GPU-accelerated Structural Diversity Search in Graphs

Jinbin Huang, Xin Huang, Jianliang Xu, Byron Choi, and Yun Peng

Abstract—The problem of structural diversity search has been widely studied recently, which aims to find out the users with the highest structural diversity in social networks. The structural diversity of a user is depicted by the number of social contexts inside his/her contact neighborhood. Three structural diversity models based on cohesive subgraph models (e.g., k-sized component, k-core, and k-truss), have been proposed. Previous solutions only focus on CPU-based sequential solutions, suffering from several key steps of that cannot be highly parallelized. GPUs enjoy high-efficiency performance in parallel computing for solving many complex graph problems such as triangle counting, subgraph pattern matching, and graph decomposition. In this paper, we provide a unified framework to utilize multiple GPUs to accelerate the computation of structural diversity search under the mentioned three structural diversity models. We first propose a GPU-based lock-free method to efficiently extract ego-networks in CSR format in parallel. Secondly, we design detailed GPU-based solutions for computing k-sized component-based, k-core-based, and also k-truss-based structural diversity scores by dynamically grouping GPU resources. To effectively optimize the workload balance among multiple GPUs, we propose a greedy work-packing scheme and a dynamic work-stealing strategy to fulfill usage. Extensive experiments on real-world datasets validate the superiority of our GPU-based structural diversity search solutions in terms of efficiency and effectiveness.

Index Terms—Structural diversity search, GPU-accelerated graph algorithms, Workload balance optimization.

1 Introduction

Graph is an important data model for representing and analyzing relations between entities. As a typical example, users and their relationships in online social networks can be formalized as graphs. An important concept of structural diversity [1] refers to the number of social contexts inside a user's ego-network (subgraph induced by his/her one-hop neighbors). In the literature, there exist a wide range of structural diversity search studies [2], [3], [4], [5], [6], [7], the goal is to find out the top-t users with largest structural diversity in a social network, which has broad applications in user recruitment, political campaigns, promotion of health practices, and commercial marketing [1].

To solve the top-t structural diversity search problem for different structural diversity models (e.g., k-sized component, k-core, k-truss), several CPU-based sequential solutions has been proposed [2], [3], [4], [5], [6], [7]. The sequential online solutions suffer from the same weakness of high computation complexity, which is caused by expensive computation steps of ego-network extraction and decomposition. Fortunately, most of the complicated computations steps in the above solutions are computationally independent. Parallelization is obviously a good choice to improve the efficiency of the online solutions. A CPU-based parallel solution is proposed to solve the parameter-free structural diversity search problem over the core-based structural diversity model. However, the improvement in efficiency is still limited due to the coarse parallel architecture and lack of computational resources of CPU. This motivates us to turn our attention to investigating the GPU-based solution because of the rich computational resources and the massive parallel thread architecture of GPU that enables us to manage the computing resources in different levels to solve different tasks flexibly.

Recently, GPU has been shown to be effective in improving the efficiency of many graph analytic tasks such as triangle listing [8], [9], constraint shortest path [10], subgraph matching [11], [12], etc. GPU-based subgraph matching methods [11], [12] focus on general subgraph pattern counting solutions, which lack specific optimization for mining particular cohesive subgraph structures like k-core and k-truss. GPU-based k-core [13], [14] and k-truss [15] decomposition can be beneficial to some sub-steps of our structural diversity search problems. However, they need extra effort to construct the required auxiliary data structure like COO+CSR. Moreover, all of the above research focuses on the computation of the entire graph. None of them can be directly applied to our ego-network extraction and structural diversity search problems.

Compared with the architecture of CPUs, GPU follows the single instruction multiple threads (SIMT) execution model and coalesced memory access pattern, which encounters several challenges when designing efficient parallel graph algorithms. Because of the special architecture of GPU, accelerating the top-t structural diversity search on it is still challenging. In the first place, it's difficult to store the intermediate data (ego-network , maps) on GPU. Because in GPU, traditional STL containers in C++ are not supported. In the second place, synchronization and branch divergence caused by data-dependent operations and conditional checking (filtering qualified vertices and edges in the decomposition phase) are expensive in GPU because of the SIMT architecture. Finally, serious workload imbalance problems may be incurred because of different workloads for computing the structural diversity of vertices with diverse ego-network structures.

To address the above challenges, we propose a unified GPU-

Jinbin Huang, Xin Huang, Jianliang Xu, and Byron Choi are with the Department of Computer Science, Hong Kong Baptist University, Hong Kong, China. E-mail: jbhuang@comp.hkbu.edu.hk, xinhuang@comp.hkbu.edu.hk, bchoi@comp.hkbu.edu.hk, xujl@comp.hkbu.edu.hk
Yun Peng is with the Department of Artificial Intelligence, Guangzhou University, Guangdong,China. E-mail: yunpeng@gzhu.edu.cn

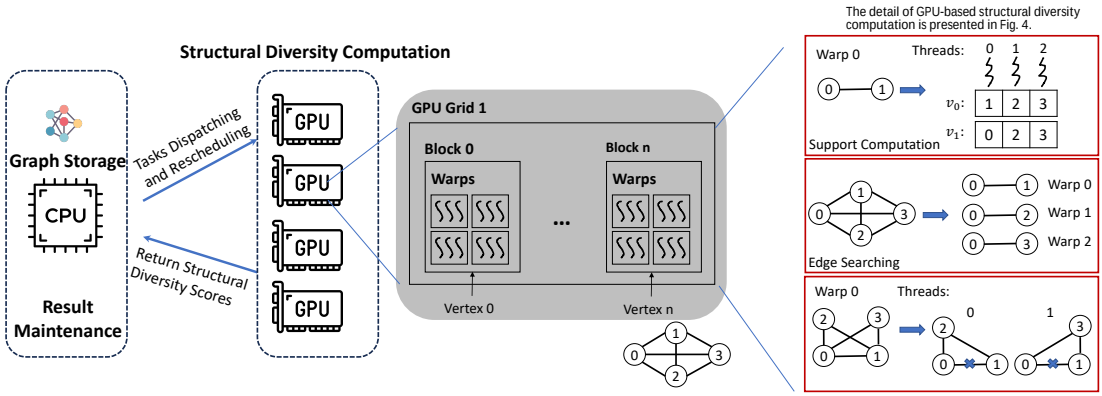


Fig. 1. A general framework for CPU+GPU based structural diversity search.

based solution for the structural diversity search problem based on three structural diversity models, which apply the idea of dynamically organizing the computing resource of GPU according to specific tasks. Figure 1 shows an example of accelerating the truss-based structural diversity search by GPU. A block in a GPU grid is used to process the ego-network of a particular vertex. In the truss decomposition phase, different levels of parallelism are used to deal with tasks with different complexity. For example, in support computation, a warp is used to deal with the support counting of an edge. Moreover, a thread in the warp is used to handle a vertex in the adjacency list of an end vertex of the edge to find out common neighbors in the adjacency list of another end vertex.

Specifically, we first propose a GPU-friendly lock-free algorithm to extract the ego-network of a vertex in CSR format, which is efficient for graph data storing and retrieving on GPU. We also propose efficient GPU-based solutions equipped with task-dependent parallelism to structural diversity computation for three structural diversity models respectively. Specifically, for the component-based structural diversity computation, we utilize the GPU parallel library GUNROCK to efficiently search qualified social contexts inside an ego-network, and design an effective algorithm to compute the final diversity score. For the core-based structural diversity computation, we propose an H-index-based GPU-friendly scheme to utilize different levels of GPU parallelism to avoid massive synchronizations. For the most challenging truss-based structural diversity computation, we first propose to use the GPU-based binary search to compute the support of all edges inside an ego-network. Secondly, we propose a GPU thread-level parallelism to search qualified edges to peel in an efficient way. Finally, we design a multi-level parallel hierarchy to identify the affected triangles efficiently and peel the edges safely. To further improve the performance, we extend our GPU-based solutions to multiple GPU environments, which achieves significant speed up in efficiency. In order to optimize the workload balance, we propose an effective global work-stealing technique to handle data skewness in complex networks.

To summarize, we make the following contributions:

- We formulate the structural diversity search problem in GPU environment to improve the efficiency of the existing solutions. (Section 2)
- We propose a unified GPU-based solution to tackle three typical structural diversity search models comp-div, core-div, and truss-div based on two commonly key steps of ego-network extraction and decomposition. (Section 3)
- We first propose a GPU-friendly and lock-free algorithm to extract the ego-network of a vertex in CSR format, which can achieve efficient processing speed. (Section 4)
- We propose comprehensive and efficient GPU-based structural diversity search solutions over three structural diversity models. Specifically, we design dynamic parallel architectures for the comp-div and core-div to utilize the computing resources of GPU. For the truss-div, we divide the entire process into different steps and propose several useful parallel strategies to leverage the strong GPU computation resource to avoid complex atomic operations. (Section 5)
- We extend the GPU-based solutions to multiple GPU scenarios to further improve the efficiency. We identify the cause of workload imbalance and propose an effective global work-stealing strategy to optimize the workload balance. (Section 6)
- We conduct extensive experiments to verify the efficiency and effectiveness of our proposed techniques. (Section 7)

We discuss related work in Section 8 and conclude the paper in Section 9.

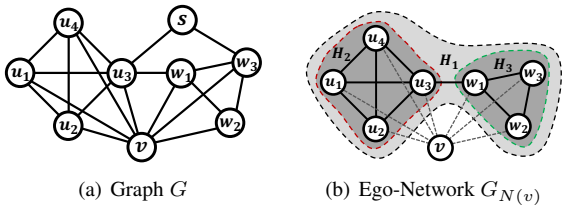
2 Preliminaries

We consider an undirected and unweighted simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. For a vertex $v \in V$, we define $N(v) = \{u \in V : (v, u) \in E\}$ as the set of v's neighbors and $d(v) = |N(v)|$ as the degree of v in G. Let d_{max} represent the maximum degree in G. For a set of vertices $S \subseteq V$, the induced subgraph of G by S is denoted by G_S, where the vertex set is $V(G_S) = S$ and the edge set is $E(G_S) = \{(v, u) \in E : v, u \in S\}$. Without loss of generality, we assume that the considered graph G is connected, indicating that $m \geq n - 1$ and $n \in O(m)$, which is similarly made in [16], [7].

2.1 Structural Diversity Models

For a given vertex v in G, we consider the structural diversity of v in the subgraph of v's 1-hop neighborhood, which is represented as an ego-network [17], [18] as follows.

Definition 1 (Ego-Network). Given a vertex $v \in V$, the ego-network of v is the subgraph of G induced by the vertex set $N(v)$, denoted by $G_{N(v)}$, where the vertex set $V(G_{N(v)}) =$



(a) Graph G (b) Ego-Network $G_{N(v)}$

Fig. 2. A running example

$N(v)$ and the edge set $E(G_{N(v)}) = \{(u, w) \in E : u, w \in N(v)\}$.

Throughout this paper, we consistently use G_v to represent the ego-network $G_{N(v)}$ for short when the context is obvious. Figure 2 (a) shows a graph G. The ego-network of vertex v is the subgraph H_1 highlighted by light gray color in Figure 2 (b). The graph G_v is induced by vertices $\{u_1, u_2, u_3, u_4, w_1, w_2, w_3\}$. The vertices s and v itself are discarded from G, excluding from G_v.

Based on the ego-network structure, we identify a social context as a densely-connected subgraph in G_v. Given an integer $k \in \mathcal{Z}^+$, we use three cohesive subgraph models of k-sized component, k-core, and k-truss to depict the structural contexts, respectively. Moreover, the structural diversity is defined as the number of social contexts in the ego-network G_v. In the following, we show three representative structural diversity models.

- comp-div model: The component-based structural diversity model treats each connected component that has the number of vertices greater than k as a distinct social context [2].
- core-div model: The core-based structural diversity model treats one maximal connected k-core as a distinct social context, in which a k-core requires that each vertex has at least k neighbors [7].
- truss-div model: The truss-based structural diversity model treats each maximal connected k-truss as a distinct social context, in which each edge of k-truss is contained by at least $k-2$ triangles [6].

In summary, we consider three structural diversity models denoted by $\mathcal{M} = \{\text{comp-div}, \text{core-div}, \text{truss-div}\}$.

Definition 2 (Structural Diversity). Given a vertex v, a structural diversity model $\mathcal{M} = \{\text{comp-div}, \text{core-div}, \text{truss-div}\}$, and an integer $k \in \mathcal{Z}^+$, the structural diversity of v is the number of social contexts $\mathsf{SC}(v)$ in ego-network G_v, denoted by $sc(v) = |\mathsf{SC}(v)|$.

Example 1. Given an ego-network of v in G in Figure 2 (b) and the parameter $k = 3$. Assume that the structural diversity model $\mathcal{M}$ uses the comp-div to treat each k-sized component as a social context. Thus, the subgraph H_1 is regarded as a social context. The structural diversity of v is 1. On the other hand, we use the core-div model and treat each maximal k-core as a social context. Thus, the subgraph H_2 will be regarded as a distinct social context since H_2 is a maximal connected 3-core. Hence, the structural diversity of v is also 1. However, when $\mathcal{M}$ is set to the truss-div model, indicating the use of maximal connected k-truss subgraphs as social contexts, H_2 and H_3 are treated as two social contexts since H_2 is a maximal connected 4-truss and H_3 is a maximal connected 3-truss. Therefore, the structural diversity of v is $sc(v) = |\mathsf{SC}(v)| = |\{H_2, H_3\}| = 2$ under the setting of truss-div model.

2.2 GPU Architecture and Problem Formulation

In the existing study, most studies of structural diversity search have been focused on in-memory computation using CPUs. Different previous studies, we intend to study generalized novel problems of GPU-based structural diversity search under alternative structural diversity model in $\mathcal{M}$.

GPU computation architecture. We first introduce the GPU computation architecture. As shown in the gray region in Figure 1, a grid is mapped to a GPU unit. Inside a GPU grid, threads are organized in computing blocks. A thread block is mapped to a multiprocessor. Consecutive threads (e.g., every 32 consecutive threads) are grouped into a warp inside a thread block. The threads inside a warp will be executed simultaneously and follow the Single Instruction Muti-Thread (SIMT) manner. The global memory of GPU can be accessed by all threads in a coalesced memory access pattern. Each thread block contains a fixed-size shared-memory (usually 64KB), which can only be accessed by the threads in the same block.

Next, we formulate the problem of GPU-based structural diversity search studied in this paper as follows.

Problem statement: Given a graph $G = (V, E)$, an integer $t \in \mathcal{Z}^+$, a structural diversity model $\mathcal{M} = \{\text{comp-div}, \text{core-div}, \text{truss-div}\}$, and a model threshold $k \in \mathcal{Z}^+$, the goal of GPU-based structural diversity search problem is to compute the top-t vertices S^* with highest structural diversity score $sc(v)$ and retrieve their social contexts $\mathsf{SC}(v)$ on one GPU-based machine, which is equipped with a single CPU and multiple GPUs.

The objective of GPU-based structural diversity search is to utilize multiple GPUs to accelerate the computation process of finding a t-sized answer $S^* \subseteq V$ where $|S^*| = t$ and $\forall v \in S^*$ with $sc(v) \geq sc(u)$ for all $u \in V \setminus S^*$.

3 Framework

In this section, we propose a unified CPU+GUP framework to utilize GPU computation sources for fast structural diversity search under three different structural diversity models.

Overview. In our framework, the input graph data is firstly stored in the CPU side. The structural diversity computation tasks for different vertices are then grouped and dispatched to the GPU side. The GPU side is only responsible for structural diversity computation. During the computation, the CPU will dynamically monitor and re-schedule the tasks among GPUs. After computation, the intermediate results will be returned to the CPU side. A top-t result list will be maintained and dynamically updated in the CPU side. A general framework is shown in Figure 1. To summarize, our framework is consisted of four phases as follows.

Phase 1: GPU-based upper bound computation. This phase first makes use of GPUs to parallel compute upper bounds for all vertices for efficiently pruning the search space. An upper bound denoted by $\overline{sc}(v)$ for each vertex v is proposed by the previous CPU-based solution based on the structural properties of each structural diversity model. Assume that it applies the structural diversity model of comp-div, $\overline{sc}(v) = \frac{|N(v)|}{k}$. Similar upper bounds can be extended to core-div and truss-div models.

Phase 2: GPU-based structural diversity computation. In this phase, we propose detailed GPU-based ego-network decomposition solution to extract ego-networks and then compute exact structural diversity scores, according to the

Algorithm 1 GPU-Accelerated Top-t Search Framework

Input: $G = (V, E)$, an integer t, a structural diversity model $\mathcal{M}$ and the model threshold k
Output: Top-t structural diversity results

1: *// Phase 1: GPU-based upper bound computation.*
2: **thread-level parallel for** $v \in V$ **do**
3: Compute the upper bound $\overline{sc}(v)$;
4: $\mathcal{L} \leftarrow$ sort all vertices V in descending order of $\overline{sc}(v)$;
5: $\mathcal{S} \leftarrow \emptyset$;
6: *// Phase 2: GPU-based structural diversity computation*
7: **while** $\mathcal{L} \neq \emptyset$
8: $v^* \leftarrow \arg\max_{v \in \mathcal{L}} \overline{sc}(v)$; Delete v^* from $\mathcal{L}$;
9: **if** $|\mathcal{S}| = t$ and $\overline{sc}(v^*) \leq \min_{v \in \mathcal{S}} sc(v)$ **then**
10: **break**;
11: Group consecutive vertices in $\mathcal{L}$ and dispatch them to GPUs for their structural diversity computation.
12: Apply Algorithm 2 on vertices of $\mathcal{L}$ for ego-network extraction.
13: Invoke GPU-based solutions for computing sc(v) using different models of comp-div in Algorithm 3, core-div in Algorithm 4, and truss-div in Algorithm 5, respectively.
14: *// Phase 3: Workload balance optimization on CPU.*
15: Monitor and reschedule the tasks dispatched to each GPU dynamically.
16: *// Phase 4: Top-k result maintenance on CPU.*
17: **for** each result $sc(v^*)$ returned from GPUs **do**
18: **if** $|\mathcal{S}| < t$ **then** $\mathcal{S} \leftarrow \mathcal{S} \cup \{v^*\}$;
19: **else if** $sc(v^*) > \min_{v \in \mathcal{S}} sc(v)$ **then**
20: $u \leftarrow \arg\min_{v \in \mathcal{S}} sc(v)$;
21: $\mathcal{S} \leftarrow (\mathcal{S}/\{u\}) \cup \{v^*\}$;
22: **return** $\mathcal{S}$ and their social contexts $\mathsf{SC}(v)$ for $v \in \mathcal{S}$;

particularly adopted structural diversity model $\mathcal{M}$ in Sections 4 and 5.

Phase 3: Workload balance optimization on CPU. To optimize the workload imbalance issue for multiple GPUs, the CPU side dynamically monitors and reschedules the computation tasks dispatched to each GPU. The objective is to balance the workload by dynamic work stealing strategy detailed presented in Section 6.

Phase 4: Top-t result maintenance on CPU. Finally, it updates the structural diversity computation results returned by GPU side to the top-t list $\mathcal{S}$ stored in the CPU side.

Our GPU-based framework. Algorithm 1 shows the detailed framework for GPU-based structural diversity search. Algorithm 1 computes the upper bound $\overline{sc}(v)$ for each vertex v in G using GPU thread-level parallelization (lines 2-3). The vertices are sorted in descending order according to their upper bounds (line 4). Then, it first checks the early stop condition by comparing the smallest upper bound in $\mathcal{L}$ with the smallest structural diversity score in the top-k result list $\mathcal{S}$ (lines 8-10). Next, it iteratively groups the consecutive candidate vertices in $\mathcal{L}$, and offloads their structural diversity computation to GPU side (line 11). In the GPU side, we first propose a lock-free GPU-based ego-network extraction method to extract the ego-network of the vertices (line 12) and apply structural diversity score computation algorithms (line 13). After that, it dynamically monitors and reschedules the computation tasks dispatched to each GPU (line 14). Finally, it updates the structural diversity computation results returned by GPU side to the top-t list $\mathcal{S}$ stored in the CPU side, until all candidates in $\mathcal{L}$ are computed or the early stop condition is triggered (lines 17-21).

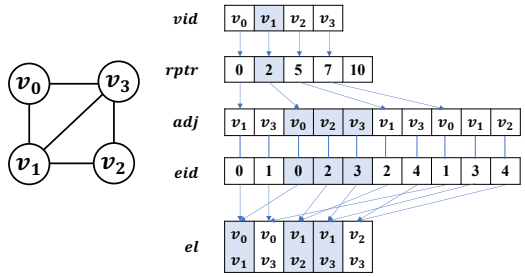


Fig. 3. An example of the CSR format.

4 Lock-free Ego-network Extraction

In this section, we present an GPU-based ego-network extraction method for multiple vertices in parallel. We first introduce an CSR format for graph storage in GPU. To make fully use of multiple threads in GPUs, we develop a lock-free ego-network extraction, including two key techniques: 1) GPU-based ego-edge retrieval, and 2) GPU-friendly Scatter-Aggregate based ego-network extraction.

Graph-based CSR format. We start with introducing the data structure of the graph storage. The in-memory storage of a graph often contains a vertex list, an edge list, and the adjacency lists of all vertices. To save space and allow efficient access, the Compressed Sparse Row (CSR) format [19] is widely adopted by existing works [20], [21], [19]. An example of the CSR format is shown in Figure 3. It consists of a compressed adjacency list adj and a list of row pointers $rptr$ that points to the starting position of corresponding adjacency list of a vertex. Edges are stored in value pairs recording the two end vertices IDs. An edge list el is generated to store all the edge information. To quickly retrieve the edge information by adjacency lists, an edge ID list eid is constructed to map the edges to the adjacency lists. For example, if we need to retrieve the neighborhood of vertex v_1, the start and end positions of its adjacency list is stored in second and the third position of the row pointer $rptr$, which is 2 and 5. Hence we can find v_1's adjacency list in adj from index 2 to index 5, which is v_0, v_2, v_3. In this manner, the graph information can be easily transferred and operated between CPU and GPU.

To compute one vertex v's structural diversity score $sc(v)$, a fundamental and necessary step is to extract the ego-network G_v. Next, we present a lock-free GPU-based ego-network extraction solution to construct CSR-based graph storages in parallel.

GPU-based ego-edge retrieval. Before constructing the CSR for each ego-network, the first step is to retrieve the edges inside each ego-network. This step is done by a global triangle listing in a previous work [6]. However, the CPU-based solution is costly in efficiency. To accelerate this process, we propose a multiple GPUs based parallel solution. The core idea is to partition the edge-centric computational task into each GPU. On the CPU side, the adjacency lists of the two end vertices of each edge are retrieved and compacted firstly. The edge data will be divided into equal size and transfer to each GPU. Inside each GPU, it utilizes block-wise parallelism to compute the triangles associated with each edge. After identifying the triangles, we are able to figure out

Algorithm 2 GPU-based Ego-network Extraction

Input: A vertex v, an edge id list E_{G_v} the CSR of global graph G
Output: The CSR of the ego-network of v G_v

1: Let $m_v \leftarrow |E_{G_v}|$, $n_v \leftarrow |N(v)|$;
2: Initialize the CSR of G_v by $adj_v[m_v * 2]$, $deg[n_v]$, $rptr_v[n_v + 1]$, $eid_v[m_v * 2]$, $el_v[m_v]$;
3: Let $MapSize$ be the largest vertex id in $N(v)$;
4: Initialize a $MapSize$-sized array map, i.e., $map[MapSize]$;
5: **Thread-level parallel for** each u in $N(v)$ **do**
6: $map[u] \leftarrow$ the index of u in $N(v)$;
7: **Thread-level parallel for** each $eid \in E_{G_v}$ **do**
8: $e = (x, y) \leftarrow el[eid]$;
9: $el_v[i] \leftarrow (map[x], map[y])$, where i is the index of eid in E_{G_v};
10: $AtomicAdd(deg[map[x]], 1)$;
11: $AtomicAdd(deg[map[y]], 1)$;
12: $rptr_v \leftarrow ParallelPrefixSum(deg)$;
13: **for** each edge $e = (u, w) \in el_v$ **do**
14: $adj_v[rptr_v[u + 1] - deg[u]] \leftarrow w$;
15: $eid_v[rptr_v[u + 1] - deg[u]] \leftarrow$ the index of e in el_v;
16: $deg[u] \leftarrow deg[u] - 1$;
17: $adj_v[rptr_v[w + 1] - deg[w]] \leftarrow u$;
18: $eid_v[rptr_v[w + 1] - deg[w]] \leftarrow$ the index of e in el_v;
19: $deg[w] \leftarrow deg[w] - 1$;
20: **Return** the ego-network CSR of G_v;

which ego-networks the edge is belong to. The temporary result will be stored in shared memory, and later transferred to global memory of the GPU in a lazy manner. After processing all edges, the results will be returned back to the CPU. On the CPU side, it will iteratively copy the edges to each ego-network.

Naive ego-network extraction. After retrieving the edges in each ego network, our next task is to extract the ego-network structures for all vertices in CSR format. Existing sequential solution iteratively scans the edge list and adds the neighbors to the corresponding adjacency lists one by one. Then the adjacency lists will be concatenated to form adj. And the prefix sum algorithm will be conduct on the degree of each vertex to obtain the row pointer array $rptr$. A great amount of temporary array will be produced in this solution, which will waste the limited GPU memory resources.

Lock-free Ego-Network Extraction. To address these issues, we present a lock-free ego-network extraction algorithm in parallel to improve efficiency, which is outlined in Algorithm 2. In Algorithm 2, it first initializes a map array map to rehash the vertex id in an ego-network , since the vertex id inside an ego-network may be sparse for storage and computing (lines 3-4). The size of the map is set to the largest vertex id in the neighbor list. This can be done in $O(1)$ because the neighbor list is well-sorted in the global CSR. Secondly, a thread-level parallel scheme is applied to each vertex in $N(v)$ to set the mapping from the original vertex id to the new vertex id starting from 0 to $|N(v)| - 1$ for each vertex v (lines 5-6). Next, an individual thread is used to process an edge in parallel (lines 7-11). It first obtains the original edge from the global CSR (line 8). Then it creates a new edge according to the new vertex id of the two end points of this edge, and inserts the new edge to the edge list of v (line 9). For the two end vertices of the new edge, increase the degree of each vertex using AtomicAdd operation (lines 10-11). Then a thread-level parallel prefix sum algorithm provided from CUDA can be applied based on the degree of each vertex to obtain the row pointer array $rptr_v$ for each v (line 12). Finally, to ensure the well-sorted feature of the adjacency list, a single thread is used to iteratively process each new edge in the edge list of v (lines 13-19). It will insert the vertex id and the corresponding edge id to the correct position of the adjacency list adj_v and edge id list eid_v. This solution is a lock-free solution that utilize the thread-level parallelism to improve the efficiency and ensure the correctness at the same time.

Algorithm 3 GPU-accelerated Comp-Div Computation

Input: A vertex v, the CSR of the G_v and a model threshold k
Output: The structural diversity score $sc(v)$

1: $sc(v) \leftarrow 0$, $\mathcal{SC} \leftarrow \emptyset$;
2: **for** each u in $N(v)$ **do**
3: **if** $visited(u) \neq FALSE$ **do**
4: $\mathcal{SC} \leftarrow$ GUNROCKBFS(u);
5: **if** $|SC| \geq k$ **do**
6: $sc(v) \leftarrow sc(v) + 1$;
7: **thread-level parallel for** each x in $\mathcal{SC}$ **do**
8: $visited(x) \leftarrow TRUE$;
9: **Return** the $sc(v)$ of G_v;

5 GPU-Accelerated Structural Diversity Computation

In this section, we present three GPU-based algorithms of structural diversity score computations, in terms of three considered structural diversity models $\mathcal{M} = \{$comp-div, core-div, truss-div$\}$, respectively. Specifically, we first propose a GPU-based parallel BFS scheme to count the connected component in each ego-network for comp-div. Secondly, we implement a H-index based core decomposition in GPU environment to compute the core-based structural diversity. Finally, for the truss-based structural diversity computation, we dynamically group different levels of GPU computing resources to propose fine-grained parallel solutions according to different sub-tasks.

5.1 Component-based Structural Diversity Search

As the component-based structural diversity model treats each connected component inside an ego-network as a distinct social context, the key step in computing component-based structural diversity is quantifying the connected components with size larger than k. Breadth-first search (BFS) is a natural approach for identifying connected components. However, performing BFS concurrently from multiple vertices encounters massive synchronizations and thread communications, which is highly expensive in GPU computing. To address this, we propose a combinational scheme to reduce the synchronizations and communications and well utilize the parallel features of GPU.

Comp-Div computation algorithm. Algorithm 3 outlines the details of the GPU-accelerated component-based structural diversity computation for (v). In each iteration, it retrieves a neighbor $u \in N(v)$ to perform BFS (line 1). If this u has not been visited, it applies the GPU-based BFS using GUNROCKBFS [22] to identify the connected component starting by u (lines 3-4).

Algorithm 4 GPU-accelerated Core-Div Computation

Input: A vertex v, the CSR of the G_v and a model threshold k
Output: The structural diversity score $sc(v)$

1: $sc(v) \leftarrow 0$, $\mathcal{SC} \leftarrow \emptyset$;
2: **Thread-Parallel for** each u in $N(v)$ **do**
3: $c(u) \leftarrow |N_{G_v}(u)|$;
4: $\mathcal{F} \leftarrow TRUE$;
5: **while** $\mathcal{F}$ **do**
6: $\mathcal{F} \leftarrow FALSE$;
7: **Warp-Parallel for** each u in $N(v)$ **do**
8: Apply thread-level parallelism to compute $H(u)$;
9: **if** $c(u)! = H(u)$
10: $c(u) \leftarrow H(u)$
11: $\mathcal{F} \leftarrow TRUE$;
12: Apply parallel BFS to compute $sc(v)$ according to the core decomposition result;
13: **Return** the $sc(v)$ of G_v;

After obtaining the connected component, it compares the size of the component and updates the structural diversity score $sc(v)$ (lines 5-6). Then it applies the GPU thread-level parallelism to update the visit array (lines 7-8). When all neighbors are visited, it returns the final results (line 9).

5.2 Core-based Structural Diversity Search

When it comes to the core-based structural diversity search, the key step for computing core-based structural diversity computation is the core decomposition of the ego-network . Traditional peeling method iteratively deletes the vertices with lowest degrees to perform the core decomposition. It consists of deep data dependency as one vertex removal leading to other neighbor degree decrement, which requires massive synchronizations when updating the degrees. This is obviously unsuitable for parallel environment like GPU. On the contrary, a H-index based core decomposition scheme proposed by [23] treats the coreness of a vertex as the H-index of the coreness among its neighbors, i.e., $H(v) = \max_{k \in \mathcal{Z}} |u \in N(v) : H(u) \geq k| \geq k$. This scheme requires more computational steps, but it's suitable for parallel environment. In our solution, we propose a novel GPU-based solution that utilizes different level of parallelism to deal with different type of tasks based on the H-index based approach.

