Accelerating Truss Decomposition on Heterogeneous Processors

Yulin Che* Zhuohang Lai* Shixuan Sun† Yue Wang‡ Qiong Luo*
Hong Kong University of Science and Technology* National University of Singapore, Singapore†
Shenzhen Institute of Computing Sciences, Shenzhen University‡
*{yche, zlai, luo}@cse.ust.hk, †{sunsx}@comp.nus.edu.sg, ‡{yuewang}@sics.ac.cn

ABSTRACT

Truss decomposition is to divide a graph into a hierarchy of subgraphs, or trusses. A subgraph is a k-truss ($k \geq 2$) if each edge is in at least $k-2$ triangles in the subgraph. Existing algorithms work by first counting the number of triangles each edge is in and then iteratively incrementing k to peel off the edges that will not appear in $(k+1)$-truss. Due to the data and computation intensity, truss decomposition on billion-edge graphs takes hours to complete on a commodity computer.

We propose to accelerate in-memory truss decomposition by (1) compacting intermediate results to optimize memory access, (2) dynamically adjusting the computation based on data characteristics, and (3) parallelizing the algorithm on both the multicore CPU and the GPU. In particular, we optimize the triangle enumeration with data skew handling, and determine at runtime whether to pursue peeling or direct triangle counting to obtain a certain k-truss. We further develop a CPU-GPU co-processing strategy in which the CPU first computes intermediate results and sends the compacted results to the GPU for further computation. Our experiments on real-world datasets show that our implementations outperform the state of the art by up to an order of magnitude. Our source code is publicly available at `https://github.com/RapidsAtHKUST/AccTrussDecomposition`.

PVLDB Reference Format:
