TDT: Tensor based Directed Truss Decomposition

Guojing Li*, Yuanyuan Zhu*, Junchao Ma*, Ming Zhong*, Tieyun Qian*, Jeffrey Xu Yu†
*School of Computer Science, Wuhan University, Wuhan, China
†The Chinese University of Hong Kong, Hong Kong, China
*{guojingli, yyzhu, junchaoma, clock, qty}@whu.edu.cn, †yu@se.cuhk.edu.hk

2025 IEEE 41st International Conference on Data Engineering (ICDE) | 979-8-3315-3603-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDE65448.2025.00140

***Abstract*—Truss decomposition is to find the hierarchy of all the k-trusses in a graph for $k \geq 2$. Existing GPU-based algorithms first compute edge support by parallelly counting the number of triangles each edge is contained in, and then iteratively peel off edges with the smallest support and update support of the affected edges in parallel. However, these algorithms perform truss decomposition on undirected graphs, which causes large storage space and numerous triangle existence checks during support update. Moreover, they are developed based on CUDA, which cannot naturally adapt to emerging hardware accelerators and support the end-to-end downstream graph machine learning (ML) tasks. In this paper, we propose a truss decomposition framework based on tensors (TDT), which can leverage the parallelism of heterogeneous hardware backends to speed up the computation and seamlessly integrate with downstream graph ML tasks. We first convert the original input graph into a directed graph and represent it by compacted tensors. Then we perform truss decomposition on the tensorized directed graph by efficient tensor operators. Such a directed-graph storage model not only saves the storage space but also naturally supports efficient support computation/update during the truss decomposition. To further accelerate truss decomposition, we also partition vertex neighbors into blocks to balance the computation workload and optimize key steps such as support computation/update in our framework. Extensive experimental studies show that our Python-based TDT algorithm not only achieves 2.3× - 8.5× speedup in most cases compared with the state-of-the-art CUDA-based algorithms, but also can efficiently deal with large graphs with hundreds of millions of nodes and billions of edges while the baseline fails due to large storage cost. Our source code is publicly available at https://github.com/LiGuojing194/TDTdecomposition.**

***Index Terms*—Truss decomposition, K-truss, Triangle counting, Cohesive subgraphs**

I. Introduction

Graphs are used to represent relationships among entities in a wide range of real-world applications, such as social networks, Web networks, biological networks, etc. Truss decomposition is a fundamental operation to uncover the hierarchy of large complex graphs. It aims to find all the k-trusses for $k \geq 2$, where a k-truss is the largest subgraph where each edge is contained in at least $k-2$ triangles within the subgraph [1]. Fig. 1 gives an example graph where different k-trusses are marked by different gray areas.

Most earlier works on truss decomposition are based on CPUs, including *in-memory* [1], *out-of-cores* [2], [3], and *parallel* [4]–[8] algorithms. The *in-memory* algorithms usually include two phases: *support initialization* to obtain the initial support of each edge (number of triangles containing this edge), and *edge peeling* to iteratively peels edges with the smallest support and update the support of affected edges [1]. The *out-of-core* algorithms mainly focus on loading the candidate subgraph from the disk into the memory to compute k-truss from the smallest/largest possible k [2], or reducing I/O by graph compression [3]. The *parallel* algorithms are either distributed algorithms based on different computation models such as MapReduce [4], bulk synchronous parallel model [9], and asynchronous vertex-centric model [5], or shared-memory algorithms based on different data structures (PKT [6] based on arrays and MSP [7] based on a doubly-linked list) or search strategies ([8] based on local search). These parallel algorithms use different strategies to avoid duplicated triangle enumeration in the support initialization phase, such as degree-based ordering and edge orientation [3], [6], [7], and the ordered pairs of edges enumeration [4], [5]. However, they still involve a large number of triangle existence checking and support update operations in the edge peeling phase.

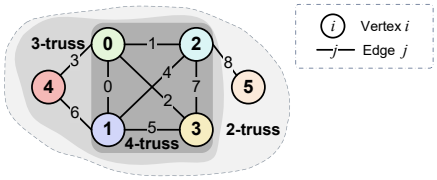


Fig. 1: An example graph with truss hierachy

Recently, *GPU-based* algorithms have been proposed to accelerate truss decomposition. [10] utilizes zero-copy and unified memory to store the adjacency list that can be directly accessed by CPU and GPU threads. [11] optimizes collaborative CPU-GPU algorithms for triangle counting and truss decomposition by short and long updates of the peeled edges. [12] is a fine-grained parallel truss decomposition algorithm based on linear algebra to achieve load balance on both CPU and GPU. OPT-HPU [13] is the most recent CPU-GPU co-processing algorithm to accelerate the bitmap-based triangle counting via word-packing, and support updating by dynamic switch between support recomputation and decrement. During support initialization, it avoids the duplicated triangle enumeration by degree-based vertex ordering. During edge peeling, it uses pivot skip merge, graph compaction, and enumeration skipping to reduce the number of triangle existence checking operations. However, it stores and manipulates the graph as undirected, which causes large storage space. Moreover, the bitmap of each vertex may exceed the limited shared memory

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00140

capacity of a thread, which cannot support efficient truss decomposition on sparse graphs with a large vertex number (e.g., graphs with more than two hundred million vertices such as web-clueweb on Nvidia V100 in the experiments).

Nevertheless, the GPU-based truss decomposition methods were developed based on CUDA, which cannot naturally adapt to other emerging hardware accelerators, such as Tensor Processing Unit (TPU), Neural Processing Unit (NPU), etc. Moreover, although these CUDA codes can be integrated into Python by tools such as Pybind, they cannot support the downstream machine learning (ML) tasks in an end-to-end manner. For example, ML tasks such as link prediction [14]–[16], edge classification [17], and popularity prediction [18] usually take community structure as input to capture coarse-grained information besides node features, where truss can be considered as a typical hierarchy community [1], [19]. Recently, researchers have explored non-ML tasks [20], [21] based on tensors in DL frameworks (Pytorch, TensorFlow, etc.). These frameworks as well as their compilers and runtime (TVM, ONNX, etc.), collectively referred to as *Tensor Computation Runtimes (TCRs)*, provide hardware-independent optimizations such as operator fusion, operator sinking, and algebraic simplification to accelerate tensor computation. Processing non-ML tasks on TCRs can: (1) uniformly leverage the parallel capability of heterogeneous hardware (GPU, TPU, NPU, etc.) for speedup by tensor operators without exploring their specific characteristics and primitives; (2) seamlessly integrate with the downstream ML tasks by sharing the same copy of data and allowing end-to-end tuning of ML models. TQP [20] maps relational queries to tensor operators and shows its out-performance over the state-of-the-art baselines. [21] transforms PageRank (PR) into tensor operators, which outperforms the PR algorithms in existing GPU-based graph systems and implies the possibility of accelerating graph algorithms by tensor operators.

However, mapping truss decomposition into the efficient tensor program is more challenging, as (1) the computation logic of truss decomposition is more complicated compared with PR that can be naturally represented by (sparse) matrix/tensor operations; (2) the workload may be imbalanced when peeling edges and computing/updating support as the number of triangles associated with each edge may vary significantly. Besides, we also need to save storage space as the memory capacity of hardware accelerators is limited compared with the host memory. Thus, in this paper, we propose a truss decomposition framework based on tensors (TDT). It first converts the original input graph into a directed graph and represents it by compacted tensors, and then performs support initialization and edge peeling on the tensorized directed graphs by tensor operators. The edge support is computed based on the intersection operation on the out-neighbors of the two vertices of each edge without duplication. During edge peeling, we identify a small set of active edges to update the affected edges efficiently. Compared with the state-of-the-art GPU-based algorithm OPT-HPU [13] with undirected graph storage and bitmap-based triangle computation, our algorithm not only saves the storage space by more than a half, but also can support computation/update efficiently. Compared with CPU-based algorithms [6], [7] which enumerate triangles in directed graph by edge orientation but still involve unnecessary triangle existence checking and support update operations during edge peeling, we can efficiently and correctly update the support by only processing the active edges. To further improve TDT algorithm, we also partition vertex neighbors into blocks for workload balance, and optimize the key steps such as support computation and support update for speedup. In summary, our contributions are as follows:

- We propose the first tensor-based truss decomposition framework based on compacted directed graph storage and efficient support computation/update, which can leverage the parallelism of heterogeneous hardware backbends for speedup and seamlessly integrate with graph ML tasks.
- We improve our TDT framework by partitioning vertex neighbors to balance computation load and optimizing support computation/update for further acceleration.
- We conduct extensive experimental studies to validate the outperformance of our TDT algorithm, including comparisons with state-of-the-art truss decomposition methods, ablation studies, and scalability testing.

Roadmap. Sec. II introduces the preliminaries. Sec. III gives our TDT framework. Sec. IV presents the optimization strategies. Experimental studies are reported in Sec. V. Sec. VI reviews the related works and Sec. VII concludes the paper.

II. Preliminaries

In this section, we first briefly review the basic concepts and state-of-the-art algorithms for k-truss, and then introduce tensor computation runtimes and the tensor operators.

A. Problem Definition

An undirected graph is denoted as $G = (V, E)$, where V and $E \subseteq V \times V$ are vertex set and edge set, respectively. We use $n = |V|$ and $m = |E|$ to denote the vertex number and edge number, respectively. For a vertex $u \in V$, its neighbor set is defined as $N(u) = \{v \mid \forall\ (u, v) \in E\}$, and its degree is defined as $d(u) = |N(u)|$. A triangle $\triangle_{u,v,w}$ in G is a cycle consisting of three edges (u, v), (u, w), and (v, w). When $G = (V, E)$ is a directed graph, the set of out-neighbors is defined as $N_{out}(u) = \{v \mid \forall\ (u, v) \in E\}$, and the set of in-neighbors is defined as $N_{in}(u) = \{v \mid \forall\ (v, u) \in E\}$. Correspondingly, its out-degree and in-degree are defined as $d_{out}(u) = |N_{out}(u)|$ and $d_{in}(u) = |N_{in}(u)|$, respectively. Based on the above definitions, we give formal definitions for k-truss as follows.

Definition 1 (Support)**.** *The support of an edge $e = (u, v)$ in an undirected graph G is the number of triangles containing e, which is defined as sup$(e, G) = |N(u) \cap N(v)|$. We use sup$(e)$ instead of sup(e, G) when the context is clear.*

Definition 2 (k-TRUSS)**.** *The k-truss of an undirected graph G is the largest subgraph, denoted as T_k, such that the support of each edge in T_k is at least $k-2$, i.e., sup$(e, T_k) \geq k-2$. The trussness (truss number) of an edge $e \in E$ is the maximum k such that e belongs to the k-truss, denoted as $\tau(e)$.*

Definition 3 (k-CLASS)**.** *The k-class of an undirected graph G is the set of edges with trussness k, which is defined as $\Phi_k = \{e \mid e \in E \wedge \tau(e) = k\}$. The union of edges in all the i-class for $i \geq k$ forms a k-truss.*

Problem Statement. The truss decomposition problem is to identify all the k-class Φ_k for $k \geq 2$ in a graph G.

B. Existing GPU-accelerated truss decompositon method

Among existing GPU-accelerated truss decomposition algorithms [10]–[13], the most recent work OPT-HPU [13] is the state-of-the-art. Thus, we briefly review OPT-HPU which includes three main stages as follows.

Pre-processing. The pre-processing stage aims to initialize the data structure of the input graph and the auxiliary indices. The input undirected graph is stored in the Compressed Sparse Row (CSR) format, which consists of row pointers $rptr$ and adjacency arrays adj. This stage generates two additional arrays, eid and E_T, where eid is a one-dimensional array to map neighbors to their corresponding edges ID and E_T records the vertex pairs for each edge, i.e., $E_T = (src, dst)$. Moreover, a support array sup is also initialized with empty values. Fig. 2 shows the data structure for the example graph in Fig. 1, where vertex 0 has a neighbor vertex 2, stored in $adj[1]$, and the edge ID of edge (0, 2) is 1, and thus $adj[1]$ corresponds to $eid[1] = 1$.

Support Initialization. The support initialization phase counts the number of triangles associated with each edge. Specifically, the support of an edge (u, v) is obtained by computing the intersection of $N(u)$ and $N(v)$. Such intersection is computed based on bitmaps. Suppose that $d(v) < d(u)$ in edge (u, v). It constructs a bitmap $B(u)$ of size $|V|$ for $N(u)$. For each $w \in N(v)$, it probes $B(u)$ to check the existence of triangle $\triangle_{u,v,w}$. To reduce the number of probes, it further packs $N(v)$ into non-zero machine words for word-wise lookups on $B(u)$.

Edge Peeling. The edge peeling phase starts on the CPU and transitions to the GPU once the number of edges is reduced to a given threshold. Specifically, edges with support 0 are peeled first; then, starting with edges of support $l = 1$, each iteration estimates the time cost between support recomputation and support decrement to choose a cheaper way for support update. This process repeats until there are no edges with support less than or equal to l in the remaining graph. Then, l is increased by 1 to start the next iteration. It terminates when the number of remaining edges equals the number of edges to peel in the current iteration.

Despite the speedup of truss decomposition, OPT-HPU also has limitations. Besides basic CSR with $rptr$ of size $n+1$ and adj of size $2m$, OPT-HPU also requires auxiliary indices such as eid of size $2m$, E_T of size $2m$, and sup of size m. Such a large data structure may exceed the limited capacity of GPU memory. Furthermore, the bitmap of size n is introduced for a single thread during support computation, which may exceed its limited shared memory capacity.

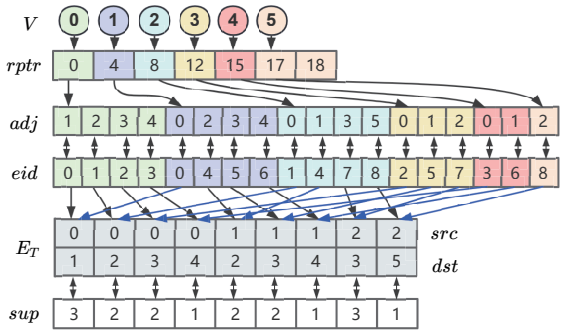


Fig. 2: Data structure for undirected truss decomposition

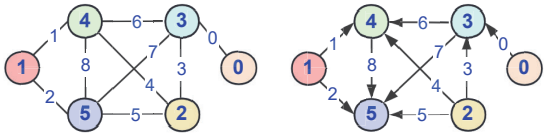
C. Tensor Computation Runtimes

Tensor, abstracted as a multidimensional array, is the basic data structure in the deep learning model. It can be a 0-dimensional scalar, 1-dimensional vector, 2-dimensional matrix, etc. To accelerate tensor computing, hardware manufacturers and cloud service providers have invested resources to develop specialized hardware such as GPUs, TPUs, and NPUs. At the same time, deep learning frameworks such as TensorFlow [22], PyTorch [23], MXNet [24], and their corresponding compilers and runtime systems such as TVM and ONNX runtimes have also emerged, which together form the Tensor Computing Runtime Systems (TCRs) [25], [26].

TCRs provide APIs that enable users to easily represent data by tensors and perform calculations by tensor programs, efficiently executed on heterogeneous hardware backends. The tensor operators provided by TCRs include: (1) *Initialization*: create a tensor, e.g., all 0/1 tensor (`ones`, `zeros`), empty tensor (`empty`), ordered tensor (`range`), etc.; (2) *Selection*: select elements from a tensor by numerical or boolean index (`tensor[indices]`, `tensor[mask]`); (3) *Comparison*: compare tensors to generate a boolean result (`eq`, `lt`, `gt`, `le`, `ge`, etc.); (4) *Arithmetic*: perform basic arithmetic operations on tensors, e.g., basic operators ($+$, $-$, $\times$, $\div$), in-place operators (`add_`, `sub_`), etc.; (5) *Reorganization*: rearrange elements in a tensor or concatenate multiple tensors (`stack`, `reshape`, `sort`, `cat`, etc.); (6) *Reduction*: aggregate over a tensor or get unique values from a tensor (`max`, `min`, `sum`, `mean`, `unique`, etc.).

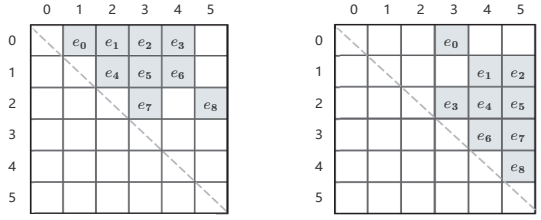
III. TENSOR BASED DIRECTED TRUSS DECOMPOSITION

In this section, we propose a tensor-based truss decomposition method, consisting of (1) a pre-processing phase to perform degree-based graph orientation and represent the directed graph as tensors; (2) a directed truss decomposition framework based on tensors (Algorithm 1); (3) key functions used in the framework (Algorithm 2).



(a) Ordered graph (b) Directed ordered graph

Fig. 3: Degree based graph orientation



(a) Matrix without degree order (b) Matrix with degree order

Fig. 4: Adjacent matrices with/without degree order

A. Pre-processing

In the pre-processing phase, we convert the undirected graph into a directed graph and represent it by compact tensors to save storage space and accelerate the computation.

Degree based graph orientation. During truss decomposition, the key step is to compute/update edge support. OPT-HPU [13] stores the input graph as undirected and avoids the duplicated triangle enumeration by vertex ordering. However, storing graph as undirected will cost more space. To save storage space, avoid duplicated enumeration, and balance the workload, degree-based ordering [27]–[29] and graph orientation [6], [7], [30] are used to convert an undirected graph to a directed one for triangle enumeration and truss decomposition. In this paper, we first sort the vertices of the undirected graph by their degrees, and then renumber the vertices based on the ascending order of the degree. Then, we gradually assign each edge an ID based on its two vertex IDs from smaller to larger. Finally, we convert each undirected edge into directed one from the starting vertex with smaller ID to the ending vertex with larger ID. Note that although we use a similar graph orientation strategy as previous truss decomposition methods [6], [7], [30], during edge peeling, we can optimize edge support update by only activating a small subset from the tensor of affected edges (see Subsec. III-B), while previous methods still involve unnecessary triangle existence checking and support update operations [13].

Fig. 3 illustrates such a graph orientation process for the example graph in Fig. 1. After renumbering the vertices and edges, we obtain the vertex-ordered graph in Fig. 3(a), and after setting the edge direction, we obtain the directed ordered graph in Fig. 3(b). Fig. 4 further gives the corresponding matrices for the directed version of the graph in Fig. 1 without degree ordering and the directed graph in Fig. 3(b) with degree ordering. The out-degrees of vertices in Fig. 4(a) vary greatly between 0 and 4, while out-degrees of vertices in Fig. 4(b) are mostly 2. Thus, the degree ordering makes the number of out-neighbors on each vertex more balanced,

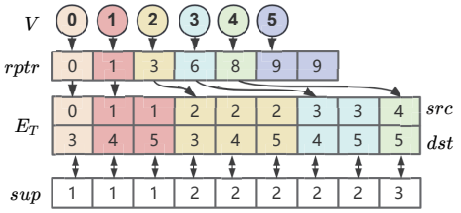


Fig. 5: Tensorized data structure of the directed ordered graph

which will speed up the parallelization of decomposition.

Compacted graph tensorization. During the edge support computation and edge peeling, we need to locate the other two edges that can form a triangle with the target edge, and the graph storage model needs to support such operation efficiently. The previous work OPT-HPU [13] stores the input graph as an undirected graph consisting of two arrays of size $n+1$ and m to record the pointer and the adjacent neighbors respectively, three arrays of size $2m$ to maintain auxiliary indices, and a dynamically constructed bitmap of size n per vertex for edge support computation, which consumes a large amount of storage space and maintenance time. In this paper, we decompose trusses based on the directed graph, where we only perform intersection on the out-neighbors of the ending vertices for an edge to obtain edge support. Such a neighbor-intersection-based method costs much less storage space ($O(|N_{out}(v)|)$ space for vertex v) compared to the bitmap-based method ($O(n)$ space per vertex) for efficient parallel edge support computation based on tensors. Moreover, in the directed ordered graph, the indices of out-neighbors adj are the same as the indices of destination vertices dst. This eliminates the need for additional mappings between out-neighbors and edges, as well as the need for the adj and eid arrays. Thus, we can store the directed graph more compactly using only four tensors, which are: src, dst, sup of size m to store the starting vertices, ending vertices, and the support value of edges in E_T; $rptr$ of size $n+1$ to store the starting/ending position of the out-neighbors of vertices in the dst array, i.e., the out-neighbors of vertex u are $dst[rptr[u] : rptr[u+1]]$. Fig. 5 shows the storage in the tensor format for the directed ordered graph in Fig. 3(b). The out-neighbors of vertex 1 are $dst[1:3]$, i.e., vertices 4 and 5. Correspondingly, the outgoing edges of vertex 1 are $E_T[:, 1:3]$, and their corresponding support values are all 1.

Our compacted tensorized representation of directed graphs reduces storage space by more than a half compared with the undirected graph [13], and enables fast direct edge location via out-neighbor vertices to accelerate truss composition.

B. Tensor based Directed Truss Decomposition

We first discuss how to correctly identify affected edges when peeling edges, and then present the TDT framework.

1) Identification of affected edges. During truss decomposition, when peeling edges, we need to find the edges affected by these peeling edges and update their support. In fact, when peeling (u, v), only edges that can form a triangle with

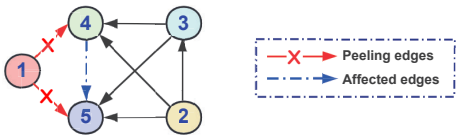


Fig. 6: The example of peeling edges and affected edges

(u, v) can be affected, which are called *affected edges* [10]–[12]. For an undirected graph, finding such affected edges is equivalent to finding the intersection of $N(u)$ and $N(v)$ in the remaining graph. For a directed graph, we can also find the affected edges similarly, i.e., compute the intersection of $N_{in}(u) \cup N_{out}(u)$ and $N_{in}(v) \cup N_{out}(v)$. However, the number of in- and out-neighbors may be large and thus cost a lot of computation time. Moreover, if two edges of the same triangle are peeled, the support of the third edge may be decreased multiple times. Thus, we propose a new scheme, which only activates the neighboring edges of the source vertex of a peeling edge so that the support of affected edges can be successfully and efficiently updated only once. We call such edges *active edges*. Based on active edges, we can identify all the affected edges by the following theorems.

Theorem 1. *When peeling an edge (u, v) in a directed graph G, the outgoing edges of its active edges include all the edges of triangles containing (u, v).*

Proof. Suppose that the peeling edge (u, v) is contained in a triangle $\triangle_{u,v,w}$. Since $u < v$, the ordering of vertices w, u, and v falls into one of the following three cases:

- $u < v < w$. (u, v) and (u, w) are active edges. The outgoing edges of (u, v) include edges of $\triangle_{u,v,w}$.
- $u < w < v$. (u, v) and (u, w) are active edges. The outgoing edges of (u, w) include edges of $\triangle_{u,v,w}$.
- $w < u < v$. (u, v) and (w, u) are active edges. The outgoing edges of (w, u) include edges of $\triangle_{u,v,w}$.

Therefore, the outgoing edges of active edges include all edges of triangles containing (u, v). □

Fig. 6 illustrates the relationship between peeling edges and affected edges. When peeling $(1, 4)$, the active edges are $(1, 4)$ and $(1, 5)$, and the outgoing edges of $(1, 4)$ include the edges of $\triangle_{1,4,5}$ where $(4, 5)$ is the affected edge. Similarly, when peeling $(1, 5)$, $(4, 5)$ is also the affected edge.

The above theorem ensures that all the triangles containing the peeling edges can be identified when we only process the out-neighbors of active edges. Next, we show that no matter how many edges are peeled in a triangle, the support of affected edge in the triangle will be only decreased by 1.

Theorem 2. *A triangle $\triangle_{u,v,w}$ in a directed ordered graph G can be identified only when the edge with the smallest ID in $\triangle_{u,v,w}$ is an active edge.*

Proof. Suppose that $u < v < w$. We consider the following cases. (1) (u, v)/(u, w) is peeled. (u, v) and (u, w) are active edges. For (u, v), w is in the intersection of $N_{out}(u)$ and $N_{out}(v)$, i.e. $\triangle_{u,v,w}$ is identified; for (u, w), v is not in the intersection of $N_{out}(u)$ and $N_{out}(w)$ for (u, w) as (v) is not the out-neighbor of w, i.e., $\triangle_{u,v,w}$ is not identified. (2) (v, w) is peeled. (u, v) and (v, w) are active edges. Similarly, for (u, v), $\triangle_{u,v,w}$ is identified; for (v, w), u is not in the intersection of $N_{out}(v)$ and $N_{out}(w)$, i.e. $\triangle_{u,v,w}$ is not identified. Thus, no matter which edge in $\triangle_{u,v,w}$ is peeled, (u, v) with the smallest edge IDs will always be the active, and $\triangle_{u,v,w}$ will be identified. □

As shown in Fig. 6, when peeling $(1, 4)$ or $(1, 5)$, $(1, 4)$ and $(1, 5)$ are active edges. For active edge $(1, 4)$, $\triangle_{1,4,5}$ is identified; for active edge $(1, 5)$, $\triangle_{1,4,5}$ is not identified. Thus, $\triangle_{1,4,5}$ is identified only once when edge $(1, 4)$ is active.

Corollary 1. *When two edges in $\triangle_{u,v,w}$ are peeled at the same time in a directed ordered graph, $\triangle_{u,v,w}$ will only be identified once and the support of the unpeeled edge in $\triangle_{u,v,w}$ is only decreased by 1.*

The above corollary follows directly from Theorem 2, with proof omitted for brevity. As shown in Fig. 6, when $(1, 4)$ and $(1, 5)$ are peeled, $(1, 4)$ and $(1, 5)$ are active edges. $\triangle_{1,4,5}$ is identified only once when $(1, 4)$ is peeled, and the support of $(4, 5)$ is decreased by 1 only once.

2) The TDT framework. The TDT framework is shown in Algorithm 1, which aims to identify 2- to $k_{\max}$-trusses on the tensorized directed graph $G_T = (E_T(src, dst), rptr)$. The output consists of edges E'_T (sorted by ascending trussness) and trussness indices $tptr$, where the k-class are edges of $E'_T[tptr[k-2] : tptr[k-1]]$ and k-truss are edges in E'_T from $tptr[k-2]$ to the end. First, we initialize the peeled edges E'_T and trussness indices $tptr$ as empty tensors (Line 1). Next, we compute the support of all the edges, sup, by function *ComputeSup* (details of this and the following functions will be illustrated in Subsection III.C.) (Line 2). Then we directly peel edges that are not contained in any triangles and update the corresponding trussness index $tptr$ (Lines 3-4). To reduce the cost of frequent edge deleting, we mark peeled edges and only perform the actual deletion once the number of marked edges reaches a certain threshold. Specifically, we use M_{del} to mark the peeled but not yet actually deleted edges and set their destination vertices dst as -1. Then, starting from $l = 1$, we iteratively find the indices of edges with support l, denoted as I_{cur} (Line 5). If I_{cur} is empty, we add the starting position of edges with $l+2$ trussness in E'_T to $tptr$, increase l by 1 to find the new indices of edges with support l, and jump to the next loop (Lines 7-12). If I_{cur} is not empty, we first identify the active edges, then update the affected edges, and finally find the next I_{cur} to peel (Lines 13-16). Next, we elaborate on how to peel non-empty edges in each iteration, reduce the size of the graph, and stop the termination early.

Edge Peeling. For the current non-empty peeling edge set $E_T[I_{cur}]$, we first identify the indices of the active edges that share a common vertex with vertices in $src[I_{cur}]$ using function *FindActiveE* (Line 13). According to the above theorems, the support of edges in a triangle containing the peeling edges will be updated only once. Then, we mark the

Algorithm 1: TDT

Input: A tensorized directed graph $G_T = (E_T(src, dst), rptr)$
Output: Edges E'_T and its truss index $tptr$ in ascending order

1 $E'_T \leftarrow \texttt{empty}(0)$; $tptr \leftarrow \texttt{empty}(0)$;
2 $sup \leftarrow$ *ComputeSup*(G_T);
3 $M_{del} \leftarrow sup == 0$; $dst[M_{del}] \leftarrow -1$;
4 $tptr \leftarrow \texttt{cat}(tptr, |E'_T| + \texttt{sum}(M_{del}))$;
5 $l \leftarrow 1$; $I_{cur} \leftarrow \texttt{where}(sup == l)$;
6 **while** $sum(M_{del}) + |I_{cur}| < |E_T|$ **do**
7 **if** $|I_{cur}| == 0$ **then**
8 $tptr \leftarrow \texttt{cat}(tptr, |E'_T| + \texttt{sum}(M_{del}))$;
9 **if** $sum(M_{del}) > \delta$ **then**
10 $E'_T \leftarrow \texttt{cat}([E'_T, E_T[M_{del}]])$;
11 $G_T, sup, M_{del} \leftarrow$ *ReduceG*(G_T, sup, M_{del});
12 $l \leftarrow l + 1$; $I_{cur} \leftarrow \texttt{where}(sup == l)$; continue;
13 $I_{act} \leftarrow$ *FindActiveE*(G_T, I_{cur});
14 $M_{del}[I_{cur}] \leftarrow true$;
15 $M_{next} \leftarrow$ *UpdateSup*$(G_T, I_{act}, M_{del}, l, sup)$;
16 $dst[I_{cur}] \leftarrow -1$; $I_{cur} \leftarrow \texttt{where}(M_{next})$;
17 $E'_T \leftarrow \texttt{cat}([E'_T, E_T[M_{del}]])$; $tptr \leftarrow \texttt{cat}(tptr, |E'_T|)$;
18 **return** $E'_T, tptr$;

edges indexed by I_{cur} as `true` in M_{del} before updating the support (Line 14). This ensures that M_{del} serves not only as a mask during graph reduction but also is used to determine whether a triangle contains peeling edges later. In function *UpdateSup*, we process the active edges in batch, search the peeling triangles that contain at least one edge from $E_T[I_{cur}]$ by excluding any deleted edges, and mark the edges to peel in the next round by M_{next} (Line 15). Finally, we mark the destination vertices of the peeled edges as -1 and obtain the new I_{cur} from M_{next} to step into the next loop (Line 16).

Graph Reduction. If the number of peeled edges $\texttt{sum}(M_{del})$ exceeds the threshold δ, we append the edges indexed by M_{del} to E'_T, and obtain a smaller G_T as well as the corresponding sup and M_{del} by *ReduceG* (Lines 9-11).

Early Termination. Generally, when M_{del} are all true, we can terminate the iteration. To avoid the peeling process of $E_T[I_{cur}]$ in the final round, we stop the iteration early when the sum of the number of peeled and peeling edges equals the total number of edges in the current graph (Line 6), and immediately output updated E'_T and $tptr$ (Line 17).

C. Key Functions Implementation based on Tensors

Now, we discuss how to implement the four functions used in the TDT framework (Algorithm 2).

The function *ReduceG* (Lines 1-6) reduces the graph size by deleting the peeled edges to accelerate truss decomposition. First, we extract the remaining subgraph using the mask M_{del}, and update E_T as well as sup through boolean indexing. Then, we count *false* elements in each segment of M_{del} (indexed by $rptr$) using `segment_csr`, storing out-neighbor counts in $sizes$. Subsequently, the cumulative sum of $sizes$ is computed by `cumsum`, and then a zero is prepended to it by the `cat` operator. Finally, M_{del} is reset to an all-false boolean tensor of the same length as E_T.

Algorithm 2: Key Functions in the TDT Framework

1 **Function** *ReduceG*(G_T, sup, M_{del})
2 $E_T \leftarrow E_T[\neg M_{del}]$; $sup \leftarrow sup[\neg M_{del}]$;
3 $sizes \leftarrow \texttt{segment\_csr}(\texttt{int}(\neg M_{del}), rptr)$;
4 $rptr \leftarrow \texttt{cat}([\texttt{zeros}(1), sizes.\texttt{cumsum}(0)])$;
5 $M_{del} \leftarrow \texttt{full}(|E_T|, false)$;
6 **return** G_T, sup, M_{del};
7 **Function** *FindActiveE*(G_T, I_{cur})
8 $P \leftarrow \texttt{unique}(src[I_{cur}])$;
9 $M_{ver} \leftarrow \texttt{full}(|V|, false)$; $M_{ver}[P] \leftarrow true$;
10 $M_{act} \leftarrow M_{ver}[dst]$;
11 $I_{dst}, ptr \leftarrow$ *batch_csr_select*$(rptr[P], rptr[P+1])$;
12 $M_{act}[I_{dst}] \leftarrow dst[I_{dst}] > 0$; $I_{act} \leftarrow \texttt{where}(M_{act})$;
13 **return** I_{act};
14 **Function** *ComputeSup*(G_T)
15 $I_u, uptr \leftarrow$ *batch_csr_select*$(rptr[src], rptr[src+1])$;
16 $I_v, vptr \leftarrow$ *batch_csr_select*$(rptr[dst], rptr[dst+1])$;
17 $M_u, M_v \leftarrow \texttt{segment\_isin}(dst[I_u], dst[I_v], uptr, vptr)$;
18 $sup \leftarrow \texttt{segment\_add}(\texttt{int}(M_u), uptr)$;
19 I_{uw}, $ct \leftarrow \texttt{unique}(I_u[M_u])$; $sup[I_{uw}] \leftarrow sup[I_{uw}] + ct$;
20 I_{vw}, $ct \leftarrow \texttt{unique}(I_v[M_v])$; $sup[I_{vw}] \leftarrow sup[I_{vw}] + ct$;
21 **return** sup;
22 **Function** *UpdateSup*$(G_T, I_{act}, M_{del}, l, sup)$
23 $U', V' \leftarrow src[I_{act}], dst[I_{act}]$;
24 $I_u, uptr \leftarrow$ *batch_csr_select*$(rptr[U'], rptr[U'+1])$;
25 $I_v, vptr \leftarrow$ *batch_csr_select*$(rptr[V'], rptr[V'+1])$;
26 $M_u, M_v \leftarrow \texttt{segment\_isin}(dst[I_u], dst[I_v], uptr, vptr)$;
27 $sizes \leftarrow \texttt{segment\_add}(\texttt{int}(M_u), uptr)$;
28 $I_{uv} \leftarrow \texttt{repeat\_interleave}(I_{act}, sizes)$;
29 $I_{uw} \leftarrow I_u[M_u]$; $I_{vw} \leftarrow I_v[M_v]$;
30 $\mathbf{M}_{tri} \leftarrow (dst[I_{uw}] > 0) \wedge (M_{del}[I_{uv}] \vee M_{del}[I_{uw}] \vee M_{del}[I_{vw}])$;
31 $I_e \leftarrow \texttt{cat}([I_{uv}[M_{tri}], I_{uw}[M_{tri}], I_{vw}[M_{tri}])$;
32 $I_{ue}, ct \leftarrow \texttt{unique}(I_e)$; $sup[I_{ue}] \leftarrow sup[I_{ue}] - ct$;
33 $M_{next} \leftarrow \texttt{full}(sup.\texttt{shape}, false)$;
34 $M_{next}[I_{ue}] \leftarrow (\neg M_{del}[I_{ue}]) \wedge (sup[I_{ue}] <= l)$;
35 **return** M_{next};
36 **Procedure** *batch_csr_select*$(starts, ends)$
37 $sizes \leftarrow ends - starts$;
38 $ptr \leftarrow \texttt{cat}([\texttt{zeros}(1), sizes.\texttt{cumsum}(0)])$;
39 $I \leftarrow \texttt{arange}(ptr[-1]) +$
 $(starts - ptr[:-1]).\texttt{repeat\_interleave}(sizes)$;
40 **return** I, ptr;

The function *FindActiveE* (Lines 7-13) effectively identifies the active edges that need to be processed in updating the support. Specifically, we first extract the unique source vertices of the peeling edges by `unique`, denoted as P. Then, we create a vertex mask M_{ver} and set $M_{ver}[P]$ to *true*. Since elements in dst for peeled edges are set to -1 and the last element of M_{ver} is false, $M_{ver}[dst]$ acts as a mask to filter the incoming edges of P, excluding deleted edges. Next, we use the procedure *batch_csr_select* to obtain the indices of the outgoing edges for P. Finally, we filter out deleted edges using $dst[I_{dst}] > 0$, mark the outgoing and incoming edges of vertices P in M_{act}, and obtain the indices of the active edges by the `where` operator.

The function *ComputeSup* (Lines 14-21) computes the support of each edge. We first use *batch_csr_select* to ex-

tract the out-neighbor indices I_u, I_v and their corresponding pointers $uptr, vptr$ for source and destination vertices, respectively. The `segment_isin` operation identifies common out-neighbors within each segment of the two input tensors $dst[I_u]$, $dst[I_v]$, producing masks M_u and M_v, where elements corresponding to common out-neighbors are marked as *true*. For each identified $\triangle_{u,v,w}$, we update sup using `segment_add` to compute the number of triangles formed on edge (u, v). Then, we further increase sup by counting the number of each unique neighboring edge for (u, w) and (v, w) using `unique`. Finally, the updated sup is returned.

The function *UpdateSup* (Lines 22–35) updates the support of the affected edges and marks the next peeling edges by processing only the active edges. It first retrieves the source and destination vertices of the active edges, denoted as U' and V', respectively, and then performs out-neighbor selection where the indices of out-neighbors (I_u for U' and I_v for V') and their corresponding pointers ($uptr$ and $vptr$) are obtained in batch via *batch_csr_select* (Lines 23-25). Next, the boolean masks M_u and M_v are generated using `segment_isin`, which indicate the segmented common out-neighbors of U' and V' in $dst[I_u]$ and $dst[I_v]$, respectively. Subsequently, we store the indices of the edges forming all triangles containing the active edges and the common neighbors in I_{uv}, I_{uw}, and I_{vw}. Using `segment_add`, the number of common out-neighbors is counted for each edge. This count is then used with `repeat_interleave` to generate I_{uv} (Lines 26-29). Since the indices of neighbors also represent the indices of neighboring edges, a mask M_{tri} is created to identify valid triangles whose edges have not been marked as deleted and that include at least one peeling edge (Line 30). Finally, the indices of edges forming valid triangles are concatenated, and the unique edges I_{ue} along with their counts ct are obtained using `unique`. The support values are updated by subtracting ct, and M_{next} is returned to identify edges whose updated support values are no longer larger than l (Lines 31-34).

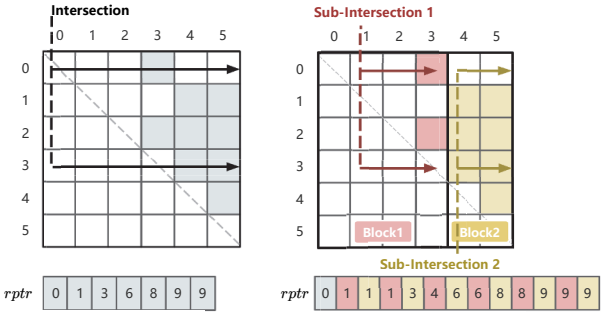
The procedure *batch_csr_select* (Lines 36-40) returns the indices of elements within multiple segmented intervals and segment pointers. It is used to batchly retrieve the neighbor indices of multiple vertices and concatenate them into a tensor. Specifically, we first calculate the size of each interval, then use `cat` and `cumsum` to generate the cumulative segment pointer, and finally use `arange` and `repeat_interleave` to generate the element indices.

IV. Optimizations

Although our TDT framework can save memory space and avoid repeated triangle computation/updates, the number of out-neighbors in the dense part of the graph may still be large, causing a costly and unbalanced workload in neighbor intersection. Thus, we further partition vertex neighbors and optimize the key functions to accelerate the framework.

A. Vertex Neighbor Partition

Our graph orientation strategy can approximately balance the workload of neighbor intersections. However, we still need to examine $|d_{out}(v)| + |d_{out}(u)|$ neighbors for edge (u, v), which is costly and may even exceed the limited share memory capacity of a thread. To further accelerate the intersection, we propose a new strategy to divide the neighbors of each vertex into n_{cut} blocks based on their index intervals. Such strategy is tailored for the tensor-based truss decomposition, differing from the existing neighbor-partition strategy [30], [31] in the following aspects.



(a) No partitioning (b) Partitioning into two blocks

Fig. 7: Vertex neighbor partition in the directed graph

Column based Neighbor Partition. From the perspective of the adjacency matrix, the column indices of the non-zero elements in the i-th row correspond to the indices of all the out neighbors of vertex i. Previous works [30], [31] divide the adjacency matrix by rows and columns simultaneously, where each block serves as a computation task to balance the workload. To fully leverage the parallelism of tensor operations, we only divide the columns of the matrix into n_{cut} blocks to reduce the computation cost of neighbor intersection while keeping the parallel computation of all the intersection tasks. Equipped with neighbor partition, we can divide the intersection of two long tensors into multiple sub-intersection operations of shorter tensors, which can speed up the intersection by skipping a sub-intersection operation if the neighbor block of a vertex is empty or has reached its end. For example, before partition, we need to compare the neighbor sets $\{3\}$ and $\{4, 5\}$ to compute the triangles containing $(0, 3)$ in Fig. 7(a); after partition, the intersection is decomposed into two sub-intersection tasks, and we can directly skip sub-intersection 1 as the neighbor block of vertex 3 is empty and skip sub-intersection 2 for a similar reason.

Lightweight Adjustment of Data Representation. To support the vertex neighbor partition, we need to adjust the neighbor index array $rptr$ accordingly. Initially, each element in $rptr$ marks the starting position of the neighbor list for the corresponding vertex; after partitioning, $rptr$ is updated to indicate the starting/ending positions of each neighbor block for every vertex. Fig. 7(b) shows the updated $rptr$ data after partitioning, where the first neighbor block of vertex 0 is $dst[rptr[0] : rptr[1]]$, and the second neighbor block is $dst[rptr[1] : rptr[2]]$. Compared with the row and column based partition [30], [31] which reuire reorganization of both of adjacent list and the corresponding pointer for each block, we only need to update the starting position in $rptr$, which

Algorithm 3: *ComputeSup+*

Input: $G_T = (E_T(src, dst), rptr)$ and n_{cut}
Output: sup

1 $sup \leftarrow$ `zeros`$(|src|)$;
2 **for** ***all*** $i, (u, v) \in$ ***enumerate***(E_T) ***in parallel*** **do**
3 **for** ***all*** $b \in \{1, 2, \cdots, n_{cut}\}$ ***in parallel*** **do**
4 $count \leftarrow 0$;
5 $j,\ end_j \leftarrow rptr[u \times n_{cut}+b],\ rptr[u \times n_{cut}+b+1]$;
6 $k,\ end_k \leftarrow rptr[v \times n_{cut}+b],\ rptr[v \times n_{cut}+b+1]$;
7 **while** $j < end_j$ ***and*** $k < end_k$ **do**
8 **if** $dst[j] == dst[k]$ **then**
9 `atomicAdd`$(\&sup[j], 1)$;
10 `atomicAdd`$(\&sup[k], 1)$;
11 $count++;\ j++;\ k++$;
12 **else if** $dst[j] < dst[k]$ **then** $j++$;
13 **else** $k++$;
14 `atomicAdd`$(\&sup[i], count)$;
15 **return** sup ;

Algorithm 4: *UpdateSup+*

Input: $G_T = (E_T(src, dst), rptr)$, I_{act}, M_{del}, l, sup, n_{cut}
Output: M_{next}

1 $M_{next} \leftarrow$ `full`$(sup.$`shape`$, false)$;
2 **for** ***all*** $i \in I_{act}$ ***in parallel*** **do**
3 **for** ***all*** $b \in \{1, 2, \cdots, n_{cut}\}$ ***in parallel*** **do**
4 $count \leftarrow 0;\ u,\ v \leftarrow src[i],\ dst[i]$;
5 $j,\ end_j \leftarrow rptr[u \times n_{cut}+b],\ rptr[u \times n_{cut}+b+1]$;
6 $k,\ end_k \leftarrow rptr[v \times n_{cut}+b],\ rptr[v \times n_{cut}+b+1]$;
7 **while** $j < end_j$ ***and*** $k < end_k$ **do**
8 **if** $dst[j] == dst[k]$ **then**
9 **if** $dst[j] != -1$ ***and*** $(M_{del}[i] \vee M_{del}[j] \vee M_{del}[k])$ **then**
10 **if** $sup[j] > l$ **then**
11 $s \leftarrow$ `atomicSub`$(\&sup[j], 1)$;
12 **if** $(s-1) == l$ **then** $M_{next}[j] \leftarrow true$;
13 **if** $sup[k] > l$ **then**
14 $s \leftarrow$ `atomicSub`$(\&sup[k], 1)$;
15 **if** $(s-1) == l$ **then** $M_{next}[k] \leftarrow true$;
16 **if** $\neg M_{del}[i]$ **then** $count++$;
17 $j++;\ k++$;
18 **else if** $dst[j] < dst[k]$ **then** $j++$;
19 **else** $k++$;
20 **if** $count > 0$ **then**
21 $s \leftarrow$ `atomicSub`$(\&sup[i], count)$;
22 **if** $(s - count) <= l$ **then** $M_{next}[i] \leftarrow true$;
23 **return** M_{next};

is a lightweight adjustment.

In addition, we adjust the input tensors of procedure *batch_csr_select* in Algorithm 2 to adapt our TDT framework to vertex neighbor partition. After partition, the starting and ending positions of the neighbors for a batch of vertices U are represented by $rptr[U \times n_{cut}]$ and $rptr[(U+1) \times n_{cut}]$, respectively. Thus, our partitioning strategy and adjustment of data representation can naturally fit our TDT framework.

B. Key Function Optimization

As introduced in Section III-C, the *ComputeSup* function retrieve the neighbors indices (I_u, I_v in Algorithm 2) of all edges to compute the edge support in parallel. However, storing neighbors indices of all the edges may need huge space ($O(|V||E|)$ in the worst case). The *UpdateSup* function also have similar issue when retrieving the neighbors indices of all the active edges. To avoid storing the neighbors indices of all processing edges and enable more flexible computations, we further optimize the functions *ComputeSup* and *UpdateSup* at CUDA levels, which directly access the neighbors without explicit neighbor indices construction and are packaged as Python APIs integrated into our framework.

ComputeSup+. Algorithm 3 outlines the optimized function *ComputeSup+*. Based on vertex neighbor partition, multiple threads can process different neighbor blocks of each edge simultaneously to accelerate the computation of intersections. We reduce atomic operations by accumulating the number of detected triangles in a per-thread local variable $count$, followed by a single atomic addition to update the support of the processing edge. n_{cut} threads parallelly compute the number of triangles on each edge (Lines 1-3). We initialize $count$ to zero. Then, using the pointer tensor $rptr$, each thread retrieves the starting and ending positions of the neighbor blocks for both vertices on the edge it is responsible for (Lines 5-6). Next, we use two pointers to traverse these two neighbor blocks. If the neighbors from both blocks match, the atomic operation `atomicAdd` is used to increment the support of two adjacent edges sharing a common neighbor, while also increasing the value of $count$. Otherwise, the pointers are moved to continue searching for the next common neighbor (Lines 7-13). Finally, `atomicAdd` is utilized to add $count$ to the support of the i-th edge (Line 14).

UpdateSup+. The optimized function *UpdateSup+* updates the support and marks the next peeling edges by processing each active edge in parallel, as detailed in Algorithm 4. *UpdateSup+* halts further reductions in the support values of affected edges and flags these edges as the next peeling edges as soon as they reach the peeling support value. This also efficiently excludes both already peeled and currently peeling edges from the next set of edges to peel. Moreover, we reduce the number of atomic operations by applying a single atomic decrement to each processing edge and only decrementing the support of neighboring edges when it exceeds the current trussness. Algorithm 4 executes the following four main steps:

(1) *Locat the neighbor blocks*. In the parallel computation of each neighbor block pair of the active edges, the variable $count$ is initialized to zero, and the source vertices u and the destination vertex v of the active edge are retrieved. Then, based on u, v, and the block number b, the start and end indices of the two neighbor blocks in dst are located for the following two-pointer intersection operation (Lines 2-6).

(2) *Identify peeling triangles*. We identify the peeling triangles that contain no edges marked for deletion and

include at least one peeling edge. Specifically, we exclude -1 as a valid common vertex, because the vertices of edges marked for deletion are set to -1 in dst. We also use the logical expression $M_{del}[i] \vee M_{del}[j] \vee M_{del}[k]$ to ensure that the triangle formed by the edges at positions i, j, k contains at least one peeling edge, as peeling edges are marked as *false* in M_{del} (Lines 7-10).

(3) *Update edge support*. We directly perform a decrement operation on the three edges of the peeling triangle. However, to reduce unnecessary atomic operations, `atomicSub` is only executed when the support of the edge is greater than the trussness value l (Lines 10-11, 13-14). Furthermore, if the currently processed active edge is not a peeling edge, we use the variable $count$ to accumulate the decrease in its support (Line 16), and then perform a single atomic decrement operation (Lines 20-21).

(4) *Mark the next peeling edges*. We leverage the property of `atomicSub` which returns the pre-subtraction value. If the returned value minus the value subtracted in `atomicSub` is less than or equal to l, we mark the processing edge for peeling in the next step (Lines 12, 15, 22).

C. Complexity Analysis

Memory Complexity. A tensorized directed Graph G_T containing src, dst, and $rptr$ requires $\mathcal{O}(n_{cut}|V| + |E|)$ space. The output tensors E'_T and $tptr$ require $\mathcal{O}(k_{\max}+|E|)$ space. The support tensor sup and mask tensors M_{del}, M_{next} require $\mathcal{O}(|E|)$ space. Since the max trussness $k_{\max}$ is far smaller than $|V|$, the total memory complexity of optimized TDT algorithm is $\mathcal{O}(n_{cut}|V| + |E|)$.

Time Complexity. Suppose that α is the average number of out-neighbors. The loop is executed β times and the graph compression is performed $\lfloor \frac{|E|}{\delta} \rfloor$ times. The time complexity of support computation based on out-neighbors is $\sum_{e\in E} \mathcal{O}(d_{out}(u) + d_{out}(v)) = \mathcal{O}(\alpha|E|)$. The time complexity for performing graph reduction is $\mathcal{O}(\lfloor \frac{|E|}{\delta} \rfloor(|V| + |E|))$. Finding active edges needs to examine each edge in every loop, whose time complexity is $\mathcal{O}(\beta|E|)$. For support update, the total number of active edges processed is $\sum_{e\in E} d_{out}(u)$ in the worst case (i.e., peeling one edge at a time); the time complexity for updating edges in the entire algorithm is $\mathcal{O}(\alpha \sum_{e\in E} d_{out}(u))$. Therefore, the total time complexity of the optimized TDT algorithm is $\mathcal{O}(\alpha \sum_{e\in E} d_{out}(u) + \lfloor \frac{|E|}{\delta} \rfloor|V| + (\lfloor \frac{|E|}{\delta} \rfloor+\alpha+\beta)|E|)$.

V. Experiments

In this section, we conduct extensive experimental studies to evaluate the performance of our TDT algorithm.

A. Experimental Setup

Baselines. We compare our TDT algorithm with four state-of-the-art CPU- and GPU-based algorithms.

- WC [2]. WC is a CPU-based truss decomposition algorithm which processes edges sequentially.
- Ros [32]. Ros is a CPU-based truss decomposition algorithm which computes support in parallel, peels edge and updates edge support sequentially.
- PKT [6]. PKT is a CPU-based parallel truss decomposition algorithm with shared-memory.
- H-IDX [8]. H-IDX is a CPU-based local truss decomposition method based on h-index.
- OPT-HPU [13]. OPT-HPU is a CPU-GPU co-processing truss decomposition algorithm which optimizes triangle counting with word-packing and accelerates support updates via dynamic switching.

TABLE I: Statistics of graph datasets

Datasets	Abbr.	$\|V\|$	$\|E\|$	$k_{\max}$
com-dblp	CD	317,080	1,049,866	114
Road-NetCA	RN	1,964,207	2,766,607	4
com-youtube	CY	1,134,890	2,987,624	19
amazon0601	A0	403,394	3,387,388	11
web-Google	WG	875,713	5,105,039	44
cit-Patents	CP	3,774,768	16,518,948	36
wiki-topcats	WT	1,791,489	28,511,807	39
soc-pokec-relationships	SP	1,632,803	30,622,564	29
Road-usa	RU	23,947,347	57,708,624	4
soc-orkut	SO	2,997,166	106,349,209	75
rgg-n-2-24-s0	RG	16,777,215	132,557,200	21
com-orkut	CO	3,072,627	234,370,166	78
mawi4	M4	128,568,730	135,117,420	3
protein4	P4	214,005,017	232,705,452	3
web-clueweb	CW	428,136,613	446,766,953	80
soc-sinaweibo	SS	58,655,850	522,642,066	80
web-cc12-PayLevel	WCP	42,889,800	582,567,291	2870
wikipedia	WP	27,154,799	1,086,367,222	428
uk-2005	UK	39,459,924	1,566,054,250	589

Environments. The source codes of the above baseline algorithms are provided in [33], where WC, ROS, PKT, and H-IDX are implemented by C++ and compiled with GCC 7.5.0 while OPT-HPU is implemented with C++/CUDA and compiled with GCC 7.5.0 and CUDA 10.1. Our algorithm TDT is implemented by Python with optimized function further implemented by CUDA, and is compiled with PyTorch 2.0.0 and compatible CUDA 11.8. The experiments are conducted on an NVIDIA V100 GPU with 32GB of VRAM memory, along with an Intel Xeon Skylake-SP CPU featuring 10 cores and 20 threads, operating at a base frequency of 3.0 GHz. It is also equipped with 72GB of RAM. Besides, a scalability test is also conducted on a machine equipped with an Intel Xeon Gold 6248R CPU and an NVIDIA A100 GPU recently released to analyze the performance of algorithms on different hardware. The operating system is Ubuntu 20.04.

Datasets. Table I provides the datasets evaluated in the experiments. All datasets are sourced from well-known platforms such as SNAP (Stanford Network Analysis Platform) [34] and Network Repository [35], covering real-world examples from social media, mobile communication, academic citation, and road traffic networks of various scales.

Parameter Setting. For graph reduction, the threshold for the number of peeled edges δ is set to 1,000,000 by default. Such value is chosen based on a trade-off between the time cost of graph reduction and the performance improvement of computations on the reduced graph, as shown later in the scalability testing. For the optimized functions ComputeSup+

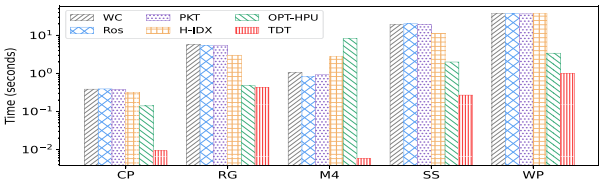


Fig. 8: Comparison of running time for support computation

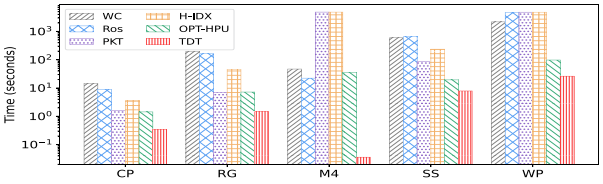


Fig. 9: Comparison of running time for edge peeling

and UpdateSup+, we allocate 512 threads per block, which is a common choice for efficient parallelism and resource utilization across most GPU architectures [36].

B. Comparison of Truss Decomposition Algorithms

The evaluated truss decomposition algorithms can be broadly divided into two phases: support computation and edge peeling. In this subsection, we provide a detailed analysis of the computation time for each phase and the overall truss decomposition.

Support Computation. Fig. 8 shows the support computation times across five representative datasets (two of small size, one of medium size, and two of large size) with increasing edge numbers for all the evaluated algorithms. Our TDT algorithm outperforms all the baselines in the support computation step across all datasets, achieving speedups of 1.5× to 150× over OPT-HPU, 6.9× to 480× over H-IDX, and 9× to 180× over WC. The general trend shows that support computation time increases with the size of the graph. However, on the M4 dataset, OPT-HPU is the slowest, whereas the other methods perform faster. This is due to M4 has an extremely large number of vertices while containing relatively few triangles. Thus, methods based on neighbor intersection operations can efficiently compute the support while the bitmap-based method OPT-HPU introduces significant overhead of bitmap construction and indexing for such a large number of vertices.

Edge Peeling. Fig. 9 shows the edge peeling times across the same five datasets. Our TDT method demonstrates exceptional performance, achieving speedups of over 1,000× on the M4 dataset and 350× on the P4 dataset compared with WC, ROS, PKT, and H-IDX. Moreover, TDT outperforms all the baselines by being more than 4× faster on the CP and RN datasets and 2.5× faster on the SS and WP datasets.

Overall Truss Decomposition. The total truss decomposition times are summarized in Table II. For datasets with extremely large number of vertices such as M4, P4, and CW, our TDT method demonstrates substantial performance advantages over baselines. This improvement benefits from our directed truss decomposition strategy, which processes triangles edge-by-edge and eliminates redundant triangle searches. In contrast, OPT-HPU utilizes a bitmap mechanism where the bitmap size is proportional to $|V|$ for each thread block, which is prohibitive for graphs with large number of vertices. For datasets with an exceptionally large number of edges, such as SS and WP, our TDT method achieves speedups of 2.6× and 3.7×, respectively, compared to OPT-HPU. However, for WCP and UK datasets, TDT is slightly slower than OPT-HPU, primarily due to increased GPU memory consumption during runtime, while OPT-HPU benefits from leveraging CPU-assisted storage and computation to mitigate such bottlenecks. Additionally, PKT and H-IDX run for over an hour on M4, WCP, WP, and UK datasets, while our algorithm TDT consistently outperforms them with a speedup of over 13× on large-scale datasets. For smaller graphs with edges less than one hundred million, TDT consistently outperforms baselines. Overall, TDT achieves 2.3×-8.5× speedup over OPT-HPU in most cases, excluding the two extremely highest speedups of 1061× and 85×.

TABLE II: Comparison of truss decomposition time (sec)

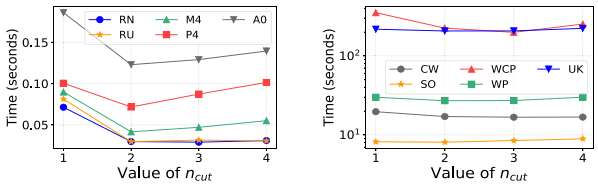
G	CPU-based				GPU-based		
	WC	Ros	PKT	H-IDX	OPT-HPU	TDT	Speedup
CD	0.619	0.373	0.155	0.265	0.636	**0.145**	4.381
RN	0.525	0.162	0.133	0.357	0.605	**0.071**	8.480
CY	2.621	2.825	0.692	1.416	0.889	**0.418**	2.126
A0	2.163	1.500	0.234	0.588	0.662	**0.123**	5.381
WG	4.116	4.258	0.842	1.868	1.103	**0.310**	3.559
CP	14.998	9.334	1.969	3.865	1.563	**0.345**	4.532
WT	55.414	72.804	31.882	53.568	2.215	**1.772**	1.250
SP	43.948	36.613	5.265	13.265	2.353	**1.007**	2.337
RU	9.928	1.992	2.480	3.234	2.527	**0.030**	85.084
SO	482.423	494.057	61.517	312.838	**12.586**	13.178	0.955
RG	202.283	167.976	12.528	48.391	7.651	**1.879**	4.072
CO	590.657	592.232	70.889	404.872	**13.892**	16.663	0.834
M4	47.736	23.023	>1h	>1h	44.136	**0.042**	1061.85
P4	55.403	11.430	8.966	24.495	-	**0.072**	-
CW	-	965.815	241.481	451.487	-	**16.750**	-
SS	616.125	664.860	104.800	246.356	21.264	**8.095**	2.627
WCP	>1h	>1h	>1h	>1h	**176.482**	198.660	0.888
WP	2201.213	4708.187	>1h	>1h	100.570	**27.092**	3.712
UK	2561.408	2562.230	>1h	>1h	**181.275**	206.661	0.877

In summary, TDT demonstrates superior performance across most datasets, especially for graphs with extremely large numbers of vertices. The main reasons are (1) workload reduction achieved by out-neighbor support computation/update while truss decomposition on undirected graphs requires intersections on longer neighbor lists or bitmaps (as shown in Fig. 8 and Fig. 9), (2) balanced workload enabled by vertex neighbor partition (to be shown later in Fig. 10), and (3) redundant computation elimination achieved by optimized key functions (to be shown later in Fig. 11). In addition, our algorithm also benefits from the hardware-independent optimizations provided by TCR compilers, such as operator fusion, operator sinking, and algebraic simplification, which can further accelerate the tensor computation on the underlying hardware, as stated in the introduction.

C. Evaluation of Optimization Strategies

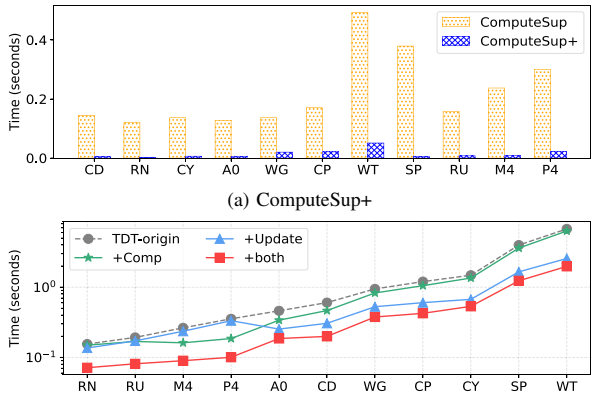
Next, we analyze the performance improvements achieved by the optimization strategies introduced in Sec. IV.

Vertex Neighbor Partition. We evaluate the impact of neighbor partitioning parameters in TDT on sparse and dense datasets, and show the results in Fig. 10. The run time for



(a) Sparse datasets (b) Dense datasets

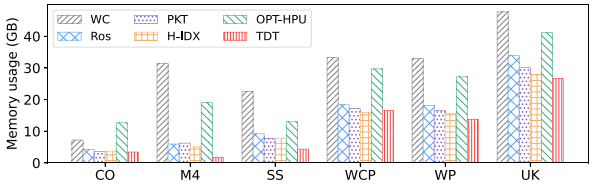
Fig. 10: Effect of neighbor partition n_{cut} on running time



(a) ComputeSup+

(b) Optimized strategy fusion

Fig. 11: Effect of ComputeSup+ and UpdateSup+



(a) Average memory cost on large datasets

(b) Average memory cost for varying graph densities (WP)

Fig. 12: Comparison of memory cost

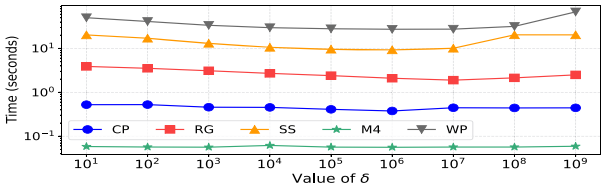


Fig. 13: Running time for different parameter δ

all datasets initially decreases sharply with the increase of partition numbers, then gradually increases. This is because, with more partitions, the computational workload is more evenly distributed across threads. However, when the number of parallel threads exceeds the physical limit supported by the hardware, it leads to an increase in thread scheduling overhead and a subsequent drop in performance. Furthermore, for the sparser graphs RN, M4, A0, RU, and P4, the partitioning optimization achieves time speedups of 2.4×, 2.2×, 1.5×, 2.7×, and 1.4×, respectively. On the denser large graph, a speedup of 1.77× is achieved for WCP, and speedups ranging from 1.1× to 1.2× are achieved for other datasets as they are already well-balanced after pre-processing.

Optimized Support Computation and Update. We first compare the computation time of *ComputeSup* and *ComputeSup+*. Fig. 11(a) shows that *ComputeSup+* achieves speedups of up to 74× on the SP dataset and a minimum of 6.8× on the WG dataset. Then, we evaluate the combinations of *ComputeSup+* and *UpdateSup+*, which has four versions: TDT-origin without any optimization; +Comp and +Update utilizing only *ComputeSup+* or *UpdateSup+*; +both using both optimization functions. As shown in Fig. 11(b), the computation time of the baseline TDT-origin is consistently the highest, while +both consistently achieves the lowest computation time. More importantly, for most datasets, the optimization provided by *UpdateSup+* is more significant. However, for datasets M4 and P4 with $k_{\max} = 3$, *ComputeSup+* demonstrates higher efficiency as support computation dominates the total run time. Overall, the combined optimizations deliver the highest speedups, boosting the original TDT by 2.2× to 3.4×.

D. Scalability Testing

We further evaluate the scalability of the algorithms regarding memory, parameter, hardware, and GPU numbers.

Scalability on the Memory Cost. We evaluate the average memory cost (combined CPU and GPU memory) during the computation process as the evaluated algorithms utilize either CPU/GPU memory or both. Due to the inefficiency of automatic GPU memory release in PyTorch, we use cuda.empty_cache() to clear the GPU memory cache at the beginning of each iteration in TDT for a fair comparison. Fig. 12(a) shows the result of large-scale datasets on which all the evaluated algorithms can successfully run, including CO, M4, SS, WCP, WP, and UK. Overall, TDT achieves the least average memory cost. Specifically, compared to WC and OPT-HPU, TDT reduces the memory cost by 35% to 94% due to (1) the hash map in WC requires substantial memory and (2) OPT-HPU stores undirected graph on both CPU and GPU; compared to other CPU-based algorithms (Ros, PKT, and H-IDX), the average memory reductions of TDT are only 33%, 25%, and 21%, as the GPU memory release mechanism in TDT is inefficient compared to CPU.

We further investigate the impact of graph density on memory optimization by randomly sampling different number of edges from the same dataset (e.g., WP) while keeping the same number of vertices. Fig. 12(b), shows that as the number of sampled edges increases, the average memory cost of the CPU-based algorithm increase almost linearly, while that of the GPU-based algorithm shows a slight fluctuation, as the

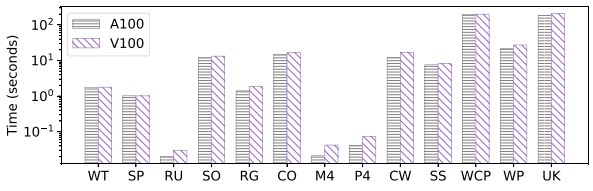


Fig. 14: Running time for different hardware

TABLE III: Running time on machine with 2 GPU (sec)

Type	Datasets					
	CP	WT	SP	SO	CO	SS
1-GPU	0.345	1.772	1.007	13.178	16.663	8.095
2-GPU	0.231	0.908	0.529	7.191	8.115	3.536
Speedup	1.494	1.951	1.902	1.833	2.053	2.289

GPU memory is not immediately released as CPU memory leading to different degree of latency for different runnings. Overall, as the number of edge/density increases, TDT still costs the least memory among all the evaluated algorithms.

Scalability on parameters. We evaluate the effect of δ in our TDT algorithm and show the result in Fig. 13. As δ increases from 10 to 10^9, the running time of TDT initially decreases and then increases slowly, and stably performs well in the range from 10^3 to 10^7. Except for too small/large δ causing too frequent/rare graph reduction, our algorithm is robust to the change of δ. We set 10^6 as the default value where TDT achieves the best performance for most cases.

Scalability on Hardware Platforms. To evaluate the performance of algorithms on different hardware, besides NVIDIA V100, we also evaluate TDT on the recently released GPU NVIDIA A100. OPT-HPU is not reported as the required running environment cannot be supported by A100. The memory costs on these two hardware are negligible, and we only report the running time, as shown in Fig. 14. Compared to the previous-generation V100, TDT on A100 achieves a speedup of 1.1× to 2.0×. It shows that our TDT can be well accelerated by recent hardware.

Scalability on Multi-GPUs. The TDT algorithm can be easily modified to adapt multi-GPU approach for task partitioning, where different GPUs handle truss subgraphs corresponding to different ranges of trussness values. In Table III, we present the results of experiments by extending the TDT algorithm to a dual-GPU setup. The dual-GPU task partitioning approach achieves a performance improvement ranging from 1.494× to 2.289×.

In summary, our TDT algorithm has the least memory cost for different graph densities and robust performance for different parameters and hardware, and can be significantly accelerated by multiple GPUs.

VI. Related Work

CPU-based Truss Computation. The k-truss model is defined by Cohen [1], where each edge is contained in at least k-2 triangles inside, and the *in-memory* algorithm is proposed. Wang et al. [2] proposed an improved *in-memory* algorithm to sort all edges in ascending order of support after support computation, and two *out-of-core* algorithms in the bottom-up and top-down manners. Wu et al. [3] further optimized the serial edge-peeling algorithm and the asynchronous h-index update algorithm using the graph compression framework Webgraph [37]. To efficiently handle large graphs, distributed k-truss decomposition algorithms based on MapReduce [4] and the bulk synchronous parallel model [9] have been developed. [32] is shared-memory algorithm which computes support in parallel, peels edge, and updates support sequentially. PKT [6] and MSP [7] are both shared-memory parallel algorithms with different data structures. Sariyüce et al. [8] developed a parallel shared-memory framework for k-truss and nucleus decompositions by extending iterative h-index computation.

GPU-based Truss Computation To further accelerate k-truss decomposition, *GPU-based algorithms* have been proposed, such as KTrussExplorer [30] providing multiple configurable kernels, a linear-algebra based method [38] that renumbers vertices to balance the workload and stores adjacency lists of the same vertex in shared memory to optimize memory access. These two algorithms are tailored to k-truss query but not truss decomposition. To deal with large-scale graphs, *CPU-GPU collaborated algorithms* are further developed. Date et al. [10] utilized "zero-copy" memory and "unified" memory to store adjacency lists, which can be directly accessed by both CPU and GPU threads to simplify memory management. Mailthody et al. [11] further optimized the computation by short and long updates of the peeled edges. Blanco et al. [12] proposed a fine-grained parallel algorithm based on linear algebra using the edge-centric eager k-truss decomposition strategy. Recently, Che et al. [13] proposed OPT-HPU to improve the bitmap-based triangle counting by a word-packing and accelerate support update by dynamic switch between support recomputation and decrement.

VII. Conclusion

In this paper, we propose the tensor based truss decomposition framework that can well leverage heterogeneous hardware for acceleration and well integrates with the downstream ML tasks. By converting graphs into directed ones and represent them by compact tensors, our approach reduces storage requirements and eliminates redundant triangle enumeration. Additionally, partitioning vertex neighbors, along with optimizing support computation and updates, further enhances performance. Experimental results demonstrate that TDT outperforms state-of-the-art methods in terms of efficiency and scalability, making it a powerful tool for large-scale graph analysis and graph machine learning tasks.

Acknowledgments

The work was supported in part by grants of National Natural Science Foundation of China (No. 62272353 and No. 62276193), the Research Grants Council of Hong Kong (No. 14205520), and CCF-Huawei Populus Grove Fund. Yuanyuan Zhu and Tieyun Qian are corresponding authors.

REFERENCES

[1] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, pp. 1–29, 2008.

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

[3] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo, "K-Truss Decomposition of Large Networks on a Single Consumer-Grade Machine," in *ASONAM*, 2018, pp. 873–880.

[4] J. Cohen, "Graph Twiddling in a MapReduce World," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[5] V. T. Chakaravarthy, A. Goyal, P. Murali, S. S. Pandian *et al.*, "Improved Distributed Algorithm for Graph Truss Decomposition," in *ICPDC*, 2018, pp. 703–717.

[6] H. Kabir and K. Madduri, "Shared-Memory Graph Truss Decomposition," in *HiPC*, 2017, pp. 13–22.

[7] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *HPEC*, 2017, pp. 1–6.

[8] A. E. Sarıyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *PVLDB*, vol. 12, no. 1, pp. 43–56, 2018.

[9] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *Big Data*, 2014, pp. 471–480.

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

[11] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (CPU + GPU) Algorithms for Triangle Counting and Truss Decomposition," in *HPEC*, 2018, pp. 1–7.

[12] M. P. Blanco, T. M. Low, and K. Kim, "Exploration of Fine-Grained Parallelism for Load Balancing Eager K-truss on GPU and CPU," in *HPEC*, 2019, pp. 1–7.

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

[14] M. Kumar, S. Mishra, S. S. Singh, and B. Biswas, "Community-enhanced link prediction in dynamic networks," *ACM Trans. Web*, vol. 18, no. 2, pp. 24:1–24:32, 2024.

[15] S. S. Singh, S. Mishra, A. Kumar, and B. Biswas, "CLP-ID: community-based link prediction using information diffusion," *Inf. Sci.*, vol. 514, pp. 402–433, 2020.

[16] A. De, S. Bhattacharya, S. Sarkar, N. Ganguly, and S. Chakrabarti, "Discriminative link prediction using local, community, and global signals," *TKDE*, vol. 28, no. 8, pp. 2057–2070, 2016.

[17] C. Song, Q. Lin, G. Ling, Z. Zhang, H. Chen, J. Liao, and C. Chen, "Locec: Local community-based edge classification in large online social networks," in *ICDE*, 2020, pp. 1689–1700.

[18] S. Ji, X. Lu, M. Liu, L. Sun, C. Liu *et al.*, "Community-based dynamic graph learning for popularity prediction," in *KDD*, 2023, pp. 930–940.

[19] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," in *SIGMOD*, 2020, pp. 2183–2197.

[20] D. He, S. C. Nakandala, D. Banda, R. Sen, K. Saur, K. Park *et al.*, "Query processing on tensor computation runtimes," *PVLDB*, vol. 15, no. 11, pp. 2811–2825, 2022.

[21] D. Koutsoukos, S. Nakandala, K. Karanasos, K. Saur, G. Alonso, and M. Interlandi, "Tensors: An abstraction for general data processing," *PVLDB*, vol. 14, no. 10, pp. 1797–1804, 2021.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat *et al.*, "Tensorflow: a system for large-scale machine learning," in *OSDI*, 2016, pp. 265–283.

[23] A. Paszke, S. Gross, F. Massa, S. Chintala, E. Cannan *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NIPs*, vol. 32, 2019.

[24] T. Chen, M. Li, Y. Li, M. Lin, N. Wang *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, 2015.

[25] T. Chen, T. Moreau, Z. Jiang *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *OSDI*, 2018, pp. 578–594.

[26] Microsoft, "Onnx runtime," 2022, accessed on October 24, 2024. [Online]. Available: https://github.com/microsoft/onnxruntime

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

[28] S. Pandey, Z. Wang, S. Zhong, C. Tian *et al.*, "Trust: Triangle counting reloaded on gpus," *TPDS*, vol. 32, no. 11, pp. 2646–2660, 2021.

[29] J. Huang, H. Wang, X. Fei, X. Wang, and W. Chen, "Tc-stream: Large-scale graph triangle counting on a single machine using gpus," *TPDS*, vol. 33, no. 11, pp. 3067–3078, 2022.

[30] S. Diab, M. G. Olabi, and I. El Hajj, "KTRUSSEXPLORER: Exploring the Design Space of K-truss Decomposition Optimizations on GPUs," in *HPEC*, 2020, pp. 1–8.

[31] A. Yaşar, S. Rajamanickam, J. W. Berry, and Ü. V. Çatalyürek, "A block-based triangle counting algorithm on heterogeneous environments," *TPDS*, vol. 33, no. 2, pp. 444–458, 2021.

[32] R. A. Rossi, "Fast Triangle Core Decomposition for Mining Large Graphs," in *PAKDD*, 2014, pp. 310–322.

[33] Y. Che, "Acctrussdecomposition," 2024. [Online]. Available: https://github.com/RapidsAtHKUST/AccTrussDecomposition

[34] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[35] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAA*, 2015, pp. 4292–4293.

[36] N. Corporation, *CUDA C Programming Guide*, 2025, [Online]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[37] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *Big Data*, 2014, pp. 9–16.

[38] R. Wang, L. Yu, Q. Wang, J. Xin, and L. Zheng, "Productive High-Performance k-Truss Decomposition on GPU Using Linear Algebra," in *HPEC*, 2021, pp. 1–7.