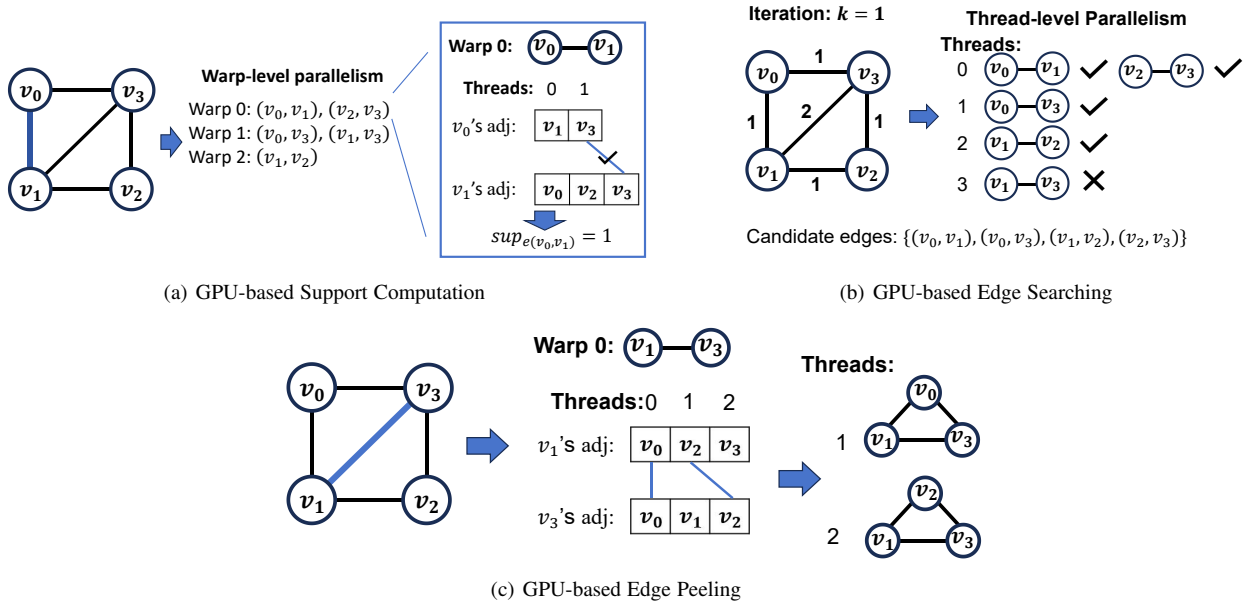
H-index based Core-Div computation algorithm. Algorithm 4 illustrates the details of the GPU-accelerated core-based structural diversity computation. We deonte $H(v)$ as the H-index of coreness of a vertex v. It first applies a thread-level parallelism to initialize the estimated coreness of each vertex u in $N(v)$ by u's degree $|N_v(u)|$ (lines 2-3). Secondly, in each iteration, it uses the H-index $H(u)$ to estimates the coreness of each vertex u. Until all the estimated coreness remain unchanged, it safely terminates the decomposition (lines 5-11). To better utilize the computing resources of GPU and achieve fine-grained parallelism, it applies a thread-level parallelism to compute the H-index $H(u)$ for each vertex u. A flag $\mathcal{F}$ is initialized to be TRUE for identifying the changing state of H-indexes (line 4). $\mathcal{F}$ is set to FALSE at the begining of each iteration (line 6). Once the H-index of an vertex changes, it alter the flag $\mathcal{F}$ back to TRUE to indicate the execution of the next round (line 11). When all of the H-indexes stay unchanged, the iteration can be terminated safely. Finally, it computes the structural diversity $sc(v)$ of v by a coreness-aware breath-first-search.

5.3 Truss-based Structural Diversity Search

Dislike the component-based and core-based structural diversity model, the computation of the truss-based structural diversity contains more computation steps caused by the truss decomposition. The truss decomposition consists of three important steps of support computation, edge searching and edge peeling, which encounters synchronization cost and complex computation steps. To address this, we propose a novel GPU-based solution using the high-level idea of a CPU-based parallel work PKT [24]. Our core idea is to dynamically utilize different levels of parallelism to handle tasks with different complexity.

Overview. Figure 4 gives an overview of our GPU-based solution for truss-based structural diversity computation. As shown in Figure 4 (a), we apply a wrap-level parallelism to process each edge inside an ego-network , and a thread-level parallelism to process the common neighbor look-up in an adjacency list. For example, the support computation of (v_0, v_1) is assigned to Warp 0. Inside Warp 0, each thread corresponds to a neighbor in v_0's adjacency list. In each thread, a binary search approach is used to find common neighbor matching in v_1's adjacency list. Finally, the support of (v_0, v_1) is 1 since they have only 1 common neighbor v_3. Figure 4 (b) and (c) further show the examples for our GPU-based edge searching and edge peeling steps in the truss decomposition process. For edge searching, each edge checking is assigned to a thread, and is checked if it is a qualified one for peeling. In this example, when the current support value $k = 1$, (v_1, v_3) is unqualified. Hence, candidate edges are $\{(v_0, v_1), (v_0, v_3), (v_1, v_2), (v_2, v_3)\}$. For edge peeling process, it uses a thread-level parallelism to perform set intersection to find out the triangle related to a given edge in a warp. In this example, the edge peeling process of (v_1, v_3) is assigned to Warp 0. Inside Warp 0, each thread corresponds to one of the neighbors of v_1. After binary look-up in v_3's adjacency list, it can identify two triangles $\triangle_{v_0 v_1 v_3}$ and $\triangle_{v_1 v_2 v_3}$ that relate to edge (v_1, v_3).

Truss-Div computation algorithm. The details of the GPU-accelerated truss-based structural diversity computation is shown in Algorithm 5. It first utilizes a warp-level parallelism to deal with the support computation of each edge e in ego-network $G_{N(v)}$ (line 3). In the support computation of each edge, it applies the GPU binary search based set intersection technique proposed in [8] to look up the common neighbors in two adjacency lists in a coalesced memory access manner (line 4). The decomposition process starts from support value 0 (line 8). For efficient edge retrieval, the boolean lists $delete, inCurr$ and $inNext$ are used to record the status of the edges of being deleted, being processed in the current iteration or to be processed in the next iteration respectively (lines 5 and 7). For each support value, it uses thread-level parallel Algorithm 6 to search the qualified edges for peeling (line 11). Next, it continues to peel the edge in a warp-level parallel manner using Algorithm 7 (line 14). The peeling algorithm is optimized by utilizing different thread level parallel based on the particular task. After peeling the qualified edges, the affected edges in the $next$ array will be also peeled in the same iteration (lines 15-16). After all the affected edges in the same level is peeled, it continues to process the edges with the next support value (line 17). After all the edges are peeled, the algorithm finishes and returns the trussness of each edge (line 9). Based



(a) GPU-based Support Computation

(b) GPU-based Edge Searching

(c) GPU-based Edge Peeling

Fig. 4. Details of GPU-based Truss Decomposition. Assume the GPU setting contains 3 warps in each block, and each warp contains 4 threads. Fig. 4 (a) shows the detail of support computation, where each edge support computation is assigned to a warp. Fig. 4 (b) shows the qualified edge searching, where each edge computation is assigned to a thread. Fig.4 (c) is shows the detail of edge peeling step under hierarchical parallelism levels.

Algorithm 5 GPU-accelerated Truss-Div Computation

Input: The CSR of a vertex v
Output: The trussness array $truss_v$ for all edges in G_v

1: *// Warp-level Parallel Support Computation*
2: Initialize $sup_v[m_{G_v}]$ with zero values;
3: **Warp-level parallel for** each edge $e = (x, y) \in E_{G_v}$ **do**
4: $sup_v[e] \leftarrow$ Apply the binary set intersection method in [8] to compute the edge support;
5: Initialize $truss_v[m_{G_v}]$; $delete \leftarrow \{false\}$;
6: $removeCount \leftarrow 0$;$curr \leftarrow \emptyset$; $inNext \leftarrow \emptyset$
7: $inCurr \leftarrow \{false\}$; $inNext \leftarrow \{false\}$;
8: $k \leftarrow 0$
9: **while** $removeCount < |el_v|$ **do**
10: *// Thread-level Parallel Edge Searching*
11: Apply Algorithm 6 to search the qualified edges;
12: **while** $|curr| > 0$ **do**
13: *// Warp-level Parallel Edge Peeling*
14: Apply Algorithm 7 to peel the qualified edges and update the information in the following arrays: $sup_v, inCurr, removeCount, next, inNext$;
15: $curr \leftarrow next$; $next \leftarrow \emptyset$;
16: $inCurr \leftarrow inNext$; $inNext \leftarrow \{false\}$;
17: $k \leftarrow k + 1$;
18: *// Score Computing*
19: Apply parallel BFS to compute $sc(v)$ according to the truss decomposition result;
20: **Return** the $sc(v)$ of G_v;

Algorithm 6 Parallel Edge Searching Procedure

Input: The CSR of a vertex v, target support k, support array sup_v, output edge list $curr$, edge flag array $inCurr$
Output: The output array $curr$, flag array $inCurr$

1: Initialize a local buffer array buf of size s for each thread;
2: $j \leftarrow 0$;
3: **Thread-level Parallel for** each $e \in el_v$ with index i **do**
4: **if** $sup_v[i]$=k **then**
5: $buf[j] \leftarrow i$; $j \leftarrow j + 1$;
6: $inCurr[i] \leftarrow true$;
7: **if** $j = s$ **then**
8: Atomically copy buf to $curr$;
9: Atomically update the end pointer of $curr$;
10: $buf \leftarrow \emptyset$; $j \leftarrow 0$;
11: **Thread-level Parallel**
12: **if** $j > 0$ **then**
13: Atomically copy buf to $curr$;
14: Atomically update the end pointer of $curr$;

on the trussness of each edge, we are then able to compute the truss-based structural diversity score using the BFS mechanism similar to the core-based solution (lines 19-20).

GPU-based edge searching. To be specific, we presents our GPU-based edge searching algorithm in Algorithm 6. Its goal is to filter all the qualified edges with the support equals to the current k value. It assigns a thread for the checking of each edge inside an ego-network (line 3). Inside each thread, it maintains a local buffer list of size s and an index j (lines 1-2). In each thread, when it finds a qualified edge, it adds the edge to the end of its local buffer and updates the end pointer j. It also set the status of this edge in $inCurr$ to be $true$ to indicate that it will be process in the current iteration (lines 4-6). If the buffer of the current thread is full, the edge IDs in the buffer will be copy to the end of $curr$ list in an atomic way (lines 7-9). The buffer and its end pointer is then reset (line 10). Finally, it continues to deal with the qualified edges in the buffer list in the last iteration in each thread (lines 11-14).

GPU-based edge peeling. Next, we introduce the details of the GPU-based edge peeling algorithm in Algorithm 7. The goal of Algorithm 7 is to peel the qualified edges filtered by the above edge searching algorithm, assign the trussness to the edges to be

Algorithm 7 Parallel Edge Peeling Procedure

Input: The CSR of a vertex v, target support k, support array sup_v, peeling edge list $curr$, edge flag array $inCurr$, affected edges $next$, edge flag array $inNext$, removed edges count $removeCount$

Output: The trussness array $truss$, affected edges $next$, edge flag array $inNext$

1: **Warp-level Parallel for** each eid in $curr$ **do**
2: $e_1 = (x, y) \leftarrow el_v[eid]$;
3: $truss[eid] \leftarrow k + 2$;
4: AtomicAdd($removeCount$,1);
5: **Thread-level Parallel** for each $\triangle_{xyz}$ identified by GPU-BinarySetIntersection [8] **do**
6: $e_2 \leftarrow (x, z)$;$e_3 \leftarrow (y, z)$; Assume $x < y < z$;
7: Update the support of e_2 and e_3 if necessary;
8: Update the $next$ and $inNext$ if necessary;
9: **Thread-level Parallel** for each eid in $curr$ **do**
10: $delete[eid] \leftarrow false$;
11: $inCurr[eid] \leftarrow false$;

peeled, and identify the affected edges by the broken triangles. It uses a warp to process an edge in the qualified edge list $curr$ (line 1). It assigns the correct trussness to the edge and increases the counter of the removed edges by 1 (lines 3-4). Next, it utilizes each thread in the current warp to identify a triangle formed by the edge processed in the warp using the set intersection operation proposed in [8] (line 5). It updates the support of the affected edges if necessary (i.e., the affected edges are not deleted, not in the current round and has support greater than k) (line 7). Accordingly, it updates the affected edges to the $next$ list and the their status in the $inNext$ list to indicate that these edges should be further processed in the same round (i.e., edges that have support k after update) (line 8). Finally, for each edge in the $curr$ list, we apply thread-level parallelism to set its status in the $delete$ and $inCurr$ to be $false$ to indicate the end of processing of this edge (lines 9-11).

6 Multiple GPUs Solution and Workload Balance Optimization

The above sections of GPU-based ego-network extractions and structural diversity computations can be processed in one CPU and one single GPU with multiple cores and threads. To further accelerate the parallel efficiency, we explore multiple-GPUs scheme and propose two effective workload balance optimization techniques in this section: 1) work packing based on estimated workload, and 2) dynamic work stealing strategy during runtime.

6.1 Workload Imbalance Analysis and Our Optimization Framework

To maximize the acceleration, a natural way is to extend our solutions to multiple GPUs. However, the workload imbalance issue will be raised when assigning different workload to each GPU. Specifically, vertices with different ego-network structures may encounter different computation times. Randomly grouping the vertices with different computation costs into a GPU may cause severe workload imbalance among multiple GPUs.

Workload balance optimization. To address the above problem, we propose a workload balance optimization framework as shown in Figure 5. The framework contains three key steps. The first step is to estimate the workload for each vertex to be computed via a well-design cost estimation function. The second step is to apply a workload balance strategies to pack the workload of each vertex to achieve workload balance among multiple GPUs in theoretical level. In the last step, we propose a global work stealing strategies to re-balance the workload according to practical running status.

Workload estimation. We propose a cost model to quantitatively optimize the workload balance discussed above. Assume that we can derive some function $c(v)$ to estimate the computational cost of the ego-network of a vertex v. $c(v)$ is a function relevant to the degree of v and the ego-network size of G_v.

Given a vertex ordering $\mathcal{O}_S$ of a set of vertex S, in which the computation of consecutive vertices will be grouped into a GPU B_i for computing. Denote the total computation cost of a GPU B_i by C_i, we have $C_i = \sum_{v \in B_i} c(v)$. The total computation cost of all ego-networks is given by $T_{cost} = \sum_{v \in S} c(v)$.

Work packing. Assume that there is a total of $m_{GPU} \in \mathcal{Z}^+$ GPUs. If the workload are balanced among all GPUs, the ideal computation cost in each GPU should be an average on the total computation cost, i.e., $\frac{T_{cost}}{m_{GPU}}$. The optimization objective is to find a vertex reordering scheme $\mathcal{O}_S$ such that:

$$\min_{\mathcal{O}_S} \sum_{i=1} |C_i - \frac{T_{cost}}{m_{GPU}}| \quad (1)$$

In our optimization goal of $|C_i - \frac{T_{cost}}{m_{GPU}}|$, we want to minimize the difference between the actual computation cost in a GPU and the average computation cost to achieve workload balance. Next, we analyze that the hardness of finding the optimal vertex reordering scheme $\mathcal{O}_S$ such that equation 1 is minimized to be NP-hard similarly as the partition problem [9].

A near-linear time greedy algorithm for vetex ordering. Since the optimization problem is NP-hard, we propose a greedy algorithm to solve the problem in a heuristic manner. The detail of the greedy algorithm is presented in Algorithm 8. For each GPU B_i, it assigns a weight of zero and inserts it to a priority queue Q (lines 1-3). For each vertex $v \in V$, it pops out the GPU B_i with the smallest workload C_i from the priority queue Q (line 5). Next, it assigns the vertex v to B_i and update the workload C_i with v's computation cost $c(v)$ (lines 6-7). The updated GPU is then inserted back to the priority queue Q (line 8). Finally, it reorders the vertex IDs according to the vertex distribution to each GPU (line 9). Instead of performing actual reordering operation to the vertices, it just groups the vertices in each GPU assignment into an individual array.

Cost function design. To estimate the workload of the structural diversity computation of a vertex over different models, we derive the cost functions based on the computation complexity of each model. For the component-based structural diversity model, the computation time estimation function of a vertex v is given by $c(v) = \alpha|E_{G_{N(v)}}|$. For the core-based structural diversity model, $c(v)$ is given by $c(v) = \alpha|N_{G_{N(v)}}| + \beta|E_{G_{N(v)}}|$. And for the truss-based model, $c(v)$ is given by $c(v) = \alpha|N_{G_{N(v)}}| + \beta|E_{G_{N(v)}}|^{1.5}$. α and β are two weighting parameters. The design of the cost function for all models are based on their computation complexities. Noted that the complexity of truss decomposition for $G_{N(v)}$ is $|E_{G_{N(v)}}|^{1.5}$ [25]. Hence, the exponent for $|E_{G_{N(v)}}|$ is 1.5 in the cost function for the truss-based model. The detail method for obtaining the weighting parameters α and β is presented in Exp-1 of Section 7.

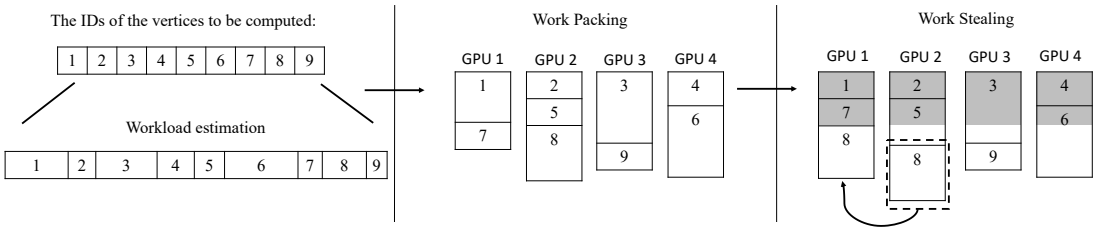
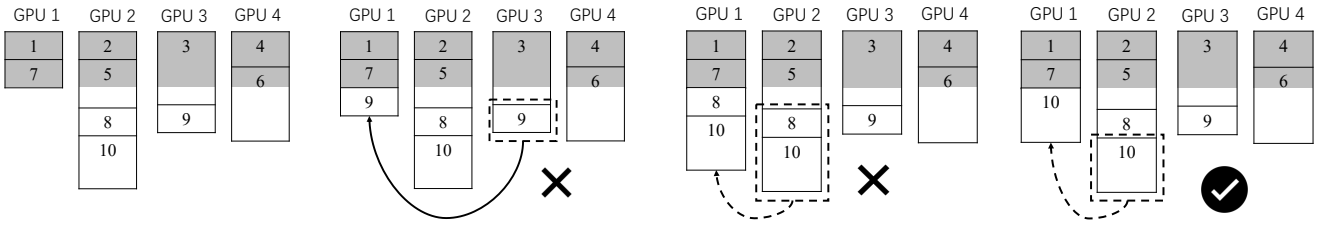


Fig. 5. The workload balance optimization framework.



(a) The workload status after GPU 1 finishes its work.

(b) A random work stealing strategy.

(c) An exhausted work stealing strategy.

(d) An optimized global work stealing strategy.

Fig. 6. A running example of the work stealing strategies.

Algorithm 8 Greedy Algorithm for Vertex Reordering

Input: A set of vertices S, computation cost estimation $c(v)$ for all v

Output: Vertex reordering $\mathcal{O}_S$, i.e., S'

```
Initialize a priority queue Q;
for each GPU B_i do
    Q.insert(B_i, 0);
for each v ∈ V do
    (B_i, C_i) ← Pop out GPU B_i with the smallest total cost C_i from Q;
    B_i ← B_i ∪ {v};
    C_i += c(v);
    Q.insert(B_i, C_i);
return O_S ← Reorder the id of vertex according to each B_i;
```

Algorithm 9 Global Workload Stealing Algorithm

Input: A max-heap Q for storing the work lists in all GPUs

Output: A new work list $\mathcal{W}$ for the current GPU.

```
W ← ∅; C ← 0;
(B_i, C_i) ← Pop the work list B_i associated with the largest computation cost C_i from Q;
while C < C_i do
    Retrieve v from the back of B_i;
    C ← C + c(v); W ← W ∪ {v};
    B_i ← B_i \ v; C_i ← C_i − c(v);
Return The new work list W;
```

Complexity analysis. Firstly, it conducts the ego-network extraction for all vertices to estimate the computation cost $c(v)$ for each vertex v, which runs in $O(m^{1.5})$ time [6]. In each iteration, it inserts back a new element to the priority queue, which runs in $O(\log m_{GPU})$ time. Hence, Algorithm 8 runs in $O(m^{1.5} + n \log m_{GPU})$ time. Since the GPU number is much less the vertex size, i.e., $|m_{GPU}| << n$, Algorithm 8 has near-linear time complexity.

6.2 Dynamic Global Work Stealing Strategy

Although the greedy work packing techniques above can theoretically alleviate the workload imbalance problem, the practical running circumstance usually differs since the work packing is based on estimation. Hence, the workload imbalance problem may still exist due to inaccurate estimation. Work-stealing strategies can be applied to optimize this workload imbalance issue. However, an ineffective work-stealing strategy may lead to limited workload balance improvement. Figure 6 illustrate a toy example. Consider the running state shown in Figure 6 (a), only GPU 1 has finished its workload at this moment. It can steal work from the other three GPU to balance the workload. GPU 2 has the heaviest workload based on estimation, which will exactly influence the total running time. We have two useful observations for work-stealing strategies as follows.

Observation 1: Non-optimized random work stealing. A random work-stealing strategy is shown in Figure 6 (b). GPU 1 will randomly steal one task associated with vertex 9 from GPU 3. However, this optimization will not contribute to the reduction of total running time, since GPU 2 still possesses the heaviest workload.

Observation 2: Unbalance exhausted work stealing. Inspired by this, the work-stealing strategy shown in Figure 6 (c) chooses to steal works from GPU 2 which has the heaviest workload. It exhaustedly steals all the unfinished tasks from GPU 2. Although this strategy can reduce the total running time, the contribution is limited since GPU 2 will be idle after it finishes the computation of vertex 5, and will suffer from frequent work stealing from others.

Figure 6 (d) shows a reasonable work-stealing strategy, it only steals one task from GPU 2, and the reduction of overall running time can be maximized. This motivates the design of our dynamic global working strategy.

To maximize the workload balance optimization, as shown in Figure 6 (d), we propose a global work-stealing strategy to dynamically manage the global workload at run time. Firstly, a status table with task lists is maintained in main memory by a CPU thread. Secondly, when a GPU finishes one of its tasks, it

TABLE 1
Network Statistics (K= 10^3 and M= 10^6)

Name	$\|V\|$	$\|E\|$	d_{max}	τ_G^*	τ_{ego}^*	$\mathcal{T}$
Wiki-Vote	7**K**	103**K**	1,065	23	22	608,389
Email-Enron	36**K**	183**K**	1,383	22	21	727,044
Epinions	75**K**	508**K**	3,044	33	32	1,624,481
Gowalla	196**K**	950**K**	14,730	29	28	2,273,138
NotreDame	325**K**	1.4**M**	10,721	155	154	8,910,005
LiveJournal	4**M**	34.7**M**	14,815	352	351	177,820,130
socfb-konect	59**M**	92.5**M**	4,960	7	6	6,378,280
Orkut	3.1**M**	117**M**	33,313	73	72	412,002,900

should update its working progress and its remaining computation cost on the status table. Thirdly, if a GPU finishes all its tasks, it clears its task list on the status table and checks the progress of other GPUs from the status table. Since the total running time may only depend on the GPU with the largest workload, it always steals tasks from the GPU with the largest unfinished workload. Moreover, an effective algorithm is designed to decide how many works can be stolen in Algorithm 9. In this manner, the workload imbalance can be alleviated significantly.

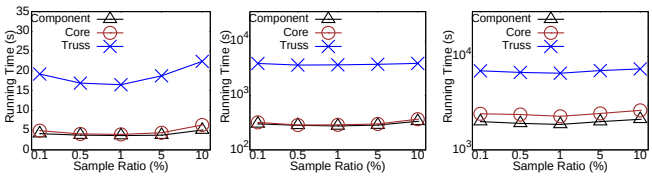
Dynamic work-stealing algorithm. Algorithm 9 shows the details of the dynamic global work-stealing strategy. After the work packing step in Algorithm 8, each GPU is assigned a work list. On the CPU side, the work lists of all GPUs are maintained in a max-heap Q according to the total cost of each work list. Algorithm 9 uses the max-heap Q as input. Each time when a GPU finishes its work, it pops a work list B_i with estimated cost C_i from the max-heap Q (line 2). In each iteration, it retrieves a task associated with a vertex v from the back of the work list B_i (line 4). Then, it pushes v to the result work list $\mathcal{W}$, and accumulates its estimated cost $c(v)$ to the current cost C (line 5). The information of v will be removed from B_i (line 6). The process will be terminated until the current cost C exceeds the remaining cost C_i in B_i (line 3). Finally, it returns the work list $\mathcal{W}$ as the result (line 7). The new work list $\mathcal{W}$ will be assigned to the current GPU for work stealing.

Discussion. Both the work packing and the work-stealing techniques are based on the estimation of the workload of each structural diversity computation task, which depends on the computation complexity of different operations. For the warp-wise workload balance optimization, the computation cost of the substeps inside each structural diversity computation task is unpredictable. Therefore, the workload balance optimization strategies above are not applicable to the warp-wise workload balancing. For the block-wise workload balance optimization, each block runs in an optimistic manner to automatically retrieve tasks from the work list when it finishes the current tasks, which causes only a little waiting time. The workload balance optimization techniques above is not necessary for this case.

7 Experiments

All algorithms are implemented by C++ and CUDA on a machine equipped with a 6-core 2.9GHz Intel CPU and four Tesla V100 GPUs. The main memory of the machine is 128GB, and each GPU has 32 GB global memory.

Datasets: We conduct our experiments on eight real-world social networks downloaded from SNAP. Table 1 shows the statistic of the datasets. $|V|$ and $|E|$ is the number of the vertices and edges of the dataset respectively. We use d_{max} to denote the maximum degree of vertex in a graph. We also report the maximum trussness



(a) Gowalla (b) LiveJournal (c) Orkut

Fig. 7. The running time against different sampling ratio for workload estimation under three structural diversity models on three graphs.

inside a graph and inside all ego-networks of the graph as τ_G^* and τ_{ego}^*. And the number of triangles inside a graph is denoted as $\mathcal{T}$. In our experiments, all networks can be stored in main memory. We fix the parameters $k = 4$ and $t = 100$ without loss of generality.

Comparing methods: We compare our proposed techniques with existing ones in term of efficiency. The existing baseline methods are listed as follows:

- BC-Div [13]: The baseline core-based structural diversity search method equipped with the SOTA GPU-based k-core decomposition solution.
- PGL-Div [15]: The baseline truss-based structural diversity search method equipped with the GPU-based k-truss decomposition solution.
- CPU-Div: The sequential algorithm for computing top-t structural diversity for all three models presented in [2], [7], [6].
- MCPU-Div: The CPU-based parallel version of the sequential algorithm [5]. Here we set the thread number to 6, since it performs the best on our machine.

We compare the above baselines with the following methods proposed in this paper:

- GPU-Div: The GPU-accelerated version on single GPU.
- MGPU-Div: The GPU-accelerated version on four GPUs.
- MGPU-Div-E: The GPU-accelerated version equipped with four GPUs and exhausted global work stealing workload balancing optimization.
- MGPU-Div-D: The GPU-accelerated version equipped with four GPUs with dynamic global work stealing workload balancing optimization as described in Algorithm 9.

Exp-1: Weighting parameter estimation for workload cost function. We obtain the two parameters α and β for different structural diversity model by sampling and regression. We first sample a small portion of vertices with different degrees from the entire graphs and offload their structural diversity computation under three structural diversity models to the GPU, and record their actual running time. Then we obtain the two parameters by linear regression. Since different sampling ratio can lead to different estimation accuracy and also different overall running time. In this experiment, we report the overall running time under different sampling ratio. Figure 7 shows the running time against different sampling ratios under three structural diversity models on the 'Gowalla', 'LiveJournal' and 'Orkut' datasets. The results suggest that at most case, the overall performance achieves the best when the sampling rate is set to around 0.5% or 1%. Since the two settings has similar effect, we fix the sampling rate to 1% in the following experiments.

Exp-2: Efficiency comparison between different structural diversity search methods. In this experiment, we test the running time of each comparing algorithm on all datasets. The results of all

TABLE 2
Running time (in seconds) for different structural diversity search models among all competitors

Models	Graph	CPU-Div	MCPU-Div	GPU-Div	MGPU-Div	MGPU-Div-E	MGPU-Div-D	R
Comp-div	Wiki-Vote	2.93	2.08	1.95	1.88	1.85	**1.81**	1.61
	Email-Enron	4.16	3.75	2.42	2.27	2.16	**2.07**	2.0
	Epinions	9.79	7.59	5.83	4.09	4.03	**3.92**	2.49
	Gowalla	17.99	12.32	8.21	4.73	3.87	**3.63**	4.95
	NotreDame	19.83	13.78	10.07	6.76	6.47	**5.66**	3.5
	socfb-konect	6.74	5.17	4.03	3.88	3.57	**3.20**	2.1
	LiveJournal	1243.71	746.93	637.82	307.28	293.6	**275.71**	4.52
	Orkut	8682.24	4863.72	3839.56	1977.36	1923.7	**1870.30**	4.64
Core-div	Wiki-Vote	3.54	2.93	2.88	2.63	2.61	**2.57**	1.38
	Email-Enron	7.81	5.64	4.22	3.65	3.57	**3.44**	2.27
	Epinions	16.43	12.34	10.38	8.09	7.64	**7.54**	2.17
	Gowalla	25.9	14.5	9.7	4.9	4.04	**3.92**	6.61
	NotreDame	32.59	20.87	14.72	7.92	7.79	**7.37**	4.42
	socfb-konect	10.37	7.68	6.46	5.87	5.49	**4.99**	2.07
	LiveJournal	1732.1	994.6	672.2	353.2	298.7	**267.5**	6.47
	Orkut	10609.2	5473.7	4965.4	2836.6	2380.7	**2284.2**	4.64
Truss-div	Wiki-Vote	8.91	6.78	4.75	3.47	3.41	**3.38**	2.63
	Email-Enron	12.82	8.95	6.78	5.53	5.36	**4.69**	2.73
	Epinions	34.83	25.27	19.69	13.92	13.07	**12.31**	2.82
	Gowalla	54.7	38.4	28.9	19.7	17.6	**16.5**	3.31
	NotreDame	297.4	155.29	116.40	109.12	98.36	**89.49**	3.32
	socfb-konect	17.22	12.15	9.49	6.86	6.23	**5.96**	2.88
	LiveJournal	10302.8	6590.5	6321.2	4824.6	3987.2	**3669.9**	2.81
	Orkut	19023.6	12082.7	11243.8	7628.5	7022.4	**6590.6**	2.89

TABLE 3
Comparison of running time (in seconds) against existing competitors in multiple GPUs scenario

Graph	Core-based		Truss-based	
	BC-Div	MGPU-Div-D	PGL-Div	MGPU-Div-D
Gowalla	12.43	3.92	39.24	16.52
LiveJournal	702.32	267.50	6644.98	3669.89
Orkut	5121.11	2384.20	12352.70	6590.63

TABLE 4
Running time (in seconds) for Ego-network Extraction in Truss-based Structural Diversity Search

Graph	CPU-Div	MCPU-Div	MGPU-Div	R_e
Gowalla	10.7	8.6	2.8	3.07
LiveJournal	1021.3	637.2	275.1	3.71
Orkut	4218.7	3629.6	1456.5	2.89

methods with respect to different structural diversity models are reported in Table 2. The winner is highlighted in bold font. The speed up ratio in the last column is computed by the running time of the CPU sequential version CPU-Div divided by the running time of the optimized multiple GPUs solution MGPU-Div-D, i.e., $R = \frac{\text{CPU-Div}}{\text{MGPU-Div-D}}$. The results show that our method MGPU-Div-D achieves 1.38 to 6.61 times faster as the existing CPU based sequential solution, which validates the superiority of our GPU solution. Moreover, comparing to the result of MCPU-Div shown in column 4, the improvement in efficiency of our parallel GPU-based solutions is still significant as shown in columns 5-8. Observed from the result that our work stealing techniques MGPU-Div-D works well comparing to the baseline multi-GPU solution MGPU-Div on structural diversity model with more complex structure, e.g., the effectiveness is more significant on the truss-based structural diversity search reported in the last three rows. This is because more complicated computation task will cause more serious workload imbalance.

Exp-3: Efficiency comparison against existing structural diversity search competitors. In this experiment, we compare our MGPU-Div-D method with the competitors BC-Div and PGL-Div equipped with existing GPU-based k-core and k-truss decomposition techniques respectively. BC-Div and PGL-Div are equipped with sequential ego-network extraction method because they require special input data structures. And the load balancing mechanism is unavailable for these two methods. All methods are run on multiple GPUs environment. The experimental results are reported in Table 3. Our method MGPU-Div-D is the clear winner on the three representative datasets, which reflects the superiority of our proposed techniques in terms of efficiency and effectiveness. Moreover, the performance of the two competitors BC-Div and PGL-Div is even worse than some baseline methods like MCPU-Div and GPU-Div, because they have to construct the special data structure for all ego-network using naive way, which is very costly.

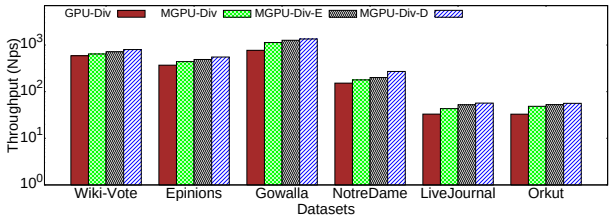


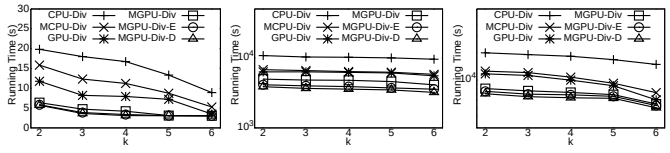
Fig. 8. Throughput (in Nps) for different work stealing strategies during truss-based structural diversity search on six datasets.

Exp-4: Ego-network extraction efficiency evaluation. In order to show the effectiveness of our GPU-based ego-network extraction technique, this experiment compares the ego-network extraction time of the CPU-based solutions CPU-Div, MCPU-Div and our GPU-based solution MGPU-Div. Without loss of generality, we report the running time of the truss-based structural diversity search in Table 4. The speed-up ratio R_s in the last column is given by $R_e = \frac{\text{CPU-Div}}{\text{MGPU-Div}}$. It shows that our GPU-accelerated ego-network extraction can achieve 2.89 to 3.71 speed-up against the CPU-based solution.

Exp-5: Effectiveness comparison for different work stealing

TABLE 5
Ablation Study: Slow Down Ratio of Removing Individual Techniques (Truss-based).

Competitor	Gowalla	LiveJournal	Orkut
NoGPUEgo	1.46X	1.19X	1.39X
NoMultipleGPU	1.64X	1.58X	1.60X
NoWorkBalance	1.19X	1.31X	1.16X
FullVersion	1X	1X	1X



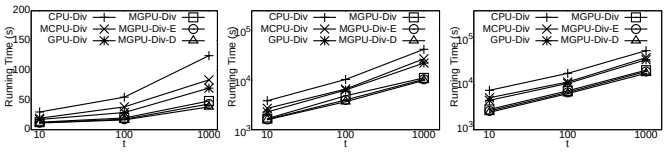
(a) Gowalla (b) LiveJournal (c) Orkut

Fig. 9. Running time (in seconds) for different setting of k in truss-based structural diversity search.

strategies. To compare the work-stealing optimization techniques for workload balancing, this experiment reports the throughput of methods GPU-Div, MGPU-Div, MGPU-Div-E and MGPU-Div-D during truss-based structural diversity search on six datasets. Here the throughput of a method is measured in the number of vertices the GPUs can process per second (Nps). The larger the value is, the better the effectiveness is. The result is reported in Figure 8. According to Figure 8(b) and (c), we can observe that the effectiveness of the exhausted work stealing technique MGPU-Div-E degrades when the ego-network structures become more complicated in larger graphs. Its improvement upon the baseline MGPU-Div is very minor on LiveJoural and Orkut datasets. This is because the data skewness tends to be much more serious on larger graphs. However, the throughput is significantly improved by MGPU-Div-D, which reveals the superiority of our global work-stealing techniques for workload balancing in terms of effectiveness.

Exp-6: Ablation study. In this experiment, we provide an ablation study to test the effectiveness of each individual technique. Table 5 reports the speedup ratio of MGPU-Div-D against each competitor that removes a corresponding technique. The results of 'NoGPUEgo' are computed by the running time of the MGPU-Div-D using sequential ego-network extraction method divided by the running time of MGPU-Div-D equipped with GPU-based ego-network extraction. Similarly, the results of 'NoMultipleGPU' are computed by the running time of a single GPU-based solution divided by the multiple GPU-based solution, i.e., $\frac{\text{GPU-Div}}{\text{MGPU-Div}}$. MGPU-Div achieves 1.58X to 1.64X speed up compared to GPU-Div. The speed up ratio is not linear to the increase of the number of GPUs, since the truss-based structural diversity search still contains atomic operations such as buffer array copying and conditional branching, which cannot fully parallelize and can cause warp divergence. The non-linear speed up feature can be also observed from the results of the two CPU solutions CPU-Div and MCPU-Div. The results of 'NoWorkBalance' are computed by the running time of the multiple-GPUs-based solution without workload balance optimization techniques divided by the running time of the solution equipped with all workload balance optimization techniques, i.e., $\frac{\text{MGPU-Div}}{\text{MGPU-Div-E}}$. The result shows the significant effectiveness of our proposed techniques.

Exp-7: Running times for different setting of parameters k and t. In this experiments, we vary the setting of parameter k and t and report the running times for the truss-based structural diversity search on three datasets 'Gowalla', 'LiveJournal', and



(a) Gowalla (b) LiveJournal (c) Orkut

Fig. 10. Running time (in seconds) for different setting of t in truss-based structural diversity search.

'Orkut'. Firstly, parameter t is fixed to $t = 100$, we vary parameter k in $\{2, 3, 4, 5, 6\}$ and report the running times of all methods in Figure 9. As the value of k increases, the running times for all methods reduce because the early termination condition is easier to trigger for larger k, which leads to a much smaller search space. Secondly, we fix parameter k to 3, and vary parameter t in $\{10, 100, 1000\}$. The running times for all methods are reported in Figure 10. The running times of all methods grow as the magnitude of t increases. However, we can observe that the growing speed of the running times of the CPU-based methods is faster than the GPU-based methods, which shows the robustness of our GPU-based solution.

8 Related Work

Our work is related to structural diversity search, GPU-based graph computing, and top-k search on graphs.

Structural diversity search. Social decisions can significantly depend on the social network structure [26], [27]. Ugander et al. [1] conducted extensive studies on the Facebook to show that the contagion probability of an individual is strongly related to its structural diversity in the ego-network . A follow-up work by Su et al. [28] conduct experimental studies on the correlation of the structural diversity and the retention. Similarly, Graham et al. [29] provide experimental studies to analyze the connection between structural diversity and information diversity. Motivated by [1], Huang et al. [2] studies the problem of structural diversity search to find k vertices with the highest structural diversity in graphs. To improve the efficiency of [2], Chang et al. [3] propose a scalable algorithm by enumerating each triangle at most once in constant time, which is a fast solution on the sequential component-based structural diversity search. However, it needs to maintain and update the connected component in the ego-network of every vertex using a disjoint-set data structure, which is difficult for extending to the parallel environment directly. Moreover, note that a recent study [6] shows that the index-based approaches can achieve much faster query time than [3], reflecting that the index construction is more feasible to parallelize in the future study. Moreover, the method proposed by [3] focuses on the component-based structural diversity model only. In this paper, we propose to study an one-off GPU-based parallel solution for all three structural diversity models, including the component-based model, core-based model, and truss-based model. Structural diversity search based on a different k-core model is further studied in [7]. Recently a parameter-free approach for the core-based model is proposed in [4], and is extended to a parallel version based on CPU environment in [5]. The problem of structural diversity search based on k-truss and the GCT-index is investigated by the study in [6]. Beside of the studies of structural diversity of a vertex, Zhang et al. [30] propose a new definition of edge-based structural diversity to study the structural diversity of an edge. Recently, Chen et al. [31] extends the structural diversity search problem to streaming graphs. Different from the above research

works that are conducted in CPU environment, in this paper, we study the structural diversity search problem on static graphs for three structural diversity models in GPU environment.

GPU-based graph computing. GPU is a new hardware equipped with massive computing units. Powered by the CUDA programming model, GPU is recently used to accelerate many traditional graph algorithms such as general graph processing [32], [33], [34], constraint shortest path [10], subgraph matching [11], [12], [35], label propagation [36], etc. Triangle counting and truss decomposition are two fundamental problem in subgraph mining. Recently, a great amount of work study the acceleration for this two problem leveraging the powerful computing resources of GPU. For triangle counting, the very basic problem of it is set intersection. State-of-the-art studies mainly work on three ways of parallel set intersection: merge-based, hash-based, bitmap-based and binary search based set intersection [37], [38], [39], [40], [8]. And the binary search based set intersection proposed in [8] is known to be the best in terms of performance in single GPU environment. To solve the workload imbalance problem in GPU-based triangle counting, a recent work [9] propose a general workload balance model to accelerate the existing GPU-based triangle counting algorithms. Comparatively, recent studies of GPU-based k-core decomposition and k-truss decomposition is more related to the sub-steps of our structural diversity search problem. For GPU-based k-core decomposition, BC-Div [13] is the SOTA method that can achieve the best performance in single GPU environment. Since k-core decomposition is just a sub-step of the core-based structural diversity search problem, in order to maximize the overall performance, we focus on solutions in multiple GPUs scenario, which will raise the problem of cross-GPU workload balancing. Existing GPU-based graph-parallel systems such as Medusa [14] and GUNROCK [22] propose multi-GPU solutions for k-core decomposition. However, they only focus on the k-core decomposition on the entire graph. In our structural diversity search problem, only offloading the k-core decomposition step for small subgraphs like ego-network to multiple GPUs is obviously an under utilization of the GPU hardware, which is impractical. For truss decomposition, PKT [24] is a CPU-based parallel solution that can achieve good performance for shared-memory system. Recently, Che et al. [19] perform detail optimization based on the PKT algorithm with CPU+GPU environment. PGL-Div [15] is a pure single GPU-based solution for the truss decomposition problem. However, it needs extra processing time for constructing the required auxiliary data structure COO. In our structural diversity problem, we need to construct the COO structure for the ego-network of all the vertices, which is an extremely expensive cost. In this paper, we study a new problem of GPU-based structural diversity search, and propose problem-oriented detailed solution based on GPU environment.

Top-k search on graphs. Top-k query processing is an important direction in graph analytics. The problem of top-k keyword search on graphs [41], [42], [43], [44] aims at finding the k closest vertices whose labels contain the input keywords. The goal of another typical problem of top-k graph pattern matching [45], [46], [47] is to find top-k subgraphs that match the input graph pattern. Recently, a new topic of top-k community search [48], [49], [50], [51] appears to discover the k most important communities on an input graph. In a recent study, Ye et al. [52] study a keyword-based top-k community search problem, which finds top-k communities with both highest keyword relevance and structural cohesiveness. In our study, we focus on finding the top-k vertices with highest structural diversity scores.

9 Conclusion

In this paper, we study the structural diversity search problem for three structural diversity models in CPU+GPU environment. To solve this problem, we propose a one-off parallel framework to leverage the GPU to accelerate the structural diversity computation. In our framework, we propose efficient GPU-based structural diversity computation techniques including a GPU-friendly lock-free ego-network extraction approach and fine-grained GPU-based parallel solutions to the ego-network decomposition according to three structural diversity models. Moreover, to address the workload imbalance issue, we design an efficient work packing scheme before computing and an effective dynamic work stealing strategy to redistribute the workload at run time. Experiments on several real-world datasets show that our method can achieve up to 6.6X speed up against the baseline sequential approach, which validates the superior efficiency and effectiveness of our proposed techniques.

10 Acknowledgment

This work is supported by National Key R&D Program of China 2023YFC3321300, Hong Kong RGC Projects Nos. 12201923, 12200424, 12202221, 12202024, C2003-23Y, RIF R1015-23, GRF HKBU12203123, NSFC Grant No. 62102341, Guangdong Basic and Applied Basic Research Foundation (Project No. 2023B1515130002), NSF of China 62472116, and NSF of Guangdong Province 2023A1515030273. Xin Huang is the corresponding author.

References

[1] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg, "Structural diversity in social contagion," *PNAS*, vol. 109, no. 16, pp. 5962–5966, 2012.

[2] X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu, "Top-k structural diversity search in large networks," *PVLDB*, vol. 6, no. 13, pp. 1618–1629, 2013.

[3] L. Chang, C. Zhang, X. Lin, and L. Qin, "Scalable top-k structural diversity search," in *ICDE*, 2017, pp. 95–98.

[4] J. Huang, X. Huang, Y. Zhu, and J. Xu, "Parameter-free structural diversity search," in *WISE*. Springer, 2020, pp. 677–693.

[5] J, Huang and X, Huang and Y, Zhu and J, Xu, "Parallel algorithms for parameter-free structural diversity search on graphs," *WWWJ*, vol. 24, pp. 397–417, 2021.

[6] J. Huang, X. Huang, and J. Xu, "Truss-based structural diversity search in large graphs," *TKDE*, 2020.

[7] X. Huang, H. Cheng, R. Li, L. Qin, and J. X. Yu, "Top-k structural diversity search in large networks," *VLDB J.*, vol. 24, no. 3, pp. 319–343, 2015.

[8] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC*, 2018, pp. 171–182.

[9] L. Hu, L. Zou, and Y. Liu, "Accelerating triangle counting on gpu," in *SIGMOD*, 2021, pp. 736–748.

[10] S. Lu, B. He, Y. Li, and H. Fu, "Accelerating exact constrained shortest paths on gpus," *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 547–559, 2020.

[11] G. Jiang, Q. Zhou, T. Jin, B. Li, Y. Zhao, Y. Li, and J. Cheng, "VSGM: view-based gpu-accelerated subgraph matching on large graphs," in *SC*. IEEE, 2022, pp. 52:1–52:15.

[12] X. Sun and Q. Luo, "Efficient gpu-accelerated subgraph matching," *PACMMOD*, vol. 1, no. 2, pp. 181:1–181:26, 2023.

[13] A. Ahmad, L. Yuan, D. Yan, G. Guo, J. Chen, and C. Zhang, "Accelerating k-core decomposition by a gpu," in *ICDE*, 2023, pp. 1818–1831.

[14] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *TPDS*, vol. 25, no. 6, pp. 1543–1552, 2013.

[15] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss decomposition on gpu," in *HPEC*. IEEE, 2019, pp. 1–7.

[16] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 458–473, 2008.

[17] F. Ding and Y. Zhuang, "Ego-network probabilistic graphical model for discovering on-line communities," *Appl. Intell.*, vol. 48, no. 9, pp. 3038–3052, 2018.

[18] J. Mcauley and J. Leskovec, "Discovering social circles in ego networks," *TKDD*, vol. 8, no. 1, p. 4, 2014.

[19] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo, "Accelerating truss decomposition on heterogeneous processors," *PVLDB*, vol. 13, no. 10, pp. 1751–1764, 2020.

[20] K. Date, K. Feng, R. Nagi, J. Xiong, N. S. Kim, and W.-M. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition on the minsky architecture: Static graph challenge: Subgraph isomorphism," in *HPEC*. IEEE, 2017, pp. 1–7.

[21] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition," in *HPEC*. IEEE, 2018, pp. 1–7.

[22] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017.

[23] A. E. Sarıyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proceedings of the VLDB Endowment*, vol. 12, no. 1.

[24] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," in *HiPC*. IEEE, 2017, pp. 13–22.

[25] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[26] J. H. Fowler and N. A. Christakis, "Cooperative behavior cascades in human social networks," *PNAS*, p. 200913149, 2010.

[27] Y. Dong, R. A. Johnson, J. Xu, and N. V. Chawla, "Structural diversity and homophily: A study across more than one hundred big networks," in *KDD*. ACM, 2017, pp. 807–816.

[28] J. Su, K. Kamath, A. Sharma, J. Ugander, and S. Goel, "An experimental study of structural diversity in social networks," in *ICWSM*. AAAI Press, 2020, pp. 661–670.

[29] A. V. Graham, J. McLevey, P. Browne, and T. Crick, "Structural diversity is a poor proxy for information diversity: Evidence from 25 scientific fields," *Soc. Networks*, vol. 70, pp. 55–63, 2022.

[30] Q. Zhang, R.-H. Li, Q. Yang, G. Wang, and L. Qin, "Efficient top-k edge structural diversity search," in *ICDE*. IEEE, 2020, pp. 205–216.

[31] K. Chen, D. Wen, W. Zhang, Y. Zhang, X. Wang, and X. Lin, "Querying structural diversity in streaming graphs," *Proc. VLDB Endow.*, vol. 17, no. 5, pp. 1034–1046, 2024.

[32] P. Cui, H. Liu, B. Tang, and Y. Yuan, "Cggraph: An ultra-fast graph processing system on modern commodity CPU-GPU co-processor," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1405–1417, 2024.

[33] Y. Zhang, Y. Liang, J. Zhao, F. Mao, L. Gu, X. Liao, H. Jin, H. Liu, S. Guo, Y. Zeng, H. Hu, C. Li, J. Zhang, and B. Wang, "Egraph: Efficient concurrent gpu-based dynamic graph processing," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 6, pp. 5823–5836, 2023.

[34] K. Meng, L. Geng, X. Li, Q. Tao, W. Yu, and J. Zhou, "Efficient multi-gpu graph processing with remote work stealing," in *ICDE*. IEEE, 2023, pp. 191–204.

[35] L. Hu, L. Zou, and M. T. Özsu, "GAMMA: A graph pattern mining framework for large graphs on GPU," in *ICDE*. IEEE, 2023, pp. 273–286.

[36] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "Large-scale graph label propagation on gpus," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 10, pp. 5234–5248, 2024.

[37] C. Gui, L. Zheng, P. Yao, X. Liao, and H. Jin, "Fast triangle counting on gpu," in *HPEC*. IEEE, 2019, pp. 1–7.

[38] Y. Hu, H. Liu, and H. H. Huang, "High-performance triangle counting on gpus," in *HPEC*. IEEE, 2018, pp. 1–5.

[39] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *HPEC*, 2017, pp. 1–7.

[40] J. Zhang, D. G. Spampinato, S. McMillan, and F. Franchetti, "Preliminary exploration of large-scale triangle counting on shared-memory multicore system," in *HPEC*. IEEE, 2018, pp. 1–6.

[41] M. Kargar and A. An, "Efficient top-k keyword search in graphs with polynomial delay," in *ICDE*. IEEE Computer Society, 2012, pp. 1269–1272.

[42] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian, "Top-k nearest keyword search on large graphs," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 901–912, 2013.

[43] Q. Zhu, H. Cheng, and X. Huang, "I/o-efficient algorithms for top-k nearest keyword search in massive graphs," *VLDB J.*, vol. 26, no. 4, pp. 563–583, 2017.

[44] Y. Yang, D. Agrawal, H. V. Jagadish, A. K. H. Tung, and S. Wu, "An efficient parallel keyword search engine on knowledge graphs," in *ICDE*. IEEE, 2019, pp. 338–349.

[45] J. Cheng, X. Zeng, and J. X. Yu, "Top-k graph pattern matching over large graphs," in *ICDE*. IEEE Computer Society, 2013, pp. 1033–1044.

[46] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *Proc. VLDB Endow.*, vol. 6, no. 13, pp. 1510–1521, 2013.

[47] X. Wang and H. Zhan, "Approximating diversified top-k graph pattern matching," in *DEXA (1)*, ser. Lecture Notes in Computer Science, vol. 11029. Springer, 2018, pp. 407–423.

[48] J. Xu, X. Fu, Y. Wu, M. Luo, M. Xu, and N. Zheng, "Personalized top-n influential community search over large social networks," *WWWJ*, vol. 23, no. 3, pp. 2153–2184, 2020.

[49] N. Rai and X. Lian, "Top-k community similarity search over large road-network graphs," in *ICDE*. IEEE, 2021, pp. 2093–2098.

[50] W. Luo, X. Zhou, J. Yang, P. Peng, G. Xiao, and Y. Gao, "Efficient approaches to top-r influential community search," *IOT*, vol. 8, no. 16, pp. 12 650–12 657, 2021.

[51] R. Sun, Y. Wu, and X. Wang, "Diversified top-r community search in geo-social network: A k-truss based model," in *EDBT*. OpenProceedings.org, 2022, pp. 2:445–2:448.

[52] J. Ye, Y. Zhu, and L. Chen, "Top-r keyword-based community search in attributed graphs," in *ICDE*. IEEE, 2023, pp. 1652–1664.

Jinbin Huang received the PhD degree from Hong Kong Baptist University (HKBU) in 2024. His research interests include graph data management and GPU-accelerated graph algorithms.

Xin Huang received the PhD degree from the Chinese University of Hong Kong (CUHK) in 2014. He is currently an Associate Professor at Hong Kong Baptist University. His research interests mainly focus on graph data management and mining.

Jianliang Xu is a Professor in the Department of Computer Science, Hong Kong Baptist University (HKBU). He held visiting positions at Pennsylvania State University and Fudan University. He has published more than 150 technical papers in these areas, most of which appeared in leading journals and conferences including SIGMOD, VLDB, ICDE, TODS, TKDE, and VLDBJ.

Byron Choi is a Professor in the Department of Computer Science at the Hong Kong Baptist University. He received the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002and 2006, respectively. His research interestsinclude graph data management and time series data analysis.

Yun Peng Yun Peng received the PhD degree from Hong Kong Baptist University. He is the associate head of the Department of Artificial Intelligence, Guangzhou University.