6-2021

GPU-accelerated graph label propagation for real-time fraud detection

Chang YE
Singapore Management University, changye.2020@phdcs.smu.edu.sg

Yuchen LI
Singapore Management University, yuchenli@smu.edu.sg

Bingsheng HE

Zhao LI

Jianling SUN

Citation

1

GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection

Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, Jianling Sun*
Singapore Management University, National University of Singapore, Alibaba Group, Zhejiang University
changye.2020@phdcs.smu.edu.sg,yuchenli@smu.edu.sg
hebs@comp.nus.edu.sg,lizhao.lz@alibaba-inc.com,sunjl@zju.edu.cn

ABSTRACT

Fraud detection is a pressing challenge for most financial and commercial platforms. In this paper, we study the processing pipeline of fraud detection in a large e-commerce platform of TaoBao. Graph label propagation (LP) is a core component in this pipeline to detect suspicious clusters from the user-interaction graph. Furthermore, the run-time of the LP component occupies 75% overhead of TaoBao's automated detection pipeline. To enable real-time fraud detection, we propose a GPU-based framework, called GLP, to support large-scale LP workloads in enterprises. We have identified two key challenges when integrating GPU acceleration into TaoBao's data processing pipeline: (1) programmability for evolving fraud detection 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. We propose novel GPU-centric optimizations by leveraging the community as well as power-law properties of large graphs. Extensive experiments have confirmed the effectiveness of our proposed optimizations. With a single GPU, GLP supports a real billion-scale graph workload from the fraud detection pipeline of TaoBao and achieves 8.2x speedup to the current in-house distributed solution running on high-end multicore machines.

ACM Reference Format:
Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3448016.3452774

1 INTRODUCTION

Large-scale 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. Graph analytics have been very popular in fraud detection in financial and commercial platforms [6, 24]. The timeliness of detecting frauds is an important factor for fraud detection systems. Meanwhile, the size and complexity of large graphs pose great challenges to many online platforms for deploying real-time analytics. A common practice is to employ graph clustering that divides a large graph into smaller partitions for downstream analytics tasks such as those in fraud detection. Among existing graph clustering approaches, Label Propagation (LP) is one of the most effective and efficient algorithms [36]. Furthermore, many LP variants have been proposed to address graph problems in different applications, such as community detection [41], graph compression [7], and many others [34, 35]. Despite having a linear complexity theoretically [28], the overhead of executing LP on large graphs is still a major bottleneck for time-critical applications.

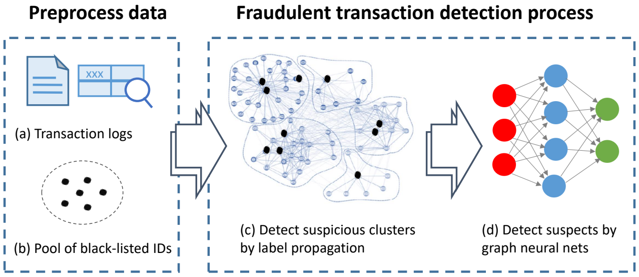

Preprocess data
Fraudulent transaction detection process
xxx
(a) Transaction logs
(b) Pool of black-listed IDs
(c) Detect suspicious clusters by label propagation
(d) Detect suspects by graph neural nets

Figure 1: TaoBao's data pipeline for fraud detection

Fraud Detection Pipeline in TaoBao (Figure 1). On TaoBao's e-commerce platform, the LP algorithm and its variants are used as a core component for analyzing the user-product network to detect fraudulent transactions. Sliding windows of recent purchases/clicks form transaction networks, which 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 [9], to discover new frauds. It is worth-noting that the LP component occupies 75% overhead of TaoBao's automated detection pipeline. Real-time fraud detection is a pressing challenge today as 21.4% of traffic to e-commerce portals are malicious bots in 2018 [1]. Hence, speeding up the LP component will drastically improve the detection latency and enable more responsive measures to stop financial losses.

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 [21, 23], pagerank [13, 30] and network motif detection [12, 20]. These existing works 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 [29], or assign a thread to independently extend a candidate motif by adding one neighborhood vertex in network motif detection [20]. In contrast, for each iteration of

*Zhao Li is the corresponding author.

SIGMOD '21, June 18–27, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452774

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. The research gap and practical impact motivate us to propose an efficient parallel solution for optimizing the LP algorithm on GPUs.

Although there have been some pioneer studies on parallelizing LP on GPUs [17, 33], we identify two major drawbacks when developing GPU-based LP for TaoBao:

- Existing works only optimize the classical LP algorithm [28] on GPUs. However, many variants of LP are used by data engineers in TaoBao to develop strategies to detect evolving fraud patterns. Implementing a correct and efficient GPU programs 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.
- Existing works simply leverage the raw computing power and high memory bandwidth of GPUs to accelerate MFL computation. As the workload is not associative, 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 [17, 33]. 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 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.

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. 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 fraud detection 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 two novel optimizations for the MFL computation. Our optimizations take advantages of the characteristics of LP algorithms, which are overlooked by the existing works.

First, we optimize memory performance even if a neighbor set does *not* fit into the shared memory. 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 [39]. To maximize shared memory processing, 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.

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 requirement on the data science pipeline. To the best of our knowledge, this is the first GPU framework that can support a range of LP algorithms. (Section 3)
- 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. (Section 4)
- Extensive experiments confirm: (1) GLP achieves significant speedups over existing CPU and GPU approaches for LP; (2) GLP processes TaoBao's fraud detection workloads on a graph of over 10 billion edges with a single GPU, while offers 8.2x speedup to the current in-house solution running on high-end multicore machines. We show that GLP can be an efficient and cost-effective solution towards real-time fraud detection pipelines. (Section 5)

2 PRELIMINARIES

2.1 Label Propagation (LP)

Fraud detection is critical in e-commerce platforms with a lot of efforts devoted to this area (see survey [27]). Graph analytics have recently been used to support fraud detection [6, 24]. In the following, we introduce the notations used in LP algorithms.

Given a graph $G = (V, E)$, where V represents the vertex set and E represents the edge set. The *incoming* neighbor set of a vertex v is denoted as $N(v)$ and the *outgoing* neighbor set of v is denoted as $N'(v)$. When the context is clear, we refer to neighbors of a vertex as the incoming neighbors. We present the classical LP [28] as a brief introduction to LP algorithms. Every vertex v_i is first initialized with a unique label L_i representing its community 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 community. T

We note that many variants of LP are proposed [5, 7, 34, 40, 41]. 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 enabling customization of different LP variants.

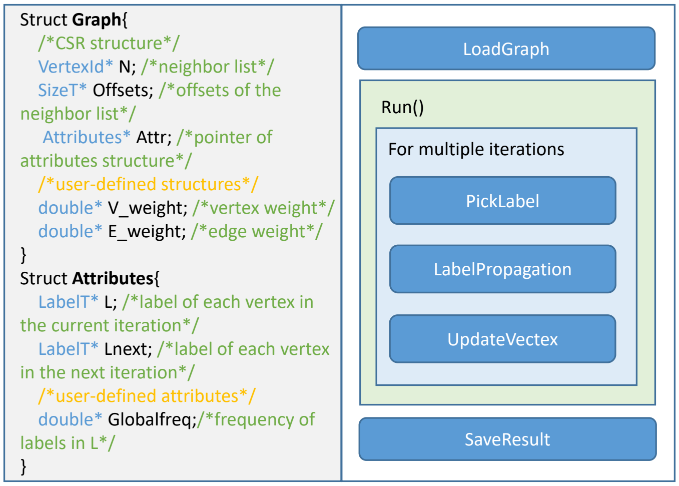
2.2 Related Studies

We discuss three related areas: (1) GPU-based label propagation; (2) Optimizing workload balance for GPU graph processing; and (3) Optimizing memory access for GPU graph processing.

GPU-based label propagation Existing works leverage the high memory bandwidth of GPUs to load the labels of neighbors for each vertex [17, 33]. 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.[14] 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 [42]. 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 [37]. Liu et al. [22] introduce iBFS for processing concurrent BFS where a warp voting technique is used to balance the workloads. Sha et al. [29] propose a GPU graph traverse approach on compressed graphs, they propose two-phase traversal and task stealing for better thread scheduling. Many scheduling approaches have been proposed for other graph applications, see [31] 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.

Optimizing memory access on GPU graph processing: Recent works on GPU-based graph processing propose solutions for addressing irregular memory accesses. Cusha [16] 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 [26]. However, the task of extracting MFL in LP is naturally non-associative, which renders the infeasibility of adopting Cusha for LP. Khorasani et al.[15] 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.



```
Struct Graph{
   /*CSR structure*/
   VertexId* N; /*neighbor list*/
   SizeT* Offsets; /*offsets of the
neighbor list*/
    Attributes* Attr; /*pointer of
attributes structure*/
   /*user-defined structures*/
   double* V_weight; /*vertex weight*/
   double* E_weight; /*edge weight*/
}
Struct Attributes{
   LabelT* L; /*label of each vertex in
the current iteration*/
   LabelT* Lnext; /*label of each vertex
in the next iteration*/
   /*user-defined attributes*/
   double* Globalfreq;/*frequency of
labels in L*/
}
```

Figure 2: Overview of GLP

3 THE GLP FRAMEWORK

In this section, we first present the overview of our proposed system GLP and sample APIs. Subsequently, we demonstrate some examples on how to implement different LP algorithms with GLP.

3.1 Overview of GLP and APIs

We follow two 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 in order to satisfy the real-time requirements from fraud detection..

Figure 2 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:

- PickLabel. This component is responsible for deciding a label for each vertex according to 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 the implementation details in Section 4.
- UpdateVertex. Given a vertex and a label picked from *Label-Propagation*, this component is responsible for updating the status of the vertex with a user-defined strategy.

Note that the aforementioned workflow can be naturally implemented on a CPU-GPU heterogeneous environment. In the case of large graphs that cannot fit into the GPU memory, the CPUs can coordinate the CPU-GPU graph data movement as well as handle PickLabel and UpdateVertex. The heavy lifting of processing LabelPropagation is then handled by one or multiple GPUs.

Table 1: 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 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 $score$.

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 1. 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 strategies against evolving fraud patterns, we present three commonly used LP variants under the GLP framework.

- Classic LP. The classic LP algorithm introduced in Section 2.1
- LLP (The layered LP algorithm [7]). 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 and v is the overall number of vertices having the same label with l. The classic LP choose l maximizing k, whereas LLP chooses the label maximizing val.
- SLP (The speaker-listener LP algorithm [38]). 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 our optimizations for implementing labelPropogation on computing MFL under the GLP framework. We focus on two types of vertices, namely high and low degree vertices, as they are the major issues for memory access overhead and workload imbalance respectively.

4.1 Handling High Degree Vertices

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 u with memory size equivalent to u'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 community. 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 [10] 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 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 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(HT) and the maximum score estimated by CMS as s(CMS). If $s(\text{HT}) \geq s(\text{CMS})$, 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

```
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  $freq := atomicAdd(\mathsf{HT}, l, weight)$
6  if unsuccessful insertion then
7      $freq := atomicAdd(\mathsf{CMS}, l, weight)$
8      $cm\_score :=$ LabelScore$(v, l, freq)$
9  else
10     $ht\_score :=$ LabelScore$(v, l, freq)$
11 $s(\mathsf{HT}) := BlockReduce(ht\_score, max())$
12 $s(\mathsf{CMS}) := BlockReduce(cm\_score, max())$
13 if $s(\mathsf{HT}) \geq s(\mathsf{CMS})$ then
14     if $s(HT) == ht\_score$ then
15         $Lnext[v] := l$
16 else
17     if $l \notin \mathsf{HT}$ then
18         $freq := GlobalInsert(\mathsf{GHT}, l, weight)$
19         $gt\_score :=$ LabelScore$(v, l, freq)$
20     else
21         $gt\_score := ht\_score$
22     $s(\mathsf{GHT}) := BlockReduce(gt\_score, max())$
23     if $s(\mathsf{GHT}) == gt\_score$ then
24         $Lnext[v] := l$
```

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 $s(\mathsf{HT}) \geq s(\mathsf{CMS})$ (Lines 13-15). Otherwise, we insert l into the 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. In the following, we show that this strategy can effectively reduce the global memory accesses with a high probability.

Theoretical Analysis: The intuition behind this approach is that high degree vertices may not have a large number of distinct labels after some iterations, thus a sufficiently large capacity of HT is enough to capture the MFL. 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. 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 $f_{\min} \leq f(l) \leq f_{\max}$. 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 $f(l^*) = f_{\max}$. Then $\mathbb{P}[l^* \notin \mathsf{HT}] \leq (1 - \frac{h}{m+k})^{2k}$ where $k = \frac{f_{\max}-1}{2}$.*

Proof. 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 $f_{\max}$ times at random positions. Hence, $l^* \notin \mathsf{HT}$ if and only if l^* does not appear in the first h positions of s. It then follows that:

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

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 $\frac{m+i-h}{m+i} \cdot \frac{m+n-i-h}{m+n-i} \leq (\frac{m+n/2-h}{m+n/2})^2 \; \forall i \in [0, n]$. Substitute $k = \frac{f_{\max}-1}{2}$ 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 $N(v)$), $\mathbb{P}[l^* \notin \mathsf{HT}] \to e^{-h}$ for large $f_{\max}$, 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 $l^* \in \mathsf{HT}$. Such a scenario implies global random accesses are needed. To establish the theoretical result, we denote $f(\mathsf{HT})$ 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. *$\mathbb{P}[\max_l g(l) > f_{\max}] \leq m\delta$ where $\delta = 2^{-d}$.*

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

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

Equation 1 holds due to the properties of the CMS [10] when w is set to $2s$. Hence, $\mathbb{P}[\max_L g(L) \geq f_{\max}] \leq m\delta$ 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 $f_{\max} \to \infty$ and $m \leq \frac{f_{\max}-1}{2}$.*

[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.

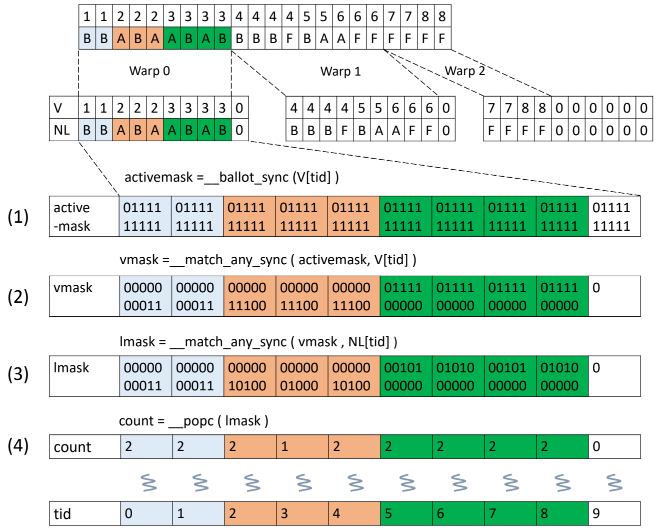


Figure 3: An example of the warp-centric approach. Each number in V represents a node ID v and the number in NL below v represents a label from a neighbor of v. Different colors represent different vertices handled by the threads in warp 0 respectively. The bit mask for thread tid represents all threads in warp 0 having the same workload as tid.

The proof naturally follows by combing Lemma 1 and Lemma 2. In practice, the maximum frequency $f_{\max}$ 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

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 [4]. One-warp-one-vertex strategy will result in many idle threads and under-utilize the computing resources of GPUs.

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 this end, 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 3.

(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 as no valid label is assigned.
(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 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 by **__match_any_sync**.
(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 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 EXPERIMENTAL EVALUATION

This section evaluates the performance of our system through experiments against state-of-the art solutions and the current system in TaoBao. Section 5.1 describes the experimental setup. We present experimental results to answer the following questions:

- What is the overall performance improvement of our proposed solution GLP against state-of-the-art solutions on multi-core CPUs and GPUs (Section 5.2)?
- Is each of the proposed optimizations effective (Section 5.3)?
- What are the advantages of GLP against the current system in TaoBao (Section 5.4)?

5.1 Experimental Setup

Datasets. `dblp`, `roadNet`, `youtube`, and `ljournal` were obtained from Stanford Network Dataset Collection [19], `twitter`, `uk-2002`,

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

Table 2: Datasets.

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

wiki-en were obtained from Kobkenz Network Collection [18], uk-2002 was obtained from Laboratory for Web Algorithmics [8], and aligraph was an open dataset provided at Tianchi [11].

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.

- TG is an implementation of the classic LP algorithm provided in TigerGraph framework [3] on multi-core CPUs.
- Ligra represents the implementations of the LP algorithms based on the Ligra framework [32] on multi-core CPUs.
- OMP represents the LP implementations using OpenMP.
- G-Sort [17] is the state-of-the-art GPU solution for the classic LP with the segmented sort approach.
- G-Hash [2] is an extended version of G-Sort by employing shared memory hash table for label counting.
- GLP is the approach proposed in this work.

We adopt the above baseline implementations from their inventors. Note that TG only supports the classic LP, we thus omit its results for LLP and SLP. The original implementations of G-Sort and G-Hash also support the classic LP only. We hence extend their codes to support LLP and SLP.

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 5.2-5.3) is conducted on a single machine with Intel(R) Xeon(R) W-2133 CPUs, 64GB RAM and one NVIDIA Titan V GPU. The second set (Section 5.4) compared GLP running on the above single machine setup with the current in-house distributed solution used in TaoBao running on a cluster of 32 machines, where each machine is equipped with 4 Intel(R) Xeon(R) Platinum 8168 CPUs and 512GB RAM.

5.2 Comparison with the State of the Art

In this section we evaluate the overall performance of our proposed approaches OMP and GLP with the CPU and GPU LP solutions over the LP algorithms. We benchmark the solutions by comparing their speedup ratios over OMP. The results are demonstrated in Figure 4, Figure 5 and Figure 7 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 smaller datasets, G-Sort outperforms G-Hash. As the datasets get larger, i.e, twitter and wiki-en, G-Hash shows the same or even better performance than G-Sort. 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 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. Compared with the baselines, GLP achieves the best performance and does not impose a large memory overhead due to our proposed optimizations in Section 4. 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. Furthermore, the APIs provided in GLP allow data engineers to quickly deploy customized LP algorithms without domain knowledge of GPU optimizations.

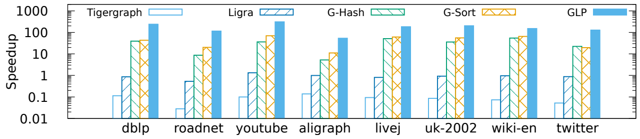


Figure 4: Speedup of all compared approaches over the OMP baseline for classic LP

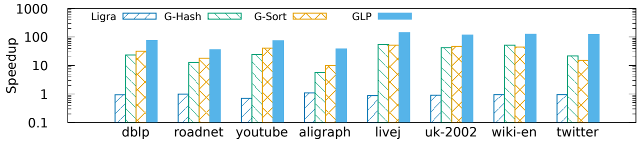


Figure 5: Speedup of all compared approaches over the OMP baseline for LLP

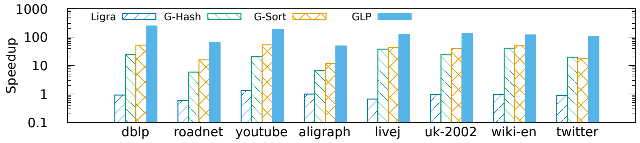


Figure 6: Speedup of all compared approaches over the OMP baseline for SLP

5.3 Effectiveness of the 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 [2].
- *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

[5] https://nvlabs.github.io

Table 3: Effectiveness of the purposed optimizations.

Dataset	dblp	roadNet	youtube	aligraph
smem	1.4x	1.2x	1.6x	7.4x
smem+warp	6.1x	13.2x	8.6x	10.1x
Dataset	ljournal	uk-2002	wiki-en	twitter
smem	1.7x	3.4x	2.2x	4.1x
smem+warp	3.6x	5.6x	3.3x	5.6x

degree vertices as those with degree less than 32. It is activated together with *smem* to show the additional improvement.

We activate our optimizations one by one and report the speedup over *global* in Table 3. The *smem* strategy shows significant speedup compared with *global*, which uses global hash table for label counting. *smem* provides higher speedup in larger datasets, i.e., 3.4x in uk-2002, 2.2x in twitter and 4.1x in wiki-en. This is because there are more high degree vertices in larger datasets, which can benefit more from the *smem* optimization and gain significant speedup. 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 2, 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 6.9x over the one without the optimizations.

5.4 Large-scale Fraud Detection Processing

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

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 4. Subsequently, a LP algorithm is invoked with the stored seeds to discover small susceptible clusters. We note that the efficiency bottleneck lies in the LP stage, which takes up a heavy 75% processing overhead of the automated detection pipeline.

Table 4: Sliding Window Workloads in TaoBao**.**

Dataset	**10days**	**20days**	**30days**	**40days**	**50days**
V(millions)	460	630	700	770	820
E(billions)	1.7	3	4.3	5.5	6.7
Dataset	**60days**	**70days**	**80days**	**90days**	**100days**
V(millions)	880	920	970	990	1010
E(billions)	7.9	9	10.1	11	11.6

We compare our proposed GLP with the current in-house distributed solution used in TaoBao. For GLP, we use the same environment as described in Section 5.1. The performance comparison between GLP and the current in-house distributed solution is presented in Figure 7. We execute the LP algorithm for 20 iterations on the real graph workloads reported in Table 4 and report the average elapsed time for one iteration. The experiment shows that GLP achieves 8.2x speedup on average against the current in-house distributed approach used in TaoBao. We also note that, as the graph size exceeds the GPU memory, GLP switches to the CPU-GPU heterogeneous mode but the memory transfer overhead is less than 10% of the overhead running time (included in the elapsed time). In addition to performance improvement, the monetary of deploying GLP is also significantly lower than the in-house solution. The CPUs used in each machine of the in-house solution cost $5890(CPU) * 4 = 23560$ dollars, whereas the CPU-GPU setup used in GLP costs $617(CPU) + 2999(GPU) = 3616$ dollars, according to official prices from Intel and NVIDIA. Hence, GLP can also provide substantial monetary savings with one machine to handle the current detention workload efficiently. To verify the efficiency of GLP on a multi-GPU environment, we add one more NVIDIA Titan V GPU to the same single-machine setting. With two GPUs, GLP further achieves 1.8x speedup on average, as shown in Figure 7.

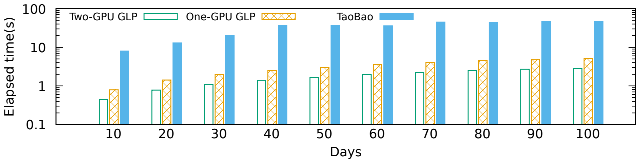


Figure 7: The elapsed time of using GLP **vs. the current in-house solution of** TaoBao **for one iteration of** LP**.**

6 CONCLUSION

From the study of fraud detection pipelines in TaoBao, we identify two key issues towards real-time fraud detection: programmability and efficiency. Programmability is for data engineers to develop strategies against evolving frauds in the e-commerce platform, and efficiency is for the real-time requirement of fraud detection applications. We also observe that LP algorithms are commonly used algorithms but also a time-consuming stage in the fraud detection pipeline. 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 outperforms other approaches by fewer global memory accesses and fine-grain workload scheduling. Experiments on real world datasets have validated the effectiveness of our proposed techniques over variants of LP algorithms, and have demonstrated that the combined optimization achieves significant speedup over the state-of-the-art CPU- and GPU-based LP solutions. Furthermore, we study a real-world fraudulent detection pipeline of a large e-commerce platform to showcase the advantages of GLP over a current enterprise solution in terms of time and monetary efficiency.

