Accelerating Triangle-Connected Truss Community Search Across Heterogeneous Hardware

JUNCHAO MA, School of Computer Science, Wuhan University, China
XIN YAN, School of Computer Science, Wuhan University, China
YUANYUAN ZHU*, School of Computer Science, Wuhan University, China
GUOJING LI, School of Computer Science, Wuhan University, China
HAO ZHANG, The Chinese University of Hong Kong, Hong Kong, China
JEFFREY XU YU, The Hong Kong University of Science and Technology (Guangzhou), China

k-truss is one of the most widely used community models where each edge is contained in at least k-2 triangles, and *Triangle-connected k-Truss Community* (k-TTC) is a strengthened variant of k-truss that further requires edges to reach each other via a series of adjacent triangles. EquiTree is the state-of-the-art tree-structured index, which is space efficient and can support time-optimal k-TTC search for query vertices. However, for large graphs with billions of edges, the sequential algorithm still needs seconds to conduct k-TTC search and hours to construct the index due to the costly triangle connectivity examination. In this paper, we study how to accelerate k-TTC search by hardware accelerators. Specifically, we propose a tensor-based framework, including index construction, online search, and index maintenance, which can be efficiently deployed and run on heterogeneous hardware. To accelerate index construction, we first propose an iterative basic algorithm which creates and refines supernodes and superedges based on each k-class layer by layer. To further boost the parallelism, we propose a triangle-based supernode and superedge creation strategy, which categorizes triangles into three types (*internal*, *marginal*, *external*), and applies tailored operations for each type to enable batch processing instead of iterative steps. Meanwhile, we propose the batch-based merging strategy to refine the index into a tree structure. We also propose tensor-based algorithms for online k-TTC search and index maintenance. Extensive experiments show that our tensor-based algorithms achieve an average speedup of two orders of magnitude over state-of-the-art methods in both index construction and community search, while efficiently maintaining indices for dynamic graphs. Moreover, we validated that our tensor-based algorithms can be smoothly deployed and run on heterogeneous hardware accelerators (NVIDIA GPUs and AMD GPUs) for acceleration.

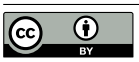
CCS Concepts: • **Information systems** → **Social networks**.

Additional Key Words and Phrases: community search, triangle connectivity, heterogeneous hardware

ACM Reference Format:
Junchao Ma, Xin Yan, Yuanyuan Zhu, Guojing Li, Hao Zhang, and Jeffrey Xu Yu. 2026. Accelerating Triangle-Connected Truss Community Search Across Heterogeneous Hardware. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 6 (February 2026), 26 pages. https://doi.org/10.1145/3786620

*Yuanyuan Zhu is the corresponding author.

Authors' Contact Information: Junchao Ma, School of Computer Science, Wuhan University, China, junchaoma@whu.edu.cn; Xin Yan, School of Computer Science, Wuhan University, China, xin_yan@whu.edu.cn; Yuanyuan Zhu, School of Computer Science, Wuhan University, China, yyzhu@whu.edu.cn; Guojing Li, School of Computer Science, Wuhan University, China, guojingli@whu.edu.cn; Hao Zhang, The Chinese University of Hong Kong, Hong Kong, China, zhanghaowuda12@gmail.com; Jeffrey Xu Yu, The Hong Kong University of Science and Technology (Guangzhou), China, jeffreyxuyu@hkust-gz.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/2-ART6
https://doi.org/10.1145/3786620

1 Introduction

Large-scale graphs in the real world are usually globally sparse but locally dense [5]. The locally dense subgraphs (often referred to as cohesive subgraphs or communities) are the key features of the graph. Existing works on communities primarily focus on *Community Detection* (CD) to find all communities in the graph [33, 41, 46] or *Community Search* (CS) to find communities containing specific vertices [18, 47], where the latter is widely used to explore the communities for interested entities, such as personal context discovery in social networks [4], group search in geo-social networks [7, 30], complex search in protein-protein interaction networks [36].

Many community models have been proposed to meet diverse topology requirements, such as clique/quasi-clique [12, 42], k-core [26, 38], and k-truss [44]. Most of them either make community search NP-hard (clique/quasi-clique) or have low cohesiveness (k-core). k-truss provides a balance, requiring that each edge occurs in at least $k-2$ triangles for higher cohesiveness, while remaining polynomial-time tractable. Triangle connectivity is further imposed on k-truss [21] to strengthen the cohesiveness, i.e., edges are connected through a chain of edge-adjacent triangles (share a common edge), while still maintaining the same time complexity. Such a model is called the Triangle-connected k-Truss Community (k-TTC). As shown in Fig. 1, in the 4-truss (blue solid and red dashed edges), edges among nodes from 0 to 6 and edges among $\{3, 7, 8, 9\}$ form two separate cohesive 4-TTCs as they are not triangle-connected. A k-TTC with n vertices has a tighter diameter bound than that of a k-truss [47]. Therefore, k-TTC is widely adopted in attributed [49], size-constrained [32], or user-preferred community search [48], as well as code graph analysis [35].

In the literature, a variety of indices have been proposed to support efficient k-TTC search for given vertices. TCP-index [21] constructs a series of Maximum Spanning Trees (MST), where edge weights represent the precomputed trussness. However, edges in the original graph may appear in multiple MSTs, and k-TTC search requires access to both the index and the original graph. EquiTruss [2] builds a summary graph based on k-truss equivalence classes to preserve trussness and triangle connectivity, enabling direct k-TTC search on the summary graph. The construction of EquiTruss was further accelerated by multi-core CPUs [19], where triangle connectivity is examined by parallel Connected Component (CC) computation. Nevertheless, the summary graph of EquiTruss in practice may be even larger than the original graph due to the fine granularity of k-truss equivalence classes, leading to high search cost [47]. EquiTree [47] merges multiple k-truss equivalence classes into a coarser-grained k-partial class to build a more compact tree-structured index. EquiTree is space-efficient ($O(m)$ where m is the number of edges in the graph) and supports time-optimal k-TTC search ($O(|\mathcal{A}|)$, where $\mathcal{A}$ is the size of returned communities) at slightly higher index construction and maintenance cost. Despite theoretical optimality, the CPU-based sequential algorithms [47] may still take hours to build EquiTree and seconds to search k-TTCs online for real-world graphs with billions of edges (e.g., the Orkut dataset to be shown in Section 6).

In this paper, we study the acceleration of EquiTree by uniformly leveraging the parallel capabilities of heterogeneous hardware accelerators (e.g., GPUs, TPUs, NPUs), based on Tensor Computation Runtimes (TCRs) which refer to Deep Learning (DL) frameworks (e.g., PyTorch, TensorFlow) and their compilers/runtimes (e.g., TVM, ONNX). TCRs provide a set of hardware-independent tensor operators, enabling users to leverage the parallel computation power of heterogeneous hardware backends without delving into their specific characteristics. Besides DL tasks, TCRs have been exploited to accelerate non-DL tasks such as relational queries [20], pagerank computation [27], and subgraph matching [28], demonstrating the potential of accelerating non-DL tasks.

However, accelerating EquiTree construction and k-TTC search based on TCRs poses significant challenges, as existing CPU-based algorithms [47] iteratively create supernodes/superedges and search k-TTC for each trussness value k. Directly converting such an iterative paradigm into

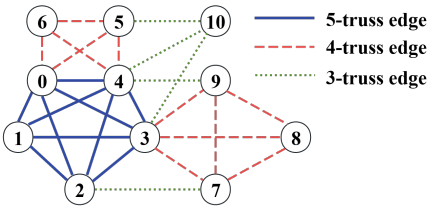


Fig. 1. An example graph

tensor programs will diminish parallelism, as tensor operators are optimized for large-batched data processing rather than fragmented, iterative computations to maximize the parallelism of hardware accelerators and minimize the overhead of frequent kernel launches. Thus, we revisit EquiTree by exploiting the properties of k-triangles, which can be further classified into *internal*, *marginal*, and *external* k-triangles based on the trussness of the three constituent edges. These three types of triangles have distinct properties and serve different roles during index construction. With tailored operations on different k-triangles, we can create all the supernodes for k-truss equivalence classes and superedges between equivalence classes simultaneously. We further refine the supernodes and superedges to the tree structure by merging all the conflicting truss equivalence classes into k-partial classes in batches. We also propose the tensor-based k-TTC search algorithm on EquiTree and the EquiTree maintenance algorithm for dynamic changes of graphs. Compared with the CPU-based algorithms, our tensor-based algorithms can accelerate index construction and k-TTC search by two orders of magnitude on average while maintaining efficient index updates. We summarize our contributions below:

- We develop the first tensor-based k-TTC search framework, which can efficiently construct/maintain EquiTree and support k-TTC search by leveraging the parallel computation capabilities of heterogeneous hardware accelerators without exploring their specific characteristics or primitives.
- To avoid the costly iteration on trussness k, we analyze the properties of k-triangles and propose a triangle-centric strategy to create all the supernodes and superedges simultaneously and a batch-based merging strategy to refine the index into the tree structure, which can maximize the parallelism of the hardware accelerators.
- We conduct extensive experimental studies, which show that our method can outperform existing baselines on both index construction and k-TTC search with an average speedup of two orders of magnitude while maintaining efficient index updates. Moreover, we validate that our tensor-based algorithms can be smoothly deployed and run on heterogeneous hardware accelerators for acceleration.

Roadmap. Section 2 introduces the background. Sections 3 and 4 give the basic and advanced index construction methods. Section 5 discusses k-TTC search and index maintenance. Experiments are reported in Section 6. Section 7 reviews the related work, and Section 8 concludes the paper.

2 Background

We first introduce the k-TTC search problem and tensor operators, and then review the state-of-the-art methods on k-TTC search.

2.1 Preliminaries

2.1.1 The k-TTC Search Problem. We use $G = (V(G), E(G))$ to denote a simple undirected graph, where $V(G)$ and $E(G)$ are the vertex set and edge set respectively. Let $n = |V(G)|$ and $m = |E(G)|$

denote the vertex number and edge number, respectively. The set of neighboring vertices of vertex $v \in V(G)$ is defined as $N(v, G) = \{u \in V(G) \mid (u, v) \in E(G)\}$. A triangle in G is a 3-length cycle, defined as $\triangle_{uvw}^{G} = \{(u, v), (v, w), (w, u) | (u, v), (v, w), (w, u) \in E(G)\}$. When the context is clear, we simplify $V(G)$, $E(G)$, $N(v, G)$, and $\triangle_{uvw}^{G}$, as V, E, $N(v)$, and $\triangle_{uvw}$, respectively.

Definition 1. *(***Edge Support***) The support of an edge $e = (u, v)$ in G, denoted as $sup(e, G)$, is the number of triangles containing e, i.e., $sup(e, G) = |\{\triangle_{uvw}^{G} \mid w \in V(G)\}|$.*

Definition 2. *(*k***-Truss*** [10]) A k-truss in G is a subgraph H s.t. $\forall e \in E(H)$, $sup(e, H) \geq k - 2$.*

The trussness of a subgraph H is the minimum support of all edges in H plus 2, defined as $\tau(H) = \min_{e \in E(H)}(sup(e, H) + 2)$. The trussness of an edge $e \in E(G)$ is the maximum trussness of subgraphs containing e, defined as $\tau(e) = \max_{e \in E(H) \wedge H \subseteq G} \tau(H)$. A k-truss H is maximal if there is no H' s.t. $\tau(H') \geq k$ and $H \subset H'$.

Two edges e_1 and e_2 are *triangle-connected*, denoted as $e_1 \leftrightarrow e_2$, iff (1) e_1 and e_2 are in the same triangle, or (2) there exists a sequence of edge-adjacent triangles $\triangle_1, \ldots, \triangle_l (l \geq 2)$ s.t. $e_1 \in \triangle_1, e_2 \in \triangle_l$, $\triangle_i \cap \triangle_{i+1} \neq \varnothing$ for $1 \leq i < l$.

Definition 3. (**Triangle-connected** k**-Truss Community** (k**-TTC**)) *A subgraph H is a k-TTC if: (1) $\tau(H) \geq k$; (2) $\forall e, e' \in E(H)$, $e \leftrightarrow e'$; (3) $\nexists H' \supsetneq H$ s.t. H' satisfies (1) and (2).*

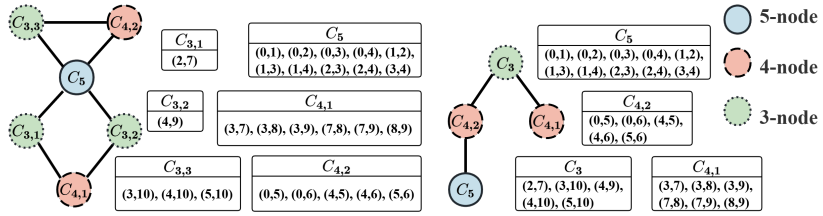
Problem. (**The** k**-TTC Search Problem**) *Given a graph $G = (V, E)$, a query vertex $v_q \in V(G)$, and an integer $k \geq 3$, find all the k-TTCs containing v_q.*

As shown in Fig. 1, the support of edge $(7, 8)$ is 2 as it is contained in $\triangle_{\{3,7,8\}}$ and $\triangle_{\{7,8,9\}}$. The subgraph consisting of blue solid and red dashed edges is a 4-truss but not a 4-TTC, as edges among $\{0, \ldots, 6\}$ and edges among $\{3, 7, 8, 9\}$ are not connected by a series of edge-adjacent 4-triangles, while the subgraph induced by $\{3, 7, 8, 9\}$ and that induced by $\{0, \ldots, 6\}$ are two separate 4-TTCs.

2.1.2 Tensor and Its Operators. A tensor, abstracted as a multidimensional array, can be a 0-dimensional scalar, 1-dimensional vector, 2-dimensional matrix, or a high-dimensional array. TCRs provide a set of tensor operators to manipulate tensors. Take PyTorch as an example. The common tensor operators include: (1) *Initialization*: create a tensor, e.g., 0/1 tensors (`zeros`/`ones`), empty tensors (`empty`), ordered tensors (`arange`); (2) *Selection*: select elements from a tensor by numerical or boolean index (e.g., `tensor[indices]`, `tensor[mask]`); (3) *Comparison*: compare tensors (e.g., `eq`, `lt`, `gt`, `le`, `ge`); (4) *Arithmetic*: perform basic arithmetic operations on tensors, e.g., basic operators (+, −, ×, ÷), in-place operators (`add_`, `sub_`); (5) *Reorganization*: rearrange elements in a tensor or concatenate multiple tensors (e.g., `stack`, `reshape`, `sort`, `cat`); (6) *Reduction*: aggregate over a tensor or get unique values from a tensor (e.g., `max`, `min`, `sum`, `mean`, `unique`). As shown in [28], most of the above tensor operators have work complexity $O(l)$ and step complexity $O(\log(l))$ where l is the length of the input tensor. Since step complexity measures the total parallel steps and work complexity quantifies the overall workload, we focus on reducing step complexity to boost the parallel computation on TCRs.

2.2 State-of-the-art Methods on k-TTC Search

2.2.1 EquiTruss [2, 19]. EquiTruss is a supergraph index built on *k-truss equivalence classes*. By imposing the constraint of trussness k, two triangle-connected edges e_1 and e_2 can be strengthened as *k-triangle-connected* ($e_1 \overset{k}{\leftrightarrow} e_2$), if (1) e_1 and e_2 are in the same *k-triangle* $\triangle$ ($\min_{e \in \triangle} \tau(e) \geq k$), or (2) there exists a sequence of *k-triangles* $\triangle_1, \ldots, \triangle_l$ s.t. $e_1 \in \triangle_1, e_2 \in \triangle_l$, $\triangle_i \cap \triangle_{i+1} \neq \varnothing$ and $\tau(\triangle_i \cap \triangle_{i+1}) = k$ for $1 \leq i < l$. A *k-truss equivalence class* consists of edges that are *k-triangle-connected* and share the same trussness k. All the truss equivalence classes form a mutually



(a) The EquiTruss index **(b) The EquiTree index**

Fig. 2. The EquiTruss and EquiTree indices

exclusive and collectively exhaustive partition of E. Each truss equivalence class is represented by a supernode, and two supernodes are connected by a superedge if they are k-triangle-connected (k is their minimum trussness). The k-TTC search for a query vertex v_q can be conducted directly on EquiTruss by first identifying supernodes containing v_q and then retrieving the maximal connected components (CC) containing these supernodes.

Fig. 2 (a) shows the EquiTruss index for the graph in Fig. 1, where each supernode represents an equivalence class. Edges in $C_{3,1}/C_{3,2}/C_{3,3}$ are 3-triangle-connected, edges in $C_{4,1}/C_{4,2}$ are 4-triangle-connected, and $C_{3,3}$ links to $C_{4,2}$ as $(3, 10)$ is 3-triangle-connected with $(4, 5)$ in $C_{4,2}$. To find the 4-TTCs for v_6, we locate $C_{4,2}$ containing v_6, find the maximal CC induced by $\{C_5, C_{4,2}\}$ with trussness at least 4, and return the edges contained in them.

After the preliminary step of truss decomposition to obtain the trussness of all the edges [44], EquiTruss can be sequentially constructed as follows. For each k from 3 to $k_{\max}$, examine each edge in the k-class $\Phi_k = \{e \mid e \in E(G) \wedge \tau(e) = k\}$ to create supernodes and superedges. Specifically, for each examined edge e, create a new supernode x representing its equivalence class, where e is considered as the initial seed edge and equivalent edges are gradually added to the supernode by BFS exploration of its adjacent k-triangles. For an edge e' in the k-triangle containing e with trussness $k' > k$, add supernode x to the list of e' to indicate that a superedge will be added between x and the supernode x' of e' when x' is created. After processing all k-triangles related to e, remove e and move to the next edge, until all edges in Φ_k have been examined.

The parallel algorithm [19] separates the creation of supernodes and superedges for each Φ_k. (1) *Create supernodes.* For all edges in Φ_k, supernodes for k-truss equivalence classes are created based on parallel CC computation. Here, each edge is considered as a vertex in the component computation, and two edges are connected as one component if they are k-triangle-connected. Specifically, each edge in Φ_k is initialized as its own parent component, then triangles incident to each e are examined in parallel by two steps: *Link* and *Compress*. *Link* connects the constituent edges to the same parent component of e if k-triangle connectivity is satisfied. Then, *Compress* runs in parallel across all edges in Φ_k with continuous linking up to the parent until all edges under the component tree are directly connected to the root. (2) *Create superedges.* First, allocate a superedge subset vector with a size equal to the number of available parallel threads so that each thread can add elements to its own superedge subset to avoid race conditions. Then, find the triangle-forming edges e_1 and e_2 with trussness less than k for each $e \in \Phi_k$ in parallel. Next, create a superedge between the supernode containing the current edge e and the supernode containing e_1 or e_2 with the minimum trussness, and add it to the superedge subset of the corresponding thread. After the above iterations from 3 to $k_{\max}$, the supergraph is finally obtained by eliminating duplicate superedges created by multiple threads.

2.2.2 EquiTree [47]. EquiTree is a tree-structured index built on a coarser-grained partition of the edge set, *k-partial classes*, defined by relaxing the condition of k-triangle connectivity to k^+-triangle

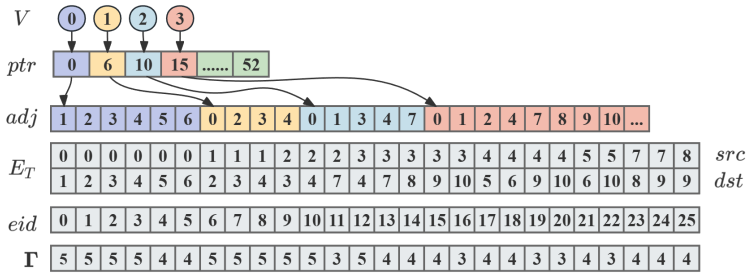


Fig. 3. Representation of the example graph

connectivity. Edges e_1 and e_2 are k^+*-triangle-connected* ($e_1 \overset{\geq k}{\leftrightarrow} e_2$), if (1) e_1 and e_2 are in the same k-triangle, or (2) there exists a sequence of k-triangles $\triangle_1, \ldots, \triangle_l$ s.t. $e_1 \in \triangle_1, e_2 \in \triangle_l$, $\triangle_i \cap \triangle_{i+1} \neq \varnothing$ and $\tau(\triangle_i \cap \triangle_{i+1}) \geq k$ for $1 \leq i < l$. A *k-partial class* is a subset of k^+-triangle-connected edges with trussness k. Two k-truss equivalence classes belong to the same k-partial class iff they are k-triangle-connected with another k'-truss equivalence class ($k' > k$). A partial class P_1 *truss-precedes* another partial class P_2 (denoted $P_1 \prec P_2$), if $\forall e_1 \in P_1, e_2 \in P_2, \tau(e_1) < \tau(e_2)$ and $e_1 \overset{\geq \tau(e_1)}{\longleftrightarrow} e_2$. EquiTree takes partial classes as supernodes, and connects any two partial classes P_1 and P_2 by a superedge iff (1) $P_1 \prec P_2$ and (2) $\nexists$ partial class P s.t. $P_1 \prec P \prec P_2$, which can capture the nested property of k-partial classes. Thus, for a query vertex v_q, the k-TTC is a subtree containing v_q and rooted at supernodes with trussness k.

Fig. 2 (b) shows the EquiTree index for the example graph in Fig. 1, where each supernode represents a partial class. The 3-partial class C_3 consists of 3-truss equivalence classes $C_{3,1}, C_{3,2}$ and $C_{3,3}$, because their edges are 3^+-triangle-connected. Moreover, C_3 links to $C_{4,1}$ and $C_{4,2}$ as it truss-precedes $C_{4,1}$ and $C_{4,2}$. To search for 4-TTCs containing v_6, we start from $C_{4,2}$ containing v_6, which is also the root of the subtree, as the trussness of C_3 is smaller than 4. Thus, we return all the edges contained in subtree $\{C_{4,2}, C_5\}$.

After the preliminary step of truss decomposition [44], EquiTree can be sequentially constructed by examining each edge e in the k-class Φ_k for each k from $k_{\max}$ to 3. Each iteration contains two steps: (1) *Creation*: create supernodes for k-truss equivalence classes and connect them based on k-triangle connectivity; (2) *Refinement*: merge conflicting supernodes that have the same trussness k and are connected to the same child node (neighbor with larger trussness) into one to represent the k-partial class. During *creation*, a supernode x for k-truss equivalence class of e is created and added to the list of edges with smaller trussness in the triangle incident to e; then, x is connected to the root node of the supernodes recorded in the list of e through the Anchored Union-Find (AUF) structure [17]. During *refinement*, the newly connected supernode pairs are added to a buffer B so that the k-truss equivalence classes connected to the same child can be merged into one. Note that the supernode for the k-partial class containing e is merged from the conflicting supernode of k-truss equivalence classes instead of directly identifying all the edges k^+-triangle-connected to e, because the latter will repeatedly access k'-triangles for all $k < k'$, leading to huge time cost.

3 Basic Index Construction

In this section, we will first discuss how to represent the graph and the index by tensors, then present a basic tensor-based algorithm to construct EquiTree, and further discuss the details on how to parallelly create supernodes for k-truss equivalence classes and superedges based on triangle connectivity during the *creation* phase and merge conflicting supernodes to obtain k-partial equivalence classes during the *refinement* phase.

Algorithm 1: Basic Tensor-based Index Construction

Input: A graph $G_T = (eid, E_T(src, dst), ptr)$ with edge trussness Γ
Output: An index $\mathcal{I}_T = (idx\_src, idx\_dst)$ with edge mapping Π

$idx\_src, idx\_dst \leftarrow \emptyset$; $\Pi \leftarrow \texttt{arange}(0, m)$; $anchor \leftarrow \texttt{arange}(0, m)$;
for $k \leftarrow k_{max}$ *to* 3 **do**
  $\Phi_k \leftarrow eid[\Gamma[eid] == k]$;
  $E_s, E_l, E_r \leftarrow ComputeKTriangle(\Phi_k)$;
  $mask_1 \leftarrow \Gamma[E_l] == k$; $mask_2 \leftarrow \Gamma[E_r] == k$;
  $Link(\texttt{cat}(E_s[mask_1], E_s[mask_2]), \texttt{cat}(E_l[mask_1], E_r[mask_2]), \Pi)$;
  $Compress(\Phi_k, \Pi)$;
  $mask_1 \leftarrow (\Gamma[E_l] > k)$; $mask_2 \leftarrow (\Gamma[E_r] > k)$;
  $k\_src \leftarrow \texttt{cat}(\Pi[E_s[mask_1]], \Pi[E_s[mask_2]])$;
  $k\_dst \leftarrow \texttt{cat}(anchor[E_l[mask_1]], anchor[E_r[mask_2]])$;
  $k\_src, k\_dst \leftarrow EdgeUnique(k\_src, k\_dst)$;
  $k\_src, k\_dst \leftarrow MergeNode(\Phi_k, k\_src, k\_dst, \Pi, anchor)$;
  $idx\_src \leftarrow \texttt{cat}(idx\_src, k\_src)$; $idx\_dst \leftarrow \texttt{cat}(idx\_dst, k\_dst)$;
return $(idx\_src, idx\_dst), \Pi$;
Procedure $Link(E_1, E_2, \Pi)$
  $mask \leftarrow E_1 \neq E_2$; $p_1 \leftarrow \Pi[E_1[mask]]$; $p_2 \leftarrow \Pi[E_2[mask]]$;
  while $|p_1| \neq 0$ **do**
    $h \leftarrow \texttt{max}(p_1, p_2)$; $l \leftarrow \texttt{min}(p_1, p_2)$;
    $mask \leftarrow \Pi[h] == h$; $\Pi[h[mask]] \leftarrow l[mask]$;
    $p_1 \leftarrow \Pi[\Pi[h[\neg mask]]]$; $p_2 \leftarrow \Pi[l[\neg mask]]$;
Procedure $Compress(E, \Pi)$
  while $\Pi[E] \neq \Pi[\Pi[E]]$ **do**
    $\Pi[E] \leftarrow \Pi[\Pi[E]]$;
Procedure $EdgeUnique(src, dst)$
  $dst, idx = \texttt{sort}(dst, \downarrow)$; $src = src[idx]$;
  $src, idx = \texttt{sort}(src, \downarrow)$; $dst = dst[idx]$;
  $mask = (\texttt{diff}(src) == 0) \& (\texttt{diff}(dst) == 0)$;
  return $src[\neg mask], dst[\neg mask]$;
Procedure $MergeNode(\Phi_k, src, dst, \Pi, anchor)$
  $anchor[\Phi_k] = \Pi[\Phi_k]$;
  $Link(src, dst, anchor)$; $Compress(\Phi_k, anchor)$;
  $\Pi[\Phi_k] = anchor[\Phi_k]$;
  return $anchor[src], dst$;

3.1 Tensor Representation of Graph and Index

We store the input graph and index based on the Compressed Sparse Row (CSR) and COOrdinate (COO) formats, which have been widely used in GPU-accelerated graph algorithms [6, 15, 29, 45], and provide auxiliary data structures to support efficient index construction and k-TTC search. We store the graph G based on CSR with two tensors: the row pointer ptr and the adjacency array adj, which enables convenient access to all neighbors of any vertex via tensor operations. To facilitate index construction, we also maintain two additional tensors, eid and E_T, where eid is a 1-D tensor to record the edge ID and $E_T = (src, dst)$ is the corresponding 2-D tensor that records the vertex

pairs of each edge. We store the index based on COO to enable fast access to superedges, where *idx_src* and *idx_dst* are sets of source and destination supernodes of superedges. Moreover, we carefully organize *idx_src* and *idx_dst* to support efficient k-TTC search. First, for each superedge, the supernode with lower trussness is stored as the source. Second, the supernodes in *idx_dst* are ordered by descending trussness, and source supernodes with the same destination in *idx_src* are ordered by descending trussness. Additionally, we maintain auxiliary tensors to support efficient search: edge trussness Γ, mapping from edges to supernodes Π, and *iptr* to record the starting positions of supernodes with different trussness in *idx_src*. Fig. 3 shows the data structure for the graph in Fig. 1, where the supernode ID is denoted by the smallest edge ID *eid* in the supernode.

3.2 Overview of Basic Index Construction

Before constructing EquiTree, we perform a preliminary step of truss decomposition [29] to compute the trussness Γ of all edges in the tensorized graph G_T. The basic idea is to compute the support of all edges based on their neighboring vertices, and iteratively (from 2 to $k_{\max}$) peel edges that do not satisfy the support constraint until all the edges have been peeled. [29] also adopts the optimization techniques in [45], such as relabeling vertex IDs and avoiding atomic operations, but with different data structures and computation flows tailored to tensor operators. These optimizations [45] are tailored for k-truss, which is a preliminary step but not a full solution for the k-TTC search problem studied in this paper.

The basic tensor-based index construction framework is shown in Alg. 1, which takes the tensorized graph G_T and the edge trussness Γ as the input, and outputs the index $\mathcal{I}_T$ and the mapping from edges to supernodes Π. At the beginning, we initialize the index $\mathcal{I}_T$ as empty, the edge-supernode mapping Π, and the anchors *anchor* of all edges as themselves (line 1). The tensor *anchor* records the anchor (current root) of each equivalence class during the construction. Then, inspired by the sequential EquiTree construction [47], based on the k-class Φ_k, we iteratively create supernodes and superedges based on k-triangle connectivity during the *creation* phase and merge conflicting supernodes based on the truss-precedence relation during the *refinement* phase for k from $k_{\max}$ to 3. The main difference is that we parallelly create supernodes and superedges based on the k-class Φ_k by tensor operators, instead of examining each edge to create supernodes and superedges one by one in a sequential manner. To achieve this, we first parallelly examine the k-triangle connectivity of all the edges in Φ_k to create supernodes representing k-truss equivalence classes and superedges (lines 3-10), and then parallelly merge conflicting supernodes into one for k-partial classes to obtain the tree structure (lines 11-13). We will elaborate on these two phases in the remaining part of this Section.

3.3 k-class based Supernode/Superedge Creation

We now introduce how to parallelly create supernodes and superedges based on k-class Φ_k during the *creation* phase. We adopt the techniques that transform the identification of k-truss equivalence classes into Connected Component (CC) computation in [19], but with the focus on how to convert this process into efficient tensor operations. Specifically, we first obtain the k-triangles which contain at least one edge in Φ_k, where the set of edges in Φ_k is stored in E_s, and the sets of the other two constituent edges are stored in E_l and E_r. Next, we select edges with trussness k in E_l and E_r, and denote them by boolean tensors $mask_1$ and $mask_2$ (line 5), and *Link* them with edges at the corresponding positions in E_s (line 6). Then, we perform *Compress* on linked edges (line 7). *Link* and *Compress* are two basic operations in parallel CC, where *Link* ensures that the elements at the corresponding positions in the two tensors belong to the same component tree and *Compress* ensures every element directly connects to the root node of the component. After creating all the supernodes for k-truss equivalence classes, we create superedges incident to these

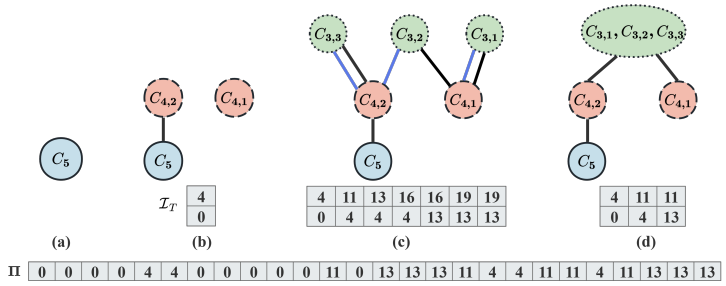


Fig. 4. Basic index construction

supernodes. For each k-triangle, we select edges with trussness larger than k in E_l and E_r (line 8), and create superedges for them, stored in k_src and k_dst temporarily (lines 9-10). Specifically, we create superedges between the root (located through *anchor* that records the current root of each equivalence class) of supernodes containing them and supernodes containing edges in E_s.

Procedures *Link* and *Compress*. We implement tensor-based *Link* (lines 15-20) and *Compress* (lines 21-23) by maintaining tensor Π, which records the parent (current node ID) of each edge. *Link* ensures that any two k-triangle-connected edges with trussness k share the same parent, or updates their parent to the same. Specifically, it first identifies two edge sets p_1 and p_2 that have not been added to the same supernode (line 16). Then, it continuously searches for the roots of these two edges (line 20), compares their labels using `max` and `min` (line 18), and connects the root with the larger label to the other tree (line 19). *Compress* reduces the depth of the component tree by continuously updating the elements in Π, ensuring all edges directly connect to the root.

3.4 Refinement by Layer-based Merging

We now introduce how to identify and merge conflicting supernodes during the *refinement* phase layer by layer for each k from $k_{\max}$ to 3. As stated in [47], at layer k, conflicts occur when multiple supernodes with trussness k connect to the same supernode with trussness larger than k. We first remove all repeated superedges in k_src and k_dst by *EdgeUnique* (line 11), then further merge the conflicting supernodes into one and update *anchor* through *MergeNode* to create the tree-structured index (line 12), and finally concatenate k_src and k_dst into idx_src and idx_dst (line 13).

Procedures *EdgeUnique* and *MergeNode*. The process of *EdgeUnique* is shown in lines 24-28. We first sort the destination *dst* (line 25) and then sort the corresponding source with the same destination in *src* (line 26) by descending trussness. Next, we use the `diff` operator to calculate the differences between adjacent elements for *src* and *dst*, respectively (line 27). If their differences are both 0, indicating that two supernodes are connected multiple times, we eliminate such repetition and return the final edge set (line 28). We use *MergeNode* (lines 29-33) to merge conflicting supernodes. First, we update *anchor* (line 30) and merge the supernodes with $\tau = k$ that belong to the same CC (line 31) through *Link* and *Compress*. Then, we map the merged supernode to Π (line 32) and return the updated superedges (line 33).

3.5 Analysis

We illustrate the process of the above basic index construction algorithm by the example graph in Fig. 1. It begins by traversing the edges in 5-class Φ_5 and creating a supernode C_5 (Fig. 4(a)). Then, we traverse the edges in Φ_4, create two supernodes $C_{4,1}$ and $C_{4,2}$, and connect C_5 to $C_{4,2}$ (recorded as $(0, 4)$ in Fig. 4(b)). Then, we traverse edges in Φ_3, create supernodes $C_{3,1}, C_{3,2}$ and $C_{3,3}$, and connect them to the child nodes $C_{4,1}$ and $C_{4,2}$. Note that the black edges $(C_{3,1}/C_{3,2}, C_{4,1})$ and

$(C_{3,3}, C_{4,2})$ represent that these supernodes are directly triangle-connected while the blue edges $(C_{3,1}, C_{4,1})$ and $(C_{3,2}/C_{3,3}, C_{4,2})$ represent that although $C_{3,1}/C_{3,2}/C_{3,3}$ are triangle-connected to C_5, they are actually connected to its anchor $C_{4,1}/C_{4,2}$. Finally, we merge the conflicting supernodes and finish the construction of EquiTree (Fig. 4(d)).

Complexity Analysis. Suppose that the number of k-triangles in each iteration is t_k. The step complexity of each iteration is $O(\log(t_k))$ and the work complexity is $O(t_k)$. Thus, the overall step complexity and work complexity are $O(\sum_{k=3}^{k_{\max}} \log(t_k))$ and $O(\sum_{k=3}^{k_{\max}} t_k) = O(m^{1.5})$, respectively. The space complexity is $O(m + |\mathcal{I}_T|)$ where $|\mathcal{I}_T|$ is the number of superedges of the index, which is much smaller than m. Thus, the overall space complexity is $O(m)$.

Limitations. From the above analysis, we can see that the basic index construction method needs to iterate from $k_{\max}$ to 3 to check k-triangle connectivity on k-class Φ_k. Such iteration will lead to high step complexity and many tensor operations on small tensors ($(k_{\max} - 3) \times n_t$ where n_t is the number of tensor operations in each iteration), which causes significant overhead of kernel launch and cannot fully leverage the parallelism of the hardware accelerators. Thus, in the following section, we further explore if it is possible to eliminate the iteration on k to boost the parallelism.

4 Advanced Index Construction

In this section, we first discuss how to create all the supernodes/ superedges simultaneously instead of examining each k-class Φ_k, then we discuss how to parallelly merge all the conflicting supernodes to obtain all the partial classes.

4.1 Triangle-centric Supernode/Superedge Creation

Recall that the basic index construction algorithm examines all the edges in each k-class Φ_k to: compute k-triangles, identify k-truss equivalence classes, create supernodes and superedges, and then merge conflicting supernodes to obtain k-partial classes. After a thorough examination of these k-triangles, we find that they exhibit different properties and require different operations to build the index. Consider a k-triangle where the trussness of its three constituent edges e_1, e_2 and e_3 are k_1, k_2 and k_3 ($k = k_1 \leq k_2 \leq k_3$), respectively. Based on their properties and required operations, we classify k-triangles into the following three categories:

Definition 4. (**Internal k-Triangle**) *A k-triangle is an internal k-triangle if the trussness of the three constituent edges are all equal, i.e., $k_1 = k_2 = k_3$.*

Definition 5. (**Marginal k-Triangle**) *A k-triangle is a marginal k-triangle if the trussness of e_1 is equal to the trussness of e_2, i.e., $k_1 = k_2 < k_3$.*

Definition 6. (**External k-Triangle**) *A k-triangle is an external k-triangle if the trussness of e_1 and e_2 are not equal, i.e., $k_1 < k_2 \leq k_3$.*

During the construction of the EquiTree index, these three types of k-triangles play distinct roles due to their unique properties. Internal k-triangles are used to create supernodes for k-truss equivalence classes; external k-triangles are used to create superedges between the supernode containing e_1 and the supernodes containing e_2/e_3. Marginal k-triangles, on the other hand, are transitional states between internal k-triangles and external k-triangles: they not only merge e_1 and e_2 into one supernode but also connect the supernode containing e_1 and e_2 to the supernode containing e_3. We formally illustrate this through the following properties.

Property 1. *The three constituent edges of an internal k-triangle are in the same supernode of EquiTree.*

Proof. From the definition of k-triangle connectivity, it is clear that the three constituent edges in an internal k-triangle are k-triangle-connected and have the same trussness k, and thus belong to the same k-truss equivalence class, i.e., the same supernode. □

Property 2. *For a marginal k-triangle where the trussness of the three constituent edges e_1, e_2 and e_3 are k_1, k_2 and k_3 ($k = k_1 = k_2 < k_3$), e_1 and e_2 belong to the same supernode, and the supernode containing e_1 and e_2 is connected to the supernode containing e_3.*

Proof. Based on the definition of k-triangle connectivity, e_1 and e_2 belong to the same supernode as they have the same trussness k and are k-triangle-connected. This supernode is also connected with the supernode containing e_3 as e_1 and e_2 are also k-triangle-connected with e_3. □

Property 3. *For an external k-triangle where the constituent edges e_1, e_2, and e_3 have trussness k_1, k_2, and k_3 ($k = k_1 < k_2 \leq k_3$), e_1 and e_2/e_3 belong to different supernodes that are connected by a superedge.*

Proof. Clearly, e_1 and e_2/e_3 belong to different supernodes as their trussness is not equal. The supernode containing e_1 and the supernode containing e_2/e_3 are connected by a superedge as e_1 and e_2/e_3 are k-triangle-connected. Note that currently e_2 and e_3 are contained in two different supernodes but not connected by a superedge, because: (a) when $k_2 < k_3$, e_2 and e_3 clearly belong to two different supernodes and they cannot be k_2-triangle-connected by this marginal k-triangle as $k < k_2$; (b) when $k_2 = k_3$, e_2 and e_3 are k-triangle-connected currently rather than k_2-triangle-connected, and therefore belong to different k_2-equivalence class that cannot be k_2-triangle-connected by this marginal triangle as $k_1 < k_2$. Nevertheless, in case (b), the supernode containing e_2 and the supernode containing e_3 might be merged into one in the future if there exists another k_2-triangles sequence that can connect them. □

Although different types of k-triangles have different properties and require different operations in index construction, the processing order of all the k-triangles does not affect the result of index construction, as illustrated in the following lemma.

Lemma 1. *The processing order of k-triangles does not affect the construction of the EquiTree index.*

Proof. During the construction of the EquiTree index, when examining k-triangles, we create supernodes for k-truss equivalence classes, and connect these supernodes with other supernodes based on k-triangle connectivity. First, we prove that the processing order of k-triangles does not affect the creation of supernodes for k-truss equivalence classes. Next, we prove that the processing order of k-triangles does not affect the creation of superedges. The creation of a superedge between supernodes representing k-truss equivalence class and other supernodes only involves external and marginal k-triangles. The processing order of external/marginal k-triangles only affects the order in which superedges are created, but not superedges themselves, as the creation of each edge is independent of the others. Since the processing order of k-triangles does not affect the creation of supernodes and superedges, which is also independent of the subsequent *refinement* phase, it does not affect the construction of the EquiTree index. □

We can further relax the internal/marginal/external k-triangle to the internal/marginal/external triangle by removing the constraint of k, and further prove that the processing of triangles does not need to strictly follow the decrease of k when constructing the EquiTree index.

Theorem 1. *The processing order of triangles does not affect the construction of the EquiTree index.*

Proof. We prove this theorem in two steps. First, we prove that the construction of supernodes for k-truss equivalence classes is not affected by considering the following two cases. (1) e_1 and e_2

belong to the same supernode with trussness k. Then, e_1 and e_2 may be (a) in the same k-triangle $\triangle$ or (b) connected by a sequence of adjacent k-triangles $\triangle_1, \ldots, \triangle_p$ where $e_1 \in \triangle_1$ and $e_2 \in \triangle_p$. For (a), we have $e_1 \overset{k}{\leftrightarrow} e_2$ when $\triangle$ is examined, which means e_1 and e_2 belong to the same supernode. For (b), let $e_{\triangle_i}$ be the common edge between $\triangle_i$ and $\triangle_{i+1}$ $(1 < i < p)$. We have $e_{\triangle_{i-1}} \overset{k}{\leftrightarrow} e_{\triangle_i}$ when $\triangle_i$ is examined ($e_1 \overset{k}{\leftrightarrow} e_{\triangle_1}$, $e_{\triangle_{p-1}} \overset{k}{\leftrightarrow} e_2$ when $\triangle_1$ and $\triangle_p$ are examined, respectively). Since k-triangle connectivity is transitive, we have $e_1 \overset{k}{\leftrightarrow} e_{\triangle_1} \overset{k}{\leftrightarrow} e_{\triangle_2} \ldots \overset{k}{\leftrightarrow} e_{\triangle_{p-1}} \overset{k}{\leftrightarrow} e_2$ after $\triangle_1, \ldots, \triangle_p$ are all examined. (2) e_1 and e_2 belong to two different supernodes. Clearly, there does not exist any k-triangle(s) that makes them k-triangle connected, and thus no matter how the processing sequence of triangles changes, they are still not k-triangle connected.

Then, we prove that the construction of superedges is not affected. As discussed before, two supernodes x_1 with trussness k_1 and x_2 with trussness k_2 ($k_1 < k_2$) are connected by a superedge if there exist two edges $e_1 \in x_1$ and $e_2 \in x_2$ that are contained in a marginal/external k_1-triangle $\triangle_1$. Thus, no matter how the processing sequence of triangles changes, such a superedge will be created as long as $\triangle_1$ is processed. □

The above theorem guarantees that we can process all the triangles in one batch to create supernodes and superedges simultaneously for all the truss equivalence classes. Such triangle-centric supernode and superedge creation can eliminate the iteration on k, thus maximizing the parallelism of new hardware accelerators. Moreover, for large graphs where all the triangles cannot fit in the device memory, we can naturally partition triangles into several batches and sequentially load them from the CPU memory to the device memory to process them batch by batch. The partition of the triangle set and processing order of the batches will not affect the structure of the final index, as guaranteed by Theorem 1.

4.2 Refinement by Batch-based Merging

After creating supernodes for k-truss equivalence classes and connecting them based on k-triangle connectivity in the *creation* phase, we discuss how to efficiently merge conflicting supernodes and adjust the corresponding superedges in the *refinement* phase to obtain the tree-structured EquiTree. Recall that in the basic index construction algorithm, we immediately merge conflicting supernodes according to the truss-precedence relation after creating supernodes and superedges based on Φ_k. Now, we explore whether we can merge all the conflicting supernodes simultaneously instead of layer by layer, since we have already parallelly created all the supernodes and superedges for all the truss equivalence classes. To achieve this, we classify the supernodes into two categories according to their neighboring supernodes as follows.

We mark a supernode as (1) *static* if it has at most one neighboring supernode with smaller trussness (i.e., it has at most one parent supernode indicating no conflicting supernodes in neighbors), and (2) *active* otherwise. Thus, the static supernodes have already satisfied the truss-precedence relation and do not need any adjustment, and we only need to examine active supernodes to merge conflicting supernodes and adjust superedges. To avoid the repeated examination of the same supernode, we start from *maximal active supernodes* that have no active neighbors with larger trussness, denoted as $\mathcal{V}_m$. Suppose that the neighboring supernodes of $x \in \mathcal{V}_m$ are $x_1, x_2, \ldots, x_{|N(x)|}$ sorted in the descending order of trussness. We adjust superedges by creating a path through these neighbors to ensure that they follow the truss-precedence relation, i.e., creating superedges between adjacent supernodes $((x, x_1), (x_1, x_2), \ldots,$ and $(x_{|N(x)|-1}, x_{|N(x)|}))$. If two adjacent supernodes in this path have the same trussness, they are conflicting supernodes and can be merged into one. Such a process can be performed in parallel on $\mathcal{V}_m$, as any $x \in \mathcal{V}_m$ will never be the neighbor of another $x' \in \mathcal{V}_m$ and thus maximal active supernodes do not affect each other.

As the index with tensors *idx_dst* and *idx_src* has already been sorted based on trussness, we can create such a path for all the supernodes in $\mathcal{V}_m$ with a simple adjustment of these two tensors. Suppose that the starting position of a supernode $x \in \mathcal{V}_m$ in *idx_dst* is i, and it has j neighboring supernodes. We only need to copy the elements at $idx\_src[i+1 : i+j-1]$ to $idx\_dst[i : i+j-2]$ to create such a path from x to the neighboring supernodes with the smallest trussness. Then, we check the trussness at the corresponding positions in *idx_dst* and *idx_src* to merge conflicting supernodes. Such an adjustment can be performed in parallel for all supernodes in $\mathcal{V}_m$.

In an ideal case, all the active supernodes may become static after one round of batch-based supernode merging on the maximal active supernodes $\mathcal{V}_m$. However, some static supernodes may become active after such merging. Suppose an active supernode x_1 with trussness k_1 has two neighbors x_2 and x_3 with trussness $k_2 < k_1$, and x_2 and x_3 respectively have only one neighbor with trussness $k_3 < k_2$, which are static. Then, when we examine x_1, merging x_2 and x_3 into one node x' will activate x' as it has two neighboring supernodes with trussness $k_3 < k_2$. Therefore, we need to perform such refinement in multiple batches until all supernodes are static. Next, we will prove the correctness of such a process, and at the same time, demonstrate that such a process can effectively reduce the number of computational rounds compared with the basic layer-based merging in Alg. 1.

Theorem 2. *After one round of refinement on the set of maximal active supernodes $\mathcal{V}_m$, supernodes in $\mathcal{V}_m$ become static and will never be activated again.*

Proof. On the one hand, after the refinement on $\mathcal{V}_m$, supernodes in $\mathcal{V}_m$ have only one neighboring supernode with smaller trussness, i.e., they become static. On the other hand, since each time we only refine *maximal active supernodes*, the neighboring supernodes of each $x \in \mathcal{V}_m$ with larger trussness are always static and will never be refined, thus x will never be activated again. □

Theorem 2 ensures that each round of refinement will reduce the number of supernodes that can be possibly active, and the index eventually converges to a state with no active supernodes, i.e., all supernodes and superedges satisfy the truss-precedence relation.

Now we analyze the maximum iteration number of refinement by batch-based merging. Suppose the number of distinct Φ_k ($\Phi_k \neq \emptyset$) is k^n in G, and the constructed supernodes representing k-truss equivalence classes and their superedges fall into n' connected components $C_1, C_2, \ldots, C_{n'}$ with $k_1^n, k_2^n, \ldots, k_{n'}^n$ distinct trussness respectively. The iteration number R is bounded by $k_{\max}^n = \max\{k_1^n, k_2^n, \ldots, k_{n'}^n\}$, which is no larger than k^n in Alg. 1 as the number of Φ_k in different components is no larger than k^n. In practice, R is smaller than $k_{\max}^n$ in most cases, as many neighboring supernodes of $\mathcal{V}_m$ become static after each refinement, which can be skipped.

4.3 Advanced Index Construction Algorithm

Based on our triangle-centric supernode/superedge creation and batch-based merging during refinement, we now present our advanced index construction method. For simplicity, we assume that the memory capacity of the device (hardware accelerator) is enough to store the graph and compute all the triangles simultaneously in a single batch, as outlined in Alg. 2. For large graphs where the number of triangles exceed device memory, we process only a portion of the graph at a time to handle the corresponding triangles on the device, executing lines 2-8 in Alg. 2 for each portion. For extremely large graphs that cannot fit in device memory, we can iteratively load a subset of vertices and their 2-hop neighbors into the device (inspired by [22]) and execute lines 2-8, and further minimize data transfer overhead based on pipeline strategies in existing CPU-GPU graph processing systems [11, 37]. In Alg. 2, after initializing the mapping tensor Π (line 1), we first compute and examine the triangles to parallelly create supernodes representing k-truss equivalence classes and connect them with superedges during the *creation* phase (lines 2-8), and then merge

Algorithm 2: Advanced Tensor-based Index Construction

Input: A graph $G_T = (eid, E_T(src, dst), ptr)$ with edge trussness Γ
Output: An index $\mathcal{I}_T = (idx\_src, idx\_dst)$ with edge mapping Π

1 $\Pi \leftarrow \texttt{arange}(0, m)$;
2 $E_s, E_l, E_r \leftarrow ComputeTriangle(G_T)$; // Creation phase
3 classify triangles into $(E_s^i, E_l^i, E_r^i), (E_s^m, E_l^m, E_r^m), and(E_s^e, E_l^e, E_r^e)$;
4 $Link(\texttt{cat}(E_s^i, E_s^i, E_s^m), \texttt{cat}(E_l^i, E_r^i, E_l^m), \Pi)$;
5 $Compress(eid, \Pi)$;
6 $idx\_src \leftarrow \texttt{cat}(E_s^e, E_s^e, E_s^m)$; $idx\_dst \leftarrow \texttt{cat}(E_l^i, E_r^e, E_r^m)$;
7 $super\_node \leftarrow \texttt{unique}(\Pi)$;
8 $idx_src, idx_dst \leftarrow$EdgeUnique$(\Pi[src], \Pi[dst])$;
9 $\Gamma_2 \leftarrow \texttt{zeros}(n_{\mathcal{I}_T})$;$\Pi_t \leftarrow \texttt{arange}(0, n_{\mathcal{I}_T})$;$\Gamma_2[\Pi] \leftarrow \Gamma[eid]$; // Refinement phase
10 $chain\_indices \leftarrow \texttt{where}(\texttt{diff}(idx\_dst) == 0)$;
11 **while** $|\ chain\_indices\ | \neq 0$ **do**
12 $idx\_dst[chain\_indices] \leftarrow idx\_src[chain\_indices - 1]$;
13 $mask \leftarrow \Gamma_2[idx\_src] == \Gamma_2[idx\_dst]$;
14 $Link(idx\_src[mask], idx\_dst[mask], \Pi_t)$;
15 $Compress(super\_node, \Pi_t)$;
16 $idx\_src \leftarrow idx\_src[\neg mask]$; $idx\_dst \leftarrow idx\_dst[\neg mask]$;
17 $idx\_src, idx\_dst \leftarrow EdgeUnique(\Pi_t[idx\_src], \Pi_t[idx\_dst])$;
18 $chain\_indices \leftarrow \texttt{where}(\texttt{diff}(idx\_dst) == 0)$;
19 $\Pi \leftarrow \Pi_t[\Pi]$;
20 **return** $\mathcal{I}_T(idx\_src, idx\_dst), \Pi$;

conflicting supernodes and adjust corresponding superedges during the *refinement* phase to obtain the final tree-structured index (lines 9-19).

Triangle-centric Supernode/Superedge Creation. We create supernodes for k-truss equivalence classes and connect them by parallelly examining the triangles based on the three properties in Section 4.1. First, we compute the triangles and store their edges by three tensors E_s, E_l, and E_r, respectively (line 2). Next, we classify these triangles into three categories: internal triangles (E_s^i, E_l^i, E_r^i), marginal triangles (E_s^m, E_l^m, E_r^m), and external triangles (E_s^e, E_l^e, E_r^e) (line 3). Then, based on Properties 1 and 2, we construct supernodes for k-truss equivalence classes (lines 4-5). Specifically, we merge all three edges of each internal triangle, along with the edges in E_s^m and E_l^m of each marginal triangle, into a supernode. Then, we connect supernodes based on Property 3, i.e., connect supernodes containing E_s^e to supernodes containing E_l^e and E_r^e for external triangles, and connect supernodes containing E_s^m to supernodes containing E_r^m (line 6). Finally, we remove duplicate supernodes and superedges (lines 7-8).

Refinement by Batch-based Merging. We merge conflicting supernodes and adjust the corresponding superedges in batch during the *refinement* phase (lines 9-19). We repeat the merging and adjusting process until all the supernodes become static. At the beginning, we use Γ_2 of size $n_{\mathcal{I}_T}$ to record the trussness of each supernode, and use Π_t to merge conflicting supernodes, where $n_{\mathcal{I}_T}$ is the initial number of supernodes (line 9). Since the supernodes in idx_dst are ordered by descending trussness and source supernodes with the same destination in idx_src are ordered by descending trussness, we can apply `diff` on idx_dst to identify all active supernodes, where 0 indicates that the supernode at this position is active and needs to be refined. We adjust the superedges (line 12) and find conflicting supernodes (line 13). For the conflicting supernodes, we

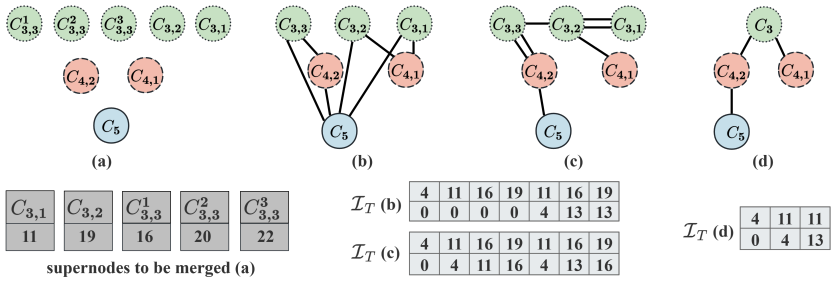


Fig. 5. Advanced index construction

merge them (lines 14-15), update the index (line 16), and delete the repeated connections (line 17). However, superedges have also been adjusted with updated Π_t, which may result in new conflicts. Therefore, the process iterates until *chain_indices* is empty, indicating that no further conflicts exist and all supernodes are static. At this point, we have completed the refinement of the index and obtained supernodes representing partial classes and superedges representing the truss-precedence relation. In the end, we update Π with the map Π_t (line 19).

Fig. 5 illustrates the process of constructing EquiTree using the advanced method. For simplicity, suppose that the internal triangles are processed first and eight supernodes are created (Fig. 5(a)). For marginal triangles where e_1 and e_2 share the same trussness ($\triangle_{3,4,10}$, $\triangle_{4,5,10}$, $\triangle_{0,4,5}$, $\triangle_{0,4,6}$), we merge the supernode containing e_1 with the supernode containing e_2, and create a superedge between the supernode containing e_1 and the supernode containing e_3. For external triangles ($\triangle_{3,4,9}$, $\triangle_{2,3,7}$), we create superedges between the supernode containing e_1 and the supernodes containing e_2 and e_3, respectively (Fig. 5(b)). Next, we perform the supernode-centric refinement process. After examining all triangles, we adjust *idx_src* as previously described (Fig. 5 (c)). Subsequently, we identify supernodes with equal trussness in *idx_src* and *idx_dst* (($C_{3,1}, C_{3,2}$) and ($C_{3,2}, C_{3,3}$)), merge them into one (C_3), update the corresponding superedges (($C_{4,2}, C_{3,3}$) and ($C_{4,1}, C_{3,2}$)), remove duplicate superedges (($C_{4,2}, C_3$)), and finally obtain the EquiTree index (Fig. 5 (d)).

Complexity. The step complexity of triangle-centric supernode/ superedge creation (lines 2-8) is $O(\log(m))$ and the work complexity is $O(m^{1.5})$. Let R denote the number of iterations required for the refinement to converge, which is no larger than $k^n_{\max}$ as analyzed before. For each round of refinement (lines 11-18), the step complexity is $O(\log(m_{\mathcal{I}_T}))$ and the work complexity is $O(m_{\mathcal{I}_T})$, and thus the total step complexity and work complexity are $O(R\log(m_{\mathcal{I}_T}))$ and $O(Rm_{\mathcal{I}_T})$ respectively. Thus, the step complexity of the advanced index construction is $O(\log(m) + R\log(m_{\mathcal{I}_T}))$ and the work complexity is $O(m + Rm_{\mathcal{I}_T})$. The space complexity of Alg. 2 is $O(m + m_{\mathcal{I}_T})$, which is the same as Alg.1.

5 k-TTC Search and Index Maintenance

In this section, we introduce our tensor-based k-TTC search algorithm, along with the index maintenance algorithm.

5.1 k-TTC Search

Our k-TTC search process contains two phases: (1) *extract* the subtree $\mathcal{I}'_T$ induced by the supernodes with trussness no less than k from the index $\mathcal{I}_T$ to reduce search space, and (2) *search* the k-TTCs on the extracted $\mathcal{I}'_T$. Since the *extract* phase can be easily completed by selecting supernodes with trussness $\geq k$ from the index, we discuss how to efficiently accelerate the *search* phase.

Algorithm 3: OPCC based k-TTC Search on EquiTree

Input: A query vertex v_q, trussness k, index $\mathcal{I}_T = (idx\_src, idx\_dst)$ with edge mapping Π, and auxiliary tensors Γ, eid, ptr and $iptr$
Output: A tensor $result$ representing k-TTC containing v_q

1 $result \leftarrow$ `full`$(-1, m)$; $nID, L_{cc} \leftarrow$ `arange`$(0, n_{\mathcal{I}_T})$;
2 $idx\_src, idx\_dst \leftarrow idx\_src[iptr[:k+1]], idx\_dst[iptr[:k+1]]$;
3 $L_{cc}[idx\_dst] \leftarrow idx\_src$; $Compress(nID, L_{cc})$;
4 $E_q \leftarrow eid[\Gamma[eid[ptr[k] : ptr[k+1]]] \geq k]$; $\mathcal{V}_q \leftarrow$ `unique`$(\Pi[E_q])$;
5 $tnodes \leftarrow nID$[`isin`$(L_{cc}, \mathcal{V}_q)$];
6 $mask \leftarrow$ `isin`$(\Pi, tnode)$; $result[mask] \leftarrow nID[\Pi[mask]]$;
7 **return** $result$;

Accelerating the *Search* Process. A naive solution for *search* on EquiTree is first finding all the supernodes containing v_q, then tracing up to the root, and finally searching down from the root to obtain subtrees containing v_q [47]. However, such *two-fold* (TF) search needs to repeatedly access supernodes from supernodes containing v_q to the root and then to the leaves layer by layer, which is time-consuming. Another way is to directly propagate all the root labels (ID of the root node) to leaves, and retrieve all the supernodes that share the same label as the supernode containing v_q. However, such *label propagation* (LP) method also visits the tree layer by layer. Thus, we propose a new search method based on *Optimized Parallel CC* (OPCC) specifically tailored for the tree structure. Note that if we consider the tree as a graph and find the subtrees containing v_q by parallel CC, many expensive connectivity examinations between supernodes will be involved. Thus, we directly update the CC label based on superedges instead of checking connectivity for any supernode pairs. Then, we connect all supernodes to the root by *Compress* to further update the CC label.

The k-TTC search algorithm. Alg. 3 presents the entire process of k-TTC search on EquiTree based on OPCC, where we first *extract* the supernodes with trussness $\geq k$ and the corresponding superedges (line 2) and then *search* for k-TTCs on the extracted supernodes (lines 3-6). It takes the query vertex v_q, trussness k, index $\mathcal{I}_T$, and mapping Π as input, with the assistance of auxiliary tensors (edge trussness Γ, edge ID eid, and row pointers ptr and $iptr$). Similar to the row pointer ptr for the original graph, we use a row pointer $iptr$ to record the starting positions of supernodes with different truss values in idx_src. First, we initialize the data structures (line 1). The tensor $result$ of length m records the community ID, where edges with the same ID belong to the same k-TTC and a value of -1 indicates that the corresponding edge does not belong to any k-TTC; nID records IDs of supernodes; L_{cc} records the root node (CC label) of each supernode, which is initialized as itself. In the *extract* phase, we directly extract the supernodes and superedges with trussness no less than k using $iptr$ (line 2). In the *search* phase, we search for k-TTCs on the extracted supernodes and superedges, i.e., locate the connected components (subtrees) containing v_q (lines 3-5). Specifically, we first compute the connected components by OPCC (line 3), where each connected component is a subtree. Next, we find the set of edges incident to v_q with $\tau \geq k$, denoted as E_q, and supernodes $\mathcal{V}_q$ containing these edges in E_q (line 4). Then, we locate the subtrees containing supernodes in $\mathcal{V}_q$ via `isin`, and obtain their supernodes $tnodes$ (line 5). At last, we find all edges that belong to each subtree and assign a community ID to the corresponding element of $result$ (line 6).

Complexity. We analyze the step and work complexity of k-TTC search in Alg. 3. The step complexity of the *extract* phase is $O(1)$, and its work complexity is $O(n_{\mathcal{V}_k})$ where $n_{\mathcal{V}_k}$ represents the number of supernodes with trussness $\geq k$. The step complexity of the *search* phase is $O(\log(n_{\mathcal{V}_k}) +$

Algorithm 4: Index Maintenance

Input: A graph $G_T = (eid, E_T(src, dst), ptr)$ with edge trussness Γ,
index $\mathcal{I}_T = (idx\_src, idx\_dst)$ with edge mapping Π, and the updated edge set E^*
Output: updated index $\mathcal{I}_T = (idx\_src, idx\_dst)$ and Π

1 identify the affected edges E_A based on E^* ;
2 extract subgraph G_A based on E_A and update the trussness of E_A;
3 $\Psi \leftarrow \texttt{unique}(\Pi[E_A])$; remove Ψ from $\mathcal{I}_T$;
4 $\mathcal{I}'_T \leftarrow$ lines 2-8 in Alg. 2 for G_V; $\mathcal{I}_T \leftarrow \texttt{cat}(\mathcal{I}_T, \mathcal{I}'_T)$;
5 $mask \leftarrow \Gamma[idx\_src] == \Gamma[idx\_dst]$;
6 merge supernodes based on *mask* and update Π;
7 lines 9-19 in Alg. 2 for $\mathcal{I}_T$;
8 **return** $\mathcal{I}_T, \Pi$;

$\log(m_{v_q})$), and its work complexity is $O(m \cdot \log(n_{\mathcal{V}_k}))$ where m_{v_q} represents the number of edges containing v_q. Thus, the overall step complexity is $O(\log(m_{v_q}) + \log(n_{\mathcal{V}_k}))$ and the work complexity is $O(m \cdot \log(n_{\mathcal{V}_k}))$. The space complexity of Alg. 3 is $O(m)$.

5.2 Index Maintenance

Real-world graphs are dynamically changing, requiring the maintenance of indices. To fully leverage the parallelism of the underlying hardware, we propose a tensor-based method to maintain EquiTree for batch edge insertions/deletions. As shown in Alg. 4, it includes the following three steps.

Identify the Scope of *Affected Edges*. Inserting/deleting edges in a graph may create/delete multiple triangles, which causes the increment/decrement of the trussness of their triangle-connected edges. Let E^* be the set of inserted/deleted edges, and $\tau(e)$ and $\hat{\tau}(e)$ be the trussness of edge e before and after E^* insertion/deletion, respectively. We discuss the scope of the *affected edges*, i.e., edges whose trussness may be updated [2], by considering two cases. (1) For an inserted edge $e^* \in E^*$, the upper bound of its trussness is $\overline{\hat{\tau}(e^*)} = \texttt{max}\{k \mid |\triangle_{e^*}^{k-1}| \geq k-2\}$ where $\triangle_{e^*}^{k-1}$ is the set of triangles containing e^* with the trussness of other two constituent edges no less than $k-1$. Thus, $\forall e \in E \cup \{e^*\}$ with $\tau(e) = k < \overline{\hat{\tau}(e^*)}$, if $e \overset{k}{\leftrightarrow} e^*$, e belongs to the *affected edges* and its trussness may be updated as $\hat{\tau}(e) = \tau(e) + 1$. (2) For a deleted edge $e^* \in E^*$, $\forall e \in E \setminus \{e^*\}$ with $\tau(e)(= k) \leq \tau(e^*)$, if $e \overset{k}{\leftrightarrow} e^*$, e belongs to the *affected edges* and its trussness may be updated as $\hat{\tau}(e) = \tau(e) - 1$. We denote the union of all the *affected edges* for edges in E^* as E_A (line 1).

Update Trussness of the Affected Edges. We use V_A to represent the vertex set of E_A, i.e., $V_A = \{u, v \mid (u, v) \in E_A\}$. First, we extract the subgraph G_A induced by V_A. Then, we recompute trussness of V_A and update the trussness of the affected edge E_A (line 2).

Update the Index. Different from existing maintenance method [47] that sequentially updates supernodes with different k values, we propose a parallel maintenance method, which can handle supernodes with different k values simultaneously. First, we find the *affected supernodes*, split them into equivalence classes, identify the affected classes Ψ containing edges in E_A, and delete them from the index (line 3). Then, we reuse lines 2-8 in Alg. 2 to check triangle connectivity between edges in G_A (line 4). Specifically, we first create supernodes (equivalence classes) for *affected edges* E_A in G_A. As boundary edges (edges in G_A except E_A) will not change trussness but preserve the relation of triangle connectivity between edges inside and outside G_A, we construct superedges inside G_A as well as superedges between G_A and $G \setminus G_A$. After that, we merge supernodes with the

Table 1. Statistics of evaluated datasets

Dataset	Vertices	Edges	d_{max}	k_{max}	Triangles
Facebook(FB)	4,039	88,234	1,045	97	1,612,010
DBLP(DB)	317,080	1,049,866	342	114	2,224,385
Youtube(YB)	113,489	2,987,624	28,754	19	3,056,386
Catster(CS)	149,700	5,449,275	80,636	207	185,462,177
LiveJournal(LJ)	3,997,962	34,681,189	14,815	352	177,820,130
Orkut(OK)	3,072,441	117,185,083	33,313	78	627,584,181
Weibo(WB)	58,655,850	261,321,071	278,491	80	212,977,684

same trussness but connected by a superedge (lines 5-6) and in the end perform supernode-centric refinement using lines 9-19 in Alg. 2 (line 7).

6 Experiments

We first introduce the experimental setting, and then evaluate index construction/maintenance and k-TTC search.

6.1 Experimental Setup

Datasets. We conduct experimental studies on seven widely-used real-world networks, ranging from small graphs that can be processed in a single batch to large ones that require multiple batches. These datasets are publicly available in SNAP and Network Repository[1]. Table 1 shows the dataset statistics, where d_{max} is the maximum vertex degree and k_{max} is the maximum edge trussness.

Algorithms. Since the *creation* phase in EquiTree construction is actually constructing EquiTruss, we also derive tensor-based EquiTruss construction algorithms from algorithms in Sections 3 and 4 for comparison. Similarly, we derive the k-TTC search and maintenance algorithms for EquiTruss from algorithms in Section 5. The evaluated algorithms are listed below.

- **SETruss** [2]/**SETree** [47]: Sequential EquiTruss/EquiTree construction/search algorithm based on CPUs.
- **MCETruss** [19]: Parallel EquiTruss construction algorithm based on multi-core CPUs. To the best of our knowledge, there is no existing study on k-TTC on GPUs. Thus, we also implement MCETruss with native CUDA for comparison.
- **TETruss-Basic**/**TETree-Basic** (Alg. 1): Our basic tensor-based index construction algorithm for EquiTruss/EquiTree.
- **TETruss**/**TETree** (Algs. 2/3/4): Our advanced EquiTruss/ EquiTree construction/search/-maintenance algorithm. Our implementation handles the general case where triangles do not fit in device memory, processing them in multiple batches. We set the default batch size as 2×10^7, which is the largest number of triangles that the evaluated GPU can handle in WB.

We obtained the C++ codes of CPU-based algorithms MCETruss, SETruss, and SETree from the authors, and implemented MCETruss on native CUDA. We implemented our algorithms in Python.

Environments. We evaluate the algorithms on two GPU servers. Server 1 is equipped with an NVIDIA A100-SXM4-80GB GPU and an Intel(R) Xeon(R) Platinum 8358P @2.60 GHz CPU with 120 GB of RAM. Server 2 is equipped with an NVIDIA GeForce RTX 3090 Ti with 24 GB of VRAM and Intel Xeon Gold 6142 @2.60 GHz with 60 GB of RAM. We also evaluate our tensor-based algorithms on AMD 7700XT GPU to validate its extensibility across heterogeneous hardware. Both

[1]https://snap.stanford.edu/data/ and https://networkrepository.com/

Table 2. Comparison of index construction time (s)

	EquiTree				EquiTruss					
Platform	GPU		CPU		GPU			CPU		
Dataset	TETree	TETree-Basic	TETree	SETree	TETruss	TETruss-Basic	MCETruss	TETruss	MCETruss	SETruss
FB	**0.086**	1.183	0.967	2.886	**0.030**	0.083	0.217	0.412	1.559	2.607
DB	**0.129**	0.673	1.419	7.151	**0.034**	0.486	0.328	0.621	3.819	4.566
YB	**0.205**	0.808	8.414	37.966	**0.076**	0.533	1.160	2.847	9.299	19.851
CS	**2.175**	119.097	144.225	920.035	**1.899**	111.926	97.128	98.340	299.602	699.600
LJ	**1.787**	32.602	192.559	938.836	**1.322**	24.318	19.379	132.801	321.063	578.267
OK	**11.056**	128.373	1124.504	5964.745	**10.325**	124.537	OOM	618.254	1857.071	5214.326
WB	**12.178**	118.919	3042.903	>2h	**10.611**	110.237	OOM	302.075	2174.243	>2h

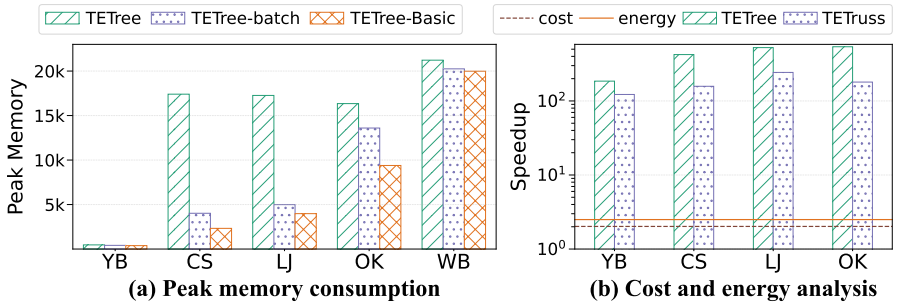


Fig. 6. Cost-performance analysis on index construction

servers share the same software configuration: Ubuntu 22.04, g++ 11.4, Python 3.11, PyTorch 2.2.2, torch-scatter 2.1.2, and CUDA 12.1. Unless otherwise specified, CPU-based algorithms run on server 1 to achieve the best performance, while our tensor-based algorithms are evaluated on the GPU of server 2 to save cost.

6.2 Index Construction/Maintenance

Overall Comparison on Index Construction. Table 2 reports the total construction time of EquiTruss and EquiTree. The performance improvement of TETree on GPUs over the sequential algorithm SETree is 336.26× on average. TETree also significantly outperforms TETree-Basic on GPUs by 16.76× on average, which validates the efficiency of our advanced index construction algorithm. TETruss has similar improvements but requires less construction time than EquiTree, as expected. Besides, TETruss on GPU achieves 154.15× average speedup over the multi-core CPU-based algorithm MCETruss, and 19.59× over our CUDA implementation of MCETruss. Moreover, MCETruss computes and stores all triangles in the GPU memory, which cannot handle OK and WB, while we can efficiently handle them via batch processing. Even on CPUs, TETruss achieves a 2.42× to 7.18× improvement over MCETruss. These results demonstrate that our methods significantly outperform existing sequential and multi-core CPU-based methods.

Cost-Performance Analysis of Index Construction. We analyze the cost-performance of the evaluated algorithms in terms of GPU memory consumption, cost-effectiveness, and energy efficiency. Fig. 6 (a) shows the peak GPU memory consumption of TETree-Basic, TETree with the default batch size (2×10^7), and TETree with a lower batch size of 4×10^6 (denoted as TETree-LB). On small graph YB, TETree consumes slightly higher memory than TETree-Basic, as the number of triangles is small relative to the graph size. On large graphs, TETree consumes significant memory, as a large number of triangles are processed in each batch while TETree-Basic only processes k-triangles in each iteration on k. However, we can significantly reduce the memory consumption

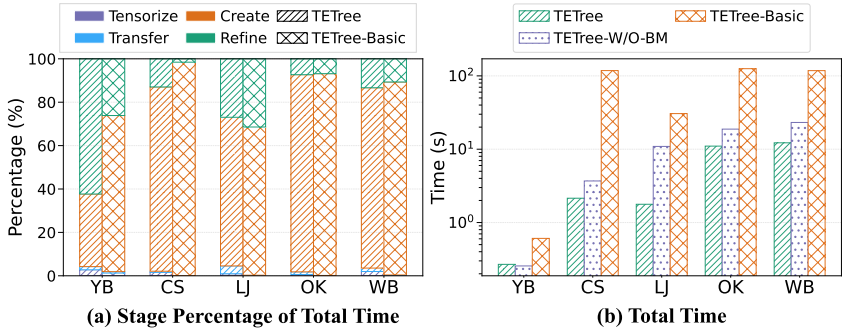


Fig. 7. Performance breakdown and ablation study

Table 3. Comparison of iterations in index construction

		TETree-Basic	TETree		
Dataset	k^n	total	total	creation	refinement
YB	17	17	**7**	1	6
CS	205	1975	**44**	10	34
LJ	315	575	**126**	9	117
OK	76	996	**73**	32	41
WB	78	343	**34**	11	23

by decreasing the batch size, as demonstrated by TETree-LB. For cost-effectiveness and energy efficiency analysis, we use the prices on Azure servers and Thermal Design Power (TDP) from Intel and NVIDIA[2]. The price and TDP of the GPU are approximately 2 and 2.5 times those of the CPU, respectively. We compare our GPU algorithms TETree/TETruss against CPU baselines SETree/MCETruss. As shown in Fig. 6 (b), the bars depict the speedup of TETree/TETruss over SETree/MCETruss, which are significantly higher than the line of price and TDP ratios, validating the outperformance of TETree/TETruss in cost effectiveness and energy efficiency.

Performance Breakdown and Ablation Study on Index Construction. Fig. 7 (a) shows the performance breakdown of our algorithms TETree and TETree-Basic across four stages: *tensorize* to convert the graph from edge list to CSR format, *transfer* to move data from CPU to GPU, *create* to build supernodes and superedges, and *refine* to adjust the index into a tree structure. The tensorization overhead is minimal (1%-3%), and the data transfer overhead is also very small (3% on average). Fig. 7 (b) shows the total time of TETree and TETree-Basic, where the execution time of each stage can be easily obtained together with Fig. 7 (a). Moreover, to evaluate the effect of batch-wise merging, we also report an additional variant of TETree, TETree-W/O-BM, by disabling this strategy. TETree-W/O-BM is consistently surpassed by TETree with speedup up to 6.15×, validating the effectiveness of batch-wise merging.

Detailed Evaluation on Different Stages in Index Construction. To further investigate performance gains of the *creation* and *refinement* during index construction, we report the number of iterations in TETree and TETree-Basic on five representative datasets. Table 3 shows the number of distinct trussness k^n in the graph, the *total* number of iterations, and the number of iterations during *creation*/*refinement*. Compared with TETree-Basic, TETree significantly reduces the iteration number, indicating that our triangle-based creation fully leverages the parallel computation power

[2]https://www.intel.com/ and https://www.nvidia.com/

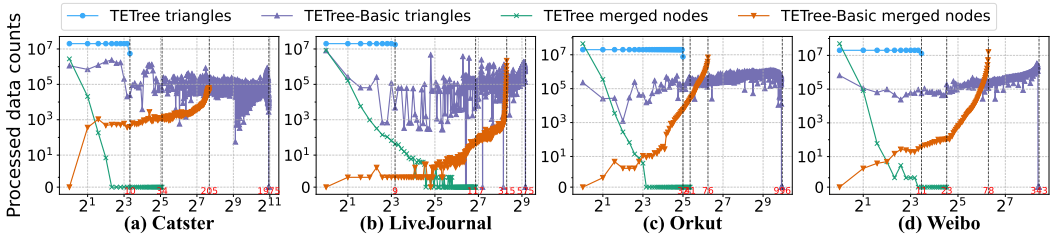


Fig. 8. Comparison of data volume processed in parallel per iteration

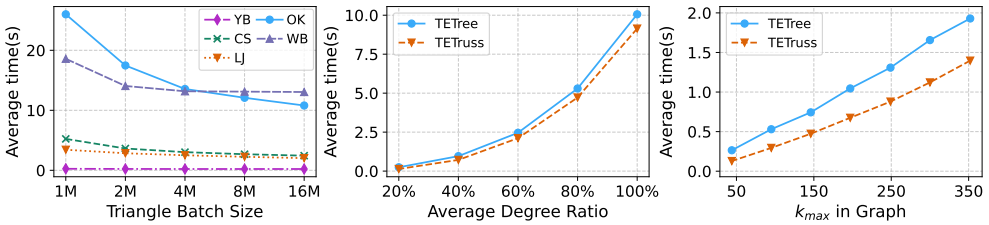


Fig. 9. Effect of parameters on index construction

of TCRs. The *total* iteration number of TETree-Basic is exactly the same as k^n and *creation* only needs one iteration for the small graph YB because all the triangles can be processed in one batch, and they become larger for the latter four large graphs as their triangles need to be partitioned into multiple batches. We further report the number of triangles processed during *creation* and supernodes merged per iteration during *refinement* for the latter four graphs in Fig. 8. TETree-Basic requires more iterations and processes a significantly varying number of triangles in different iterations. In contrast, TETree processes a consistent number of triangles per iteration and completes the task in fewer iterations. During the *refinement* phase, TETree merges almost all conflicting supernodes in the first few iterations and then only adjusts superedges.

Evaluation of Parameters in Index Construction. We evaluate the impact of three key parameters: triangle batch size, vertex degree, and maximal trussness. Fig. 9 (a) shows that as the batch size increases, construction time decreases due to fewer required batches and eventually stabilizes once the batch size is sufficiently large. The improvement is marginal for YB, which has a small number of triangles. To evaluate the impact of degree skew, we select OK with the maximum average degree in the datasets and randomly remove a percentage (from 20% to 80%) of edges. Fig. 9 (b) shows that as the average degree increases, the construction time increases accordingly. To evaluate the impact of $k_{\max}$ on index construction, we select LJ with the maximum $k_{\max}$ in the datasets, and randomly remove edges to obtain datasets with smaller $k_{\max}$. Fig. 9 (c) indicates that as $k_{\max}$ increases, the index construction time also rises. Both Fig. 9 (b) and (c) show that changes in the average degree and $k_{\max}$ of the original graph affect the number of triangles, which directly influences the total time of index construction.

Evaluation on Index Maintenance. We also evaluate the efficiency of the batched index maintenance algorithms for both EquiTree and EquiTruss, and report the maintenance time for insertions and deletions across edge counts ranging from 10^0 to 10^4 in Fig. 10. Compared with reconstructing the index from scratch, our maintenance algorithm demonstrates superior performance, especially for a small number of edge updates.

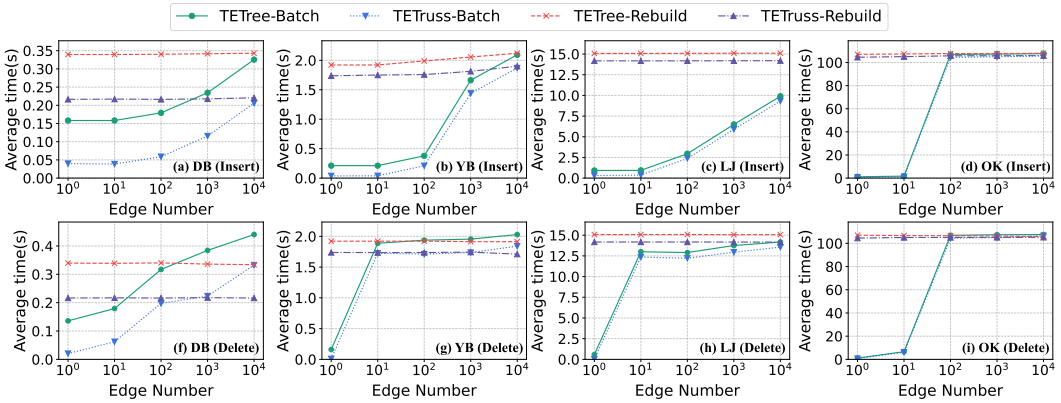


Fig. 10. Evaluation of index maintenance

Table 4. Comparison on different hardware accelerators (s)

	TETree-Basic			TETree (Speedup)			TETruss		
Dataset	A100	3090	7700	A100	3090	7700	A100	3090	7700
FB	**0.962**	1.183	1.625	**0.061** (15.7×)	0.086 (13.7×)	0.159 (10.2×)	**0.020**	0.030	0.108
DB	**0.514**	0.673	0.637	**0.104** (4.9×)	0.129 (5.2×)	0.440 (1.4×)	**0.023**	0.034	0.125
YB	**0.655**	0.808	3.397	**0.172** (3.8×)	0.205 (3.9×)	1.299 (2.6×)	**0.067**	0.076	0.462
CS	**113.077**	119.097	377.534	**1.578** (71.6×)	2.175 (54.7×)	6.905 (54.6×)	**1.363**	1.899	5.759
LJ	**23.025**	32.602	77.242	**1.226** (18.7×)	1.787 (18.2×)	9.770 (7.9×)	**0.867**	1.322	5.575

Extensibility Evaluation on Different Hardware. To evaluate the extensibility of our tensor-based algorithm across heterogeneous hardware, besides two servers equipped with NVIDIA GPUs A100 and 3090 Ti, we also evaluate our algorithms on a desktop computer with an AMD 7700XT GPU. Here, we only report index construction time in Table 4, since the maintenance has a similar trend. Compared with the NVIDIA 3090, the NVIDIA A100 can further accelerate algorithms, which shows that more advanced hardware leads to better performance. The performance on AMD 7700XT is not as significant as that on NVIDIA A100 due to the physical limitations and the inadequate optimization of PyTorch operators on the ROCm backends, but they still significantly outperform CPU algorithms. Overall, we can successfully run TETree-Basic/TETree/TETruss on different hardware backends and achieve considerable performance improvements. Compared with TETree-Basic, TETree gained significant performance improvement on different hardware, demonstrating the effectiveness and extensibility of our optimization strategies across different hardware.

6.3 k-TTC Search

Overall Comparison on k-TTC Search. We evaluate the overall k-TTC search of TETree and TETruss on GPUs with the sequential baseline. From each graph, we randomly select 1000 query vertices and set the default trussness k as 4. Table 5 shows that our tensor-based algorithms TETree and TETruss significantly outperform the baseline algorithms SETree and SETruss, achieving average speedups of 108× and 100×. SETree and SETruss failed to complete truss decomposition and index construction on the WB dataset within 2 hours. Meanwhile, TETree and SETree respectively outperform TETruss and SETruss by 7.64× and 7.44×, validating the superiority of the EquiTree in k-TTC search.

Table 5. Comparison of k-TTC search time (ms).

	EquiTree		EquiTruss	
Dataset	TETree	SETree	TETruss	SETruss
FB	**2.613**	4.905	**11.194**	16.887
DB	**2.076**	8.227	**8.656**	19.207
YB	**2.142**	31.165	**9.155**	255.148
CS	**3.521**	518.617	**45.697**	7155.426
LJ	**5.557**	1900.391	**42.672**	10028.834
OK	**22.143**	3219.456	**208.368**	37261.628
WB	**16.251**	—	**143.721**	—

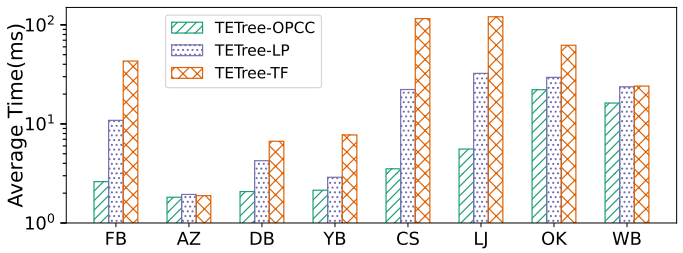


Fig. 11. Comparison of optimization in EquiTree search

Comparison of Different Search Strategies. We evaluate the effect of different k-TTC search strategies on the EquiTree index by comparing the OPCC-based algorithm TETree-OPCC with the LP-based algorithm TETree-LP and the TF-based algorithm TETree-TF on GPUs. Fig. 11 shows that TETree-OPCC achieves the best performance on all evaluated datasets, with an average improvement of 3.21× and 11.73× over TETree-LP and TETree-TF, respectively.

Evaluation on Varying Parameters. We examine the effect of degrees of query vertices and the trussness k on k-TTC search. For degree evaluation, we denote the degree rank of a vertex as 10% if its degree is in the top 10%, and 20% if its degree falls in the top [10%, 20%], and other degree ranks 30%, . . ., 100% are defined similarly. For each rank, we randomly select 1,000 vertices and set parameter $k = 6$. Fig. 12 (a)–(e) show that TETree-OPCC consistently outperforms the other two algorithms. As the rank increases, the degree of v_q decreases, which implies a reduction in the number of edges and supernodes to access. If $\tau(v_q) < 6$ where $\tau(v_q) = \max\{\tau((v_q, u))|(v_q, u) \in E\}$, the search is skipped to reduce search time. The search time on YB increases at rank 100% due to more and larger communities than average. Furthermore, we record the number of skipped queries for each rank, where 167 queries are skipped for every 1,000 vertices in ranks 10%–90% while only 5 queries are skipped at rank 100%. To evaluate the effect of parameter k, we randomly choose 1,000 vertices and run k-TTC search with varying k. Fig. 12 (f)–(j) show that TETree-OPCC performs the best for all k, especially on large graphs. Moreover, with the increase of k, the average search time significantly decreases as the number and the size of the communities decrease.

7 Related Works

Community search [18, 21, 43] aims to retrieve all communities containing a given query vertex from a graph. To meet diverse topology requirements, numerous community models have been proposed, such as k-clique/quasi-clique [8, 12, 42], k-core [26, 38, 39], and k-truss [6, 9, 44]. Among them, k-clique is the most cohesive model but is essentially NP-hard [13, 25, 31]. k-core and k-truss are relaxed models with a different trade-off between cohesiveness and computation cost, i.e., k-core is a subgraph in which each vertex has at least k neighbors, which can be decomposed in

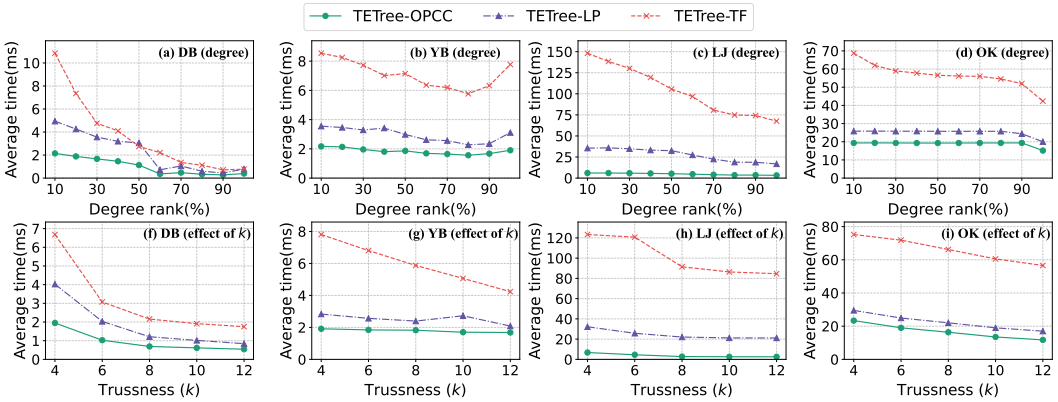


Fig. 12. Comparison of search efficiency with varying parameters

$O(m)$ time [3], while k-truss has a stricter constraint that each edge is contained in at least $k-2$ triangles but needs $O(m^{1.5})$ decomposition time [44]. A k-truss is also a $(k-1)$-core [9]. To find more cohesive communities while maintaining the same time complexity, *triangle connectivity* was further imposed on k-truss [21], i.e., any two edges are connected through a chain of adjacent triangles in a k-truss.

Many indices have been proposed to preserve the trussness and triangle connectivity for efficient triangle-connected k-truss community search. TCP [21] preserves the trussness and triangle adjacency relationship in a compact tree index for each node of the original graph. However, the index structure is large, and the original graph needs to be accessed frequently during community search. EquiTruss [2], a supergraph index based on k-truss equivalence class, stores the information of triangle connectivity and trussness, and the community containing a query vertex can be directly retrieved from the index. EquiTree [47], based on k-partial classes, further compresses the index structure and speeds up the community search. To handle large graphs, [19] proposes the parallel index construction algorithm for EquiTruss based on multi-core CPUs.

Note that many parallel algorithms have been proposed for other community models. [50] proposes parallel k-clique enumeration algorithms on multi-core CPUs. [14, 23] perform k-core decomposition under the multi-core CPU environment with shared memory and [1, 34] investigate the k-core decomposition on GPUs. [24, 40] and [6, 15, 16, 45] respectively investigate k-truss decomposition in multi-core CPUs and GPUs.

8 Conclusion

In this paper, we propose a new tensor-based framework for efficient k-TTC search. To eliminate iteration on k during index construction and fully leverage parallel computing on new hardware, we classify triangles into *internal, marginal,* and *external* triangles based on the trussness of their constituent edges and process them in parallel to accelerate index construction. Meanwhile, we propose a batch-based merging strategy to efficiently refine the index into the tree structure. We also propose tensor-based algorithms to search k-TTC and maintain the index. Experimental results demonstrate that our tensor-based index construction and search algorithms achieve an average speedup of two orders of magnitude, while efficiently maintaining index updates.

Acknowledgments

This work was supported in part by grants of National Natural Science Foundation of China (No. 62272353 and No. 62276193). Junchao Ma and Xin Yan are co-first authors of this work.

References

[1] Akhlaque Ahmad, Lyuheng Yuan, Da Yan, Guimu Guo, Jieyang Chen, and Chengcui Zhang. 2023. Accelerating k-Core Decomposition by a GPU. In *ICDE*. 1818–1831.

[2] Esra Akbas and Peixiang Zhao. 2017. Truss-Based Community Search: A Truss-Equivalence Based Indexing Approach. *PVLDB* 10, 11 (2017), 1298–1309.

[3] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *ArXiv* cs.DS/0310049 (2003).

[4] Tanmoy Chakraborty, Sikhar Patranabis, Pawan Goyal, and Animesh Mukherjee. 2015. On the Formation of Circles in Co-authorship Networks. In *ACM SIGKDD*. 109–118.

[5] Lijun Chang. 2022. Efficient Maximum k-Plex Computation over Large Sparse Graphs. *PVLDB* 16 (2022), 127–139.

[6] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *PVLDB* 13, 10 (2020), 1751–1764.

[7] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, Jeffrey Xu Yu, and Jianxin Li. 2020. Finding Effective Geo-Social Group for Impromptu Activities with Diverse Demands. In *ACM SIGKDD* . 698–708.

[8] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[9] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report* 16 (2008).

[10] Jonathan D. Cohen. 2019. Trusses and Trapezes: Easily-Interpreted Communities in Social Networks.

[11] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-Fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *PVLDB* 17, 6 (2024), 1405–1417.

[12] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online Search of Overlapping Communities. In *SIGMOD*. 277–288.

[13] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. 589–598.

[14] Naga Shailaja Dasari, Desh Ranjan, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. *IEEE Big Data* (2014), 9–16.

[15] Ketan Date, Keven Feng, Rakesh Nagi, Jinjun Xiong, Nam Sung Kim, and Wen-Mei Hwu. 2017. Collaborative (CPU + GPU) Algorithms for Triangle Counting and Truss Decomposition on the Minsky Architecture: Static Graph Challenge: Subgraph Isomorphism. In *HPEC*. 1–7.

[16] Safaa Diab, Mhd Ghaith Olabi, and Izzat El Hajj. 2020. KTRussExPLORER: Exploring the Design Space of K-truss Decomposition Optimizations on GPUs. In *HPEC*. 1–8.

[17] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[18] Yixiang Fang, Xin Huang, Lu Qin, Y. Zhang, W. Zhang, Reynold Cheng, and Xuemin Lin. 2019. A survey of community search over big graphs. *The VLDB Journal* 29 (2019), 353 – 392.

[19] Md Abdul Motaleb Faysal, Maximilian H. Bremer, Cy Chan, John Shalf, and Shaikh Arifuzzaman. 2023. Fast Parallel Index Construction for Efficient K-truss-based Local Community Detection in Large Graphs. *ICPP* (2023), 132–141.

[20] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *PVLDB* 15, 11 (2022), 2811–2825.

[21] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *ACM SIGMOD* . 1311–1322.

[22] Yuli Jiang, Xin Huang, and Hong Cheng. 2021. I/O efficient k-truss community search in massive graphs. *VLDB J.* 30, 5 (2021), 713–738.

[23] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-Core Decomposition on Multicore Platforms. *IPDPSW* (2017), 1482–1491.

[24] Humayun Kabir and Kamesh Madduri. 2017. Shared-Memory Graph Truss Decomposition. In *HiPC*. 13–22.

[25] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *50 Years of Integer Programming*.

[26] Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. 2019. K-Core: Theories and Applications. *Physics Reports* 832 (2019), 1–32.

[27] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: An abstraction for general data processing. *PVLDB* 14, 10 (2021), 1797–1804.

[28] Guanghua Li, Hao Zhang, Xibo Sun, Qiong Luo, and Yuanyuan Zhu. 2024. TenGraph: A Tensor-Based Graph Query Engine. *PVLDB* 17 (2024), 4571–4584.

[29] Guojing Li, Yuanyuan Zhu, Junchao Ma, Ming Zhong, Tieyun Qian, and Jeffrey Xu Yu. 2025. TDT: Tensor Based Directed Truss Decomposition . In *IEEE ICDE*. 1828–1840.

[30] Qiyan Li, Yuanyuan Zhu, Junhao Ye, and Jeffrey Xu Yu. 2023. Skyline Group Queries in Large Road-Social Networks Revisited. *IEEE TKDE* 35, 3 (2023), 3115–3129.
[31] Ronghua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-clique Listing. *PVLDB* 13, 11 (2020), 2536–2548.
[32] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient Community Search with Size Constraint. In *IEEE ICDE*. 97–108.
[33] Fanzhen Liu, Shan Xue, Jia Wu, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Jian Yang, and Philip S. Yu. 2021. Deep learning for community detection: progress, challenges and opportunities. In *IJCAI*. Article 693, 4981-4987 pages.
[34] Amir Mehrafsa, Sean Chester, and Alex Thomo. 2020. Vectorising k-Core Decomposition for GPU Acceleration. *SSDBM* (2020), 1–4.
[35] Mathieu Nassif and Martin P. Robillard. 2023. Identifying Concepts in Software Projects. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3660–3674.
[36] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435 (2005), 814–818.
[37] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys*. ACM, Article 12, 16 pages.
[38] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.* 25, 3 (2016), 425–447.
[39] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287.
[40] Shaden Smith, Xing Liu, Nesreen K. Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. 2017. Truss decomposition on shared-memory parallel systems. In *IEEE, HPEC*. 1–6.
[41] Xing Su, Shan Xue, Fanzhen Liu, Jia Wu, Jian Yang, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Di Jin, Quan Z. Sheng, and Philip S. Yu. 2024. A Comprehensive Survey on Community Detection With Deep Learning. *IEEE TNNLS* 35, 4 (2024), 4682–4702.
[42] Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. 2013. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *ACM SIGKDD*. 104–112.
[43] Chaokun Wang and Junchao Zhu. 2019. Forbidden Nodes Aware Community Search. In *AAAI*, Vol. 33. 758–765.
[44] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *PVLDB* 5, 9 (2012), 812–823.
[45] Runze Wang, Linchen Yu, Qinggang Wang, Jie Xin, and Long Zheng. 2021. Productive High-Performance k-Truss Decomposition on GPU Using Linear Algebra. In *HPEC*. 1–7.
[46] 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* 45, 4 (2013), 1–35.
[47] Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. 2022. Efficient Triangle-Connected Truss Community Search In Dynamic Graphs. *PVLDB* 16, 3 (2022), 519–531.
[48] Zhibang Yang, Xiaoxue Li, Xu Zhang, Wensheng Luo, and Kenli Li. 2022. K-truss community most favorites query based on top-t. *WWWJ* 25, 2 (2022), 949–969.
[49] Junhao Ye, Yuanyuan Zhu, and Lu Chen. 2023. Top-r keyword-based community search in attributed graphs. In *IEEE ICDE*. 1652–1664.
[50] Zhirong Yuan, You Peng, Peng Cheng, Li Han, Xuemin Lin, Lei Chen, and Wenjie Zhang. 2022. Efficient k-clique listing with set intersection speedup. In *ICDE*. 1955–1968.

Received July 2025; revised October 2025; accepted November 2025