and the majority of nodes possess low degrees.

Thus, we opt to leave vertices with a degree less than 32 uncompressed in order to minimize the decoding overheads. By doing so, we achieve several advantages. First, we effectively reduce the workload imbalance among threads, resulting in a more even distribution of work and allowing threads to complete their tasks in a more synchronized manner, ultimately reducing idle time. Second, we lower the

TABLE 3
Compression ratio with a degree threshold

Dataset	≤ 0	≤ 2	≤ 4	≤ 8	≤ 16	≤ 32
dblp	1.78	1.73	1.72	1.70	1.61	1.56
roadNet	2.03	2.03	2.01	1.24	1.0	1.0
youtube	1.93	1.93	1.86	1.83	1.77	1.68
aligraph	15.43	15.43	15.43	15.43	15.43	15.43
livej	2.83	2.81	2.73	2.53	2.49	2.45
uk-2002	9.90	9.84	9.52	9.30	8.71	8.43
wiki-en	6.32	6.32	6.28	6.23	6.21	5.99
twitter	2.05	2.03	2.03	2.02	2.01	1.97

overhead associated with the decoding process, as uncompressed vertices do not require the additional processing steps involved in decoding. This leads to faster execution and reduced resource consumption. Lastly, as observed earlier, compressing low-degree vertices does not significantly improve the compression ratio. By keeping them uncompressed, we maintain a reasonable compression ratio while simplifying the decoding process and reducing the complexity of the overall algorithm. This approach results in a more efficient and optimized utilization of thread resources.

6 EXPERIMENTAL EVALUATION

We evaluate the performance of our solutions through experiments against the state-of-the art methods and the in-house LP system of TAOBAO. We present experimental results to answer the following questions:

- How much is the improvement of our *in-memory* solution against the state-of-the-art methods on multi-core CPUs and GPUs? (Section 6.2)
- Is each of the proposed in-memory optimizations effective? (Section 6.3)
- How much is the improvement of our *out-of-core* solution against the state-of-the-art GPU solutions? (Section 6.4)
- What are the advantages of our LP solutions against the in-house LP system in TAOBAO? (Section 6.5)

6.1 Experimental Setup

Datasets. dblp, roadNet, youtube, and livej were obtained from Stanford Network Dataset Collection [45], twitter, uk-2002, wiki-en were obtained from Kobkenz Network Collection [46], uk-2002 was obtained from Laboratory for Web Algorithmics [47], and aligraph was an open dataset provided at Tianchi [48]. We use the method proposed in [11] to encode each dataset and obtains the compressed graph format for out-of-core evaluations. We report the encoding cost in Table 4. Note that the encoding process is conducted offline and the compressed graph format can be shared by different processing tasks as discussed in [11], [37], [38] to take advantage of saved memory cost.

LP algorithms. We evaluate three common LP algorithms. For classic LP, we run 20 iterations. For LLP, we set $\gamma = 2^i$, $i = 0, 1, 2, \ldots, 9$, and run 20 iterations for each γ. For SLP, set the maximum number of labels of each vertex to 5, and run 20 iterations.

Compared Approaches. We compare our proposed approaches with the state-of-the-art GPU-based solutions.

TABLE 4
Dataset Statistics

Dataset	V	E	Ave-Degree	Encoding
dblp	317,080	1,049,866	6.6	0.67s
roadNet	1,965,206	2,766,607	2.8	0.27s
youtube	1,134,890	2,987,624	5.2	1.65s
aligraph	14,933	29,804,566	3991.8	29.22s
livej	3,997,962	34,681,189	17.3	28.76s
uk-2002	18,520,486	298,113,762	16.1	55.69s
wiki-en	15,150,976	378,142,420	24.9	88.77s
twitter	41,652,230	1,468,365,182	35.3	281.37s

- TG is the implementation of classic LP provided in the TigerGraph framework [49] on multi-core CPUs. The number of threads is set to 12 to match with the number of physical cores by default.
- Ligra represents the LP algorithms implemented based on the Ligra framework [50] on multi-core CPUs. The number of threads is set to 12 to match with the number of physical cores by default.
- OMP represents the LP implementations using OpenMP.
- G-Sort [15] is the state-of-the-art GPU solution for classic LP with the segmented sort approach.
- G-Hash [51] is an extended version of G-Sort by employing shared memory hash table for label counting.
- GLP is our proposed in-memory GPU solution for LP without the compression technique.
- CGLP is our proposed out-of-core GPU solution for compressed LP using two kernels. One decodes the adjacency list from CGR, and the other does label propagation directly on the adjacency list.
- CGLP+ is our proposed out-of-core GPU solution which adopts the partial decoding optimization.

Environment. We conduct two sets of experiments. All codes are compiled by GCC-7.4 and CUDA 10.0 with optimization -O3. The first set (Sections 6.2-6.4) is conducted on a single machine with Intel(R) Xeon(R) W-2133 CPUs, 64GB RAM and two NVIDIA 2080ti GPUs (only one GPU is enabled). The second set (Section 6.5) compares GLP and CGLP+ running on the above single machine setup (two GPUs are enabled and connected by Nvlink) with the current in-house solution used in TAOBAO running on a high end machine, which is equipped with 4 Intel(R) Xeon(R) Platinum 8168 CPUs and 512GB RAM.

6.2 In-memory performance evaluation

In this section, we evaluate the in-memory performance of the compared approaches by running LP algorithms on the five datasets that can fit into the memory of one GPU. The other three datasets wiki-en,twitter and twitter will be used in the out-of-core evaluation. We benchmark the solutions by comparing their speedup ratios over OMP. The results are demonstrated in Figure 9(a), Figure 9(b) and Figure 9(c) respectively.

For classic LP comparisons, OMP and Ligra show similar performance on most of the datasets, and both approaches are more efficient than TG. For those datasets, G-Sort performs slightly better than G-Hash. This is because G-Sort employs an efficient segmented sort kernel from CUB [5]. The segmented sort achieves the best performance when the size of each segment is small. As the

5. https://nvlabs.github.io

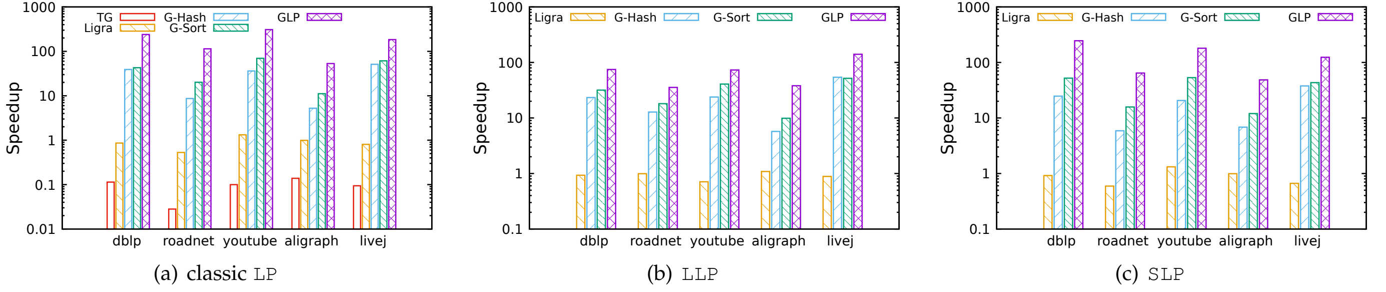

Fig. 9. Speedup of all compared approaches over the OMP baseline.

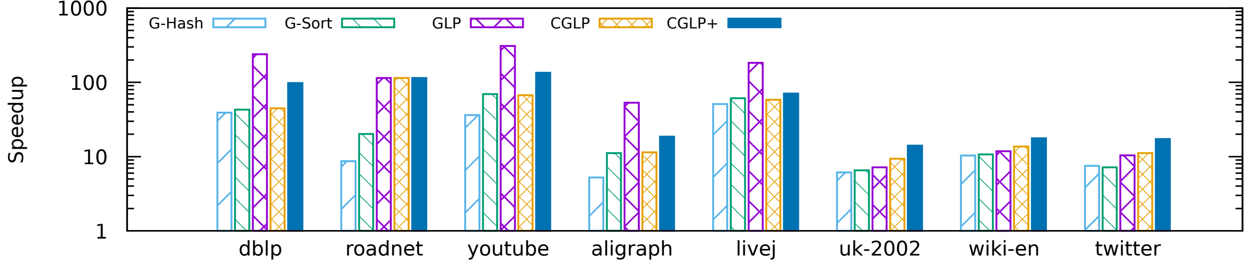

Fig. 10. Speedup of all GPU-based compared methods over the OMP baseline.

neighbors of a vertex correspond to a segment, G-Sort demonstrates good performance for small neighborhoods. Furthermore, both G-Sort and G-Hash require additional global memory equivalent to the graph size to process the neighborhood labels. The memory consumption of GLP is 20% and 24% lower than that of G-Hash and G-Sort on average respectively, as illustrated in Table 5. GLP achieves the best performance among the three. G-Hash deploys a hash table in the GPU global memory which records the occurrence of labels in the neighbors. Each neighbor requires at least one memory access to the hash table, resulting in frequent global memory accesses. However, GLP mitigates this issue through the efficient shared memory approach reducing redundant global access. G-Sort deploys the segmented sort for MFL computing, which necessitates multiple scans to rearrange the neighbors. Whereas GLP requires only one scan of the neighbors. Furthermore, GLP deploys warp-level intrinsics to efficiently handle low degree vertices, which are not optimized in G-Hash and G-Sort. In particular, GLP achieves 4.5x and 7x speedup over G-Sort and G-Hash on average, respectively. For LLP and SLP, the results are consistent with those of classic LP. In summary, the results have demonstrated that GLP significantly outperforms the state-of-the-art solutions on both CPUs and GPUs.

TABLE 5
GPU Memory consumption for in-memory datasets(MB)

Dataset	G-Hash	G-Sort	GLP
dblp	52.12	55.15	42.12
roadNet	180.50	199.22	154.11
youtube	157.25	168.08	128.75
aligraph	1137.52	1137.65	853.28
livej	1475.47	1513.51	1144.73

Since the performance results of the three LP variants are highly similar, we use classic LP to benchmark the compared methods in the rest of this paper. Furthermore, we omit the results of Ligra and TG for subsequent evaluation since they have significant inferior performance compared with the GPU-based solutions.

6.3 Effectiveness of the in-memory optimizations

To demonstrate the effectiveness of our proposed optimizations, we compare the following approaches for computing the MFL in the classic LP algorithm on GPUs:

- *global*. A hash table in the global memory is employed for each vertex to count the neighborhood label frequency with the help of GPU caching mechanism, which is used in G-Hash [51].
- *smem*. Our proposed approach of combing CMS and HT (Section 4.1) to minimize workloads for counting label frequencies of high degree vertices. We set the high degree vertices as those with degree larger than 128.
- *smem+warp*. The optimization proposed in Section 4.2 to handle multiple low degree vertices with one warp. We set the low degree vertices as those with degree less than 32. It is activated together with *smem* to show the additional improvement.

TABLE 6
Effectiveness of the purposed optimizations (speedup over *global*)

Dataset	dblp	roadNet	youtube	aligraph	livej
smem	1.4x	1.2x	1.6x	7.4x	1.7x
smem+warp	6.1x	13.2x	8.6x	10.1x	3.6x

We activate our optimizations one by one and report the speedup over *global* in Table 6. The *smem* strategy shows significant speedup compared with *global*, which uses global hash table for label counting. There is a special case in aligraph, where *smem* achieves 7.4x speedup over *global* which is much higher than other datasets. The reason is that the aligraph dataset has the largest average degree across all datasets as shown in Table 4, where most of the vertices can benefit from *smem*.

The *warp* strategy adds on top of *smem* and optimizes the process of finding the MFL for low degree vertices. It provides superior speedup on small graphs. Especially in roadNet, *warp* offers an additional 11x speedup. This is due

to the fact that roadNet is a road network and thus each vertex has a small constant degree, which leads to heavy workload imbalance for *global*. On average, combining *smem* and *warp* achieves a speedup of 8.3x over the one without the optimizations.

6.4 Out-of-core performance evaluation

In this section, we evaluate the performance of all GPU-based approaches, including our out-of-core solutions, CGLP and CGLP+, across all datasets, including the three out-of-core datasets in Figure 10.

As the graph size expands, G-Sort's performance declines due to the diminished efficiency of segmented sorting on larger segments, which is caused by an increased neighborhood size. Consequently, G-Sort performs equally or worse than G-Hash. However, GLP surpasses both G-Hash and G-Sort in terms of performance for both in-core and out-of-core datasets.

Next, we compare the performance of our out-of-core solutions with GLP. For the first five datasets, where the data can fit into GPU memory, GLP achieves the highest speedup over CGLP and CGLP+. For instance, in dblp, the speedup achieved by GLP is 239.7x, while CGLP and CGLP+ achieve 44.7x and 97.9x, respectively. This is because CGLP and CGLP+ incur additional costs for decoding, making them less efficient on in-memory datasets.

However, for the last three datasets that cannot fit into GPU memory, CGLP+ achieves the highest speedup. For example, in uk-2002, CGLP+ achieves a 14.1x speedup, while CGLP and GLP achieve 9.4x and 8.3x speedup, respectively. Overall, GLP is the most efficient solution for in-memory datasets, while CGLP+ is the best for out-of-core datasets. In the special case of roadNet, all three methods yield the same results because most vertices in roadNet have a low degree, which skips the compressing process as discussed in Section 5.

In order to further examine the costs associated with processing out-of-core datasets, we profile the expenses related to CPU-GPU data transfer and in-memory LP processing in GLP, CGLP, and CGLP+, as presented in Table 7. The results demonstrate that the data transfer cost constitutes over 60% of the total running time in GLP when handling out-of-core datasets. In contrast, CGLP+ exhibits a significantly lower percentage of data transfer costs, accounting for less than 20%. Generally, CGLP+ reduces the data transfer cost for LP processing on out-of-core datasets by 88.4%. This reduction is attributed to the fact that GLP must repeatedly reload the entire neighbor list from the CPU to the GPU in each iteration, while CGLP and CGLP+ only need to load the compressed adjacency list to the GPU memory once, decoding the neighbor list continually in each iteration instead. Like CGLP+, CGLP also incurs a small memory transfer cost. Because both of them use the same compressed graph format. However, CGLP has a huge computational cost due to decoding. In CGLP+, the computational cost is decreased by 36%, 25%, and 38% in uk-2002,wiki-en, and twitter respectively when compared to CGLP. This reduction underscores the effectiveness and benefits of employing CGLP+ for reducing decoding cost. The reason is that CGLP requires additional loads of decoded adjacency list from GPU memory, whereas CGLP+ integrated decoding and label propagation with only one scan of the compressed adjacency list.

TABLE 7
The percentage of running time of computing and memory transferring on out-of-core datasets

Datasets	GLP		
	GPU compute	PCI-e transfer	total
uk-2002	32.7%(1.62s)	67.3%(3.35s)	4.97s
wiki-en	38.8%(3.23s)	61.2%(5.10s)	8.33s
twitter	39.3%(6.72s)	60.7%(10.38s)	17.1s
Datasets	CGLP		
	GPU compute	PCI-e transfer	total
uk-2002	93.9%(3.58s)	6.0%(0.23s)	3.81s
wiki-en	90.8%(6.55s)	9.1% (0.66s)	7.21s
twitter	90.1%(14.35s)	9.8%(1.57s)	15.92s
Datasets	CGLP+		
	GPU compute	PCI-e transfer	total
uk-2002	90.9%(2.30s)	19.1%(0.23s)	2.56s
wiki-en	88.1%(4.91s)	11.9% (0.66s)	5.57s
twitter	84.7%(8.73s)	15.3%(1.57s)	10.3s

Though CGLP+ incurs a higher computation cost than GLP for all three datasets due to the decoding process, its overall running time remains smaller than that of GLP for out-of-core datasets. This finding highlights the effectiveness of CGLP+ in reducing the costs associated with processing large-scale graphs that exceed available memory.

6.5 Large-scale Fraud Detection Processing

We study a real-world data science pipeline to further demonstrate the superiority of our proposed methods.

Fraud Detection Pipeline in TAOBAO. The overview of the detection pipeline has been presented in Figure 1. The pipeline maintains sliding windows containing the transactions in the past 10-100 days. A graph is constructed for each sliding window connecting the entities in the transactions. The graph sizes of different sliding window configurations are presented in Table 8. Subsequently, LP algorithms are invoked with known fraudulent entities to discover small susceptible clusters. The identified clusters are then fed into more complex algorithms, e.g., graph neural nets [8], to detect suspicious transactions/users in a fine-grained manner. We note that the efficiency bottleneck lies in the LP stage, which takes up a heavy 75% processing overhead of the above automated detection pipeline.

We compare our proposed methods GLP and CGLP+ with the current in-house LP solution used in TAOBAO. As reported in Table 8, GLP cannot support in-memory LP processing on the 10-days workloads unless with two GPUs. Even with two GPUs, GLP cannot maintain an in-memory processing over workloads more than 10 days. In contrast, CGLP+ can support in-memory computation for workloads within 20 days with single GPU and workloads within 40 days with two GPUs.

We report the performance comparison in Figure 11. We report the average elapsed time for one LP iteration. The experiment shows that GLP and CGLP+ achieves 8.2x and 13.2x speedup on average against the current in-house approach with a single GPU. Additionally, GLP and CGLP+ further achieve 1.8x and 1.9x speedup on average with two GPUs. We note that with one GPU, CGLP+ always performs better than GLP for each workload. With two GPUs, GLP outperforms CGLP+ for workloads of 10 days but ends in larger

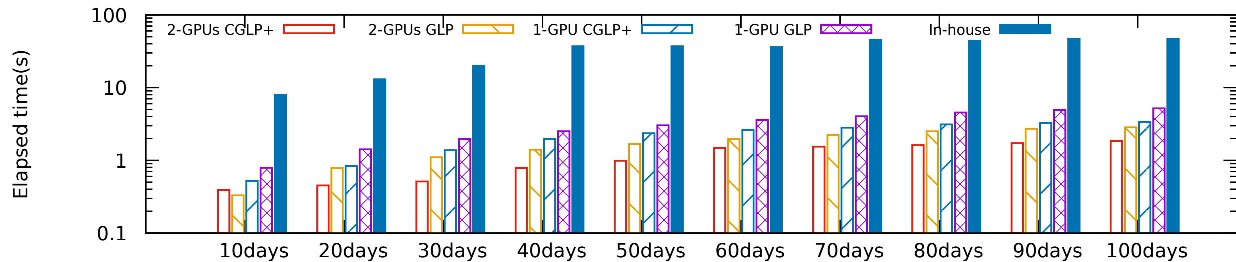

Fig. 11. The elapsed time of using `CGLP+` vs. `GLP` and the current in-house solution of TaoBao for one iteration of `LP`.

TABLE 8
Graph Workloads in TaoBao (OOC is short for out-of-core)

Dataset	**10days**	**20days**	**30days**	**40days**	**50days**
V(M)	460	630	700	770	820
E(B)	1.7	3	4.3	5.5	6.7
`GLP`	2 GPUs	OOC	OOC	OOC	OOC
`CGLP+`	1 GPU	1 GPU	2 GPUs	2 GPUs	OOC
Dataset	**60days**	**70days**	**80days**	**90days**	**100days**
V(M)	880	920	970	990	1010
E(B)	7.9	9	10.1	11	11.6
`GLP`	OOC	OOC	OOC	OOC	OOC
`CGLP+`	OOC	OOC	OOC	OOC	OOC

running time when the workload increases. The reason is that 10-days workload can still be processed in-memory with two GPUs. In addition to performance improvement, the monetary of deploying those methods is always a key factor. The monetary of deploying `GLP` or `CGLP+` is also significantly lower than the in-house solution. The CPUs used in the in-house solution cost $5890(CPU) * 4 = 23560$ dollars, whereas the CPU-GPU setup used in `GLP` costs $617(CPU)+999(GPU) = 1616$ dollars, according to official prices from Intel and NVIDIA.

7 Conclusion

In this paper, we propose a lightweight yet efficient GPU framework for `LP` algorithms. We design flexible APIs to help users deploy `LP` algorithms on GPUs with ease. Our `GLP` surpasses alternative methods when workloads can be accommodated within the GPU global memory, while `CGLP+` excels in comparison to other approaches in out-of-core scenarios. Experiments on real world datasets have validated the effectiveness of our proposed techniques over variants of `LP` algorithms, and have demonstrated that our optimizations achieves significant speedup over the state-of-the-art CPU- and GPU-based `LP` solutions. Furthermore, we assess the performance of `GLP` and `CGLP+` using real workloads within the fraud detection pipeline. Our solutions demonstrate superior advantages in terms of both time and cost efficiency when compared to the in-house solution employed by TaoBao.

References

[1] A. Beutel, L. Akoglu, and C. Faloutsos, "Fraud detection through graph-based user behavior modeling," in *SIGSAC*, 2015, pp. 1696–1697.
[2] "Stop fraud rings in their tracks with neo4j," https://neo4j.com/use-cases/fraud-detection/, accessed: 2020-7-7.
[3] R. Mao, Z. Li, and J. Fu, "Fraud transaction recognition: A money flow network approach," in *CIKM*, 2015, pp. 1871–1874.
[4] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *WWW*, 2019, pp. 417–426.
[5] M. Wang, C. Wang, J. X. Yu, and J. Zhang, "Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework," *PVLDB*, vol. 8, no. 10, pp. 998–1009, 2015.
[6] S. E. Garza and S. E. Schaeffer, "Community detection with the label propagation algorithm: a survey," *Physica A: Statistical Mechanics and its Applications*, vol. 534, p. 122058, 2019.
[7] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical review E*, vol. 76, no. 3, p. 036106, 2007.
[8] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *TKDE*, vol. 30, no. 9, pp. 1616–1637, 2018.
[9] H. Liu and H. H. Huang, "Enterprise: breadth-first graph traversal on gpus," in *SC*, 2015, pp. 1–12.
[10] H. Liu, H. H. Huang, and Y. Hu, "ibfs: Concurrent breadth-first search on gpus," in *SIGMOD*, 2016, pp. 403–416.
[11] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *SIGMOD*, 2019, pp. 775–792.
[12] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang, "Realtime top-k personalized pagerank over large graphs on gpus," *PVLDB*, vol. 13, no. 1, pp. 15–28, 2019.
[13] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A gpu approach," *TKDE*, vol. 29, no. 3, pp. 513–528, 2016.
[14] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *IPDPS*, 2011, pp. 568–579.
[15] Y. Kozawa, T. Amagasa, and H. Kitagawa, "Gpu-accelerated graph clustering via parallel label propagation," in *CIKM*, 2017, pp. 567–576.
[16] J. Xie and B. K. Szymanski, "Community detection using a neighborhood strength driven label propagation algorithm," in *NSW*, pp. 188–195.
[17] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "Gpu-accelerated graph label propagation for real-time fraud detection," in *SIGMOD*, 2021, pp. 2348–2356.
[18] M. J. Barber and J. W. Clark, "Detecting network communities by propagating labels under constraints," *Physical Review E*, vol. 80, no. 2, p. 026129, 2009.
[19] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *WWW*, 2011, pp. 587–596.
[20] J. Xie, B. K. Szymanski, and X. Liu, "Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process," in *ICDMW*, 2011, pp. 344–349.
[21] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *WSDM*, 2013, pp. 507–516.
[22] J. Xie and B. K. Szymanski, "Labelrank: A stabilized label propagation algorithm for community detection in networks," in *NSW*, 2013, pp. 138–143.
[23] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *SIGPLAN Notices*, vol. 46, no. 8, 2011, pp. 267–276.
[24] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *TPDS*, vol. 25, no. 6, pp. 1543–1552, 2013.
[25] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *SIGPLAN Notices*, vol. 47, no. 8, 2012, pp. 117–128.
[26] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *SIGPLAN Notices*, vol. 51, no. 8, 2016, p. 11.

[27] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," *CSUR*, vol. 50, no. 6, p. 81, 2018.

[28] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on gpus," in *HPDC*, 2014, pp. 239–252.

[29] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh, "Incremental maintenance for non-distributive aggregate functions," in *PVLDB*, 2002, pp. 802–813.

[30] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *PACT*, 2015, pp. 39–50.

[31] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "Emogi: efficient memory-access for out-of-memory graph-traversal in gpus," *PVLDB*, vol. 14, no. 2, pp. 114–127, 2020.

[32] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on gpus with unified memory," *PVLDB*, vol. 13, no. 7, pp. 1119–1133, 2020.

[33] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu, "A yoke of oxen and a thousand chickens for heavy lifting graph processing," in *PACT*, 2012, pp. 345–354.

[34] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a gpu," in *PACT*, 2017, pp. 233–245.

[35] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication." in *USENIX Annual Technical*, 2017, pp. 195–207.

[36] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *EuroSys*, 2020, pp. 1–16.

[37] K. Kaczmarski, P. Przymus, and P. Rzka.zewski, "Improving high-performance gpu graph traversal with compression," in *ADBIS*, 2015, pp. 201–214.

[38] H. Yin, Y. Shao, X. Miao, Y. Li, and B. Cui, "Scalable graph sampling on gpus with compressed graph," in *CIKM*, 2022, pp. 2383–2392.

[39] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *CSUR*, vol. 45, no. 4, pp. 1–35, 2013.

[40] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[41] L. A. Adamic and B. A. Huberman, "Power-law distribution of the world wide web," *science*, vol. 287, no. 5461, pp. 2115–2115, 2000.

[42] J. Markussen, L. B. Kristiansen, P. Halvorsen, H. Kielland-Gyrud, H. K. Stensland, and C. Griwodz, "Smartio: Zero-overhead device sharing through pcie networking," *TOCS*, vol. 38, no. 1-2, pp. 1–78, 2021.

[43] K. Eshghi and R. Micheloni, "Ssd architecture and pci express interface," *Inside solid state drives (SSDs)*, pp. 19–45, 2013.

[44] U. Farooq, Z. Marrakchi, H. Mehrez, U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*, pp. 7–48, 2012.

[45] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[46] J. Kunegis, "Konect: the koblenz network collection," in *WWW*, 2013, pp. 1343–1350.

[47] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," in *SIGIR Forum*, vol. 42, no. 2, 2008, pp. 33–38.

[48] Z. Du, X. Wang, H. Yang, J. Zhou, and J. Tang, "Sequential scenario-specific meta learner for online recommendation," *arXiv preprint arXiv:1906.00391*, 2019.

[49] "Tigergraph," https://www.tigergraph.com/, 2020.

[50] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *PPoPP*, 2013, pp. 135–146.

[51] "Ghash," https://github.com/ykzw/galp, 2020, accessed: 2020-07-07.

Chang Ye is a PhD student with the School of Computing and Information Systems, Singapore Management University (SMU). He received his BSc and MSc degrees in the area of Electronic Engineering from the Xidian University, in 2014 and 2018, respectively. His research interests are graph analytics and heterogeneous computing.

Yuchen Li is an assistant professor with the School of Computing and Information Systems, Singapore Management University (SMU). He received double BSc degrees in applied math and computer science (both with first-class honors) and a Ph.D. degree in computer science from the National University of Singapore (NUS), in 2013 and 2016, respectively. His research interests include graph analytics and heterogeneous computing. He published in top database and data mining venues such as SIGMOD, VLDB, ICDE, KDD, and TheWebConf. He received best paper awards in KDD and a Lee Kong Chian fellowship.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is a Professor in School of Computing, National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Zhao Li received the Ph.D. degree in computer science from the University of Vermont with the Excellent Graduate Award in 2012. He is currently an adjunct professor with Zhejiang University, Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China. He is also the funder of Link2Do Technology Ltd., Hangzhou. He has published several papers in prestigious conferences and journals, including the World Wide Web Conference (WWW), the AAAI Conference on Artificial Intelligence (AAAI), IEEE International Conference on Data Engineering (ICDE), International Conference on Very Large Data Bases (VLDB), IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING (TKDE), and so on. His research interest lies in reinforcement learning, big-data-driven security, and large-scale graph computing.

Jianling Sun is a professor at the School of Computer Science and Technology, Zhejiang University. He received his PhD degrees in computer science from Zhejiang University, China in 1993. His research interests include Database Systems, Distributed Computing, and Machine Learning. He currently serves as the director of the Lab of Next Generation Database Technologies of Alibaba-Zhejiang University Joint Institute of Frontier Technologies.