Acknowledgment. This work was supported by Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies. Yuchen Li is partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-2-065).

REFERENCES

[1] 2019. Distil Networks: The 2019 Bad Bot Report. https://www.bluecubesecurity.com/wp-content/uploads/bad-bot-report-2019LR.pdf

[2] 2020. Ghash. https://github.com/ykzw/galp. Accessed: 2020-07-07.

[3] 2020. Tigergraph. https://www.tigergraph.com/.

[4] Lada A Adamic and Bernardo A Huberman. 2000. Power-law distribution of the world wide web. *science* 287, 5461 (2000), 2115–2115.

[5] Michael J Barber and John W Clark. 2009. Detecting network communities by propagating labels under constraints. *Physical Review E* 80, 2 (2009), 026129.

[6] Alex Beutel, Leman Akoglu, and Christos Faloutsos. 2015. Fraud detection through graph-based user behavior modeling. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1696–1697.

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*. 587–596.

[8] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. In *ACM SIGIR Forum*, Vol. 42. ACM, 33–38.

[9] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[10] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[11] Zhengxiao Du, Xiaowei Wang, Hongxia Yang, Jingren Zhou, and Jie Tang. 2019. Sequential Scenario-Specific Meta Learner for Online Recommendation. *arXiv preprint arXiv:1906.00391* (2019).

[12] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 International Conference on Management of Data*. 1067–1082.

[13] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel personalized pagerank on dynamic graphs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 93–106.

[14] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Acm Sigplan Notices*, Vol. 46. ACM, 267–276.

[15] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 39–50.

[16] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 239–252.

[17] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2017. GPU-Accelerated Graph Clustering via Parallel Label Propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 567–576.

[18] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 1343–1350.

[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

[20] Wenqing Lin, Xiaokui Xiao, Xing Xie, and Xiao-Li Li. 2016. Network motif discovery: A GPU approach. *IEEE transactions on knowledge and data engineering* 29, 3 (2016), 513–528.

[21] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[22] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 403–416.

[23] Shengliang Lu, Bingsheng He, Yuchen Li, and Hao Fu. 2020. Accelerating exact constrained shortest paths on GPUs. *Proceedings of the VLDB Endowment* 14, 4 (2020), 547–559.

[24] Renxin Mao, Zhao Li, and Jinhua Fu. 2015. Fraud transaction recognition: A money flow network approach. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 1871–1874.

[25] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Acm Sigplan Notices*, Vol. 47. ACM, 117–128.

[26] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental maintenance for non-distributive aggregate functions. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 802–813.

[27] Clifton Phua, Vincent Lee, Kate Smith, and Ross Gayler. 2010. A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119* (2010).

[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[29] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 775–792.

[30] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Real-time top-k personalized pagerank over large graphs on GPUs. *Proceedings of the VLDB Endowment* 13, 1 (2019), 15–28.

[31] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* 50, 6 (2018), 81.

[32] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[33] Jyothish Soman and Ankur Narang. 2011. Fast community detection algorithm with gpus and multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 568–579.

[34] Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM, 507–516.

[35] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. 2014. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 568–579.

[36] Meng Wang, Chaokun Wang, Jeffrey Xu Yu, and Jun Zhang. 2015. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proceedings of the VLDB Endowment* 8, 10 (2015), 998–1009.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11.

[38] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)* 45, 4 (2013), 1–35.

[39] Jierui Xie and Boleslaw K Szymanski. 2011. Community detection using a neighborhood strength driven label propagation algorithm. In *2011 IEEE Network Science Workshop*. IEEE, 188–195.

[40] Jierui Xie and Boleslaw K Szymanski. 2013. Labelrank: A stabilized label propagation algorithm for community detection in networks. In *2013 IEEE 2nd Network Science Workshop (NSW)*. IEEE, 138–143.

[41] Jierui Xie, Boleslaw K Szymanski, and Xiaoming Liu. 2011. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *2011 ieee 11th international conference on data mining workshops*. IEEE, 344–349.

[42] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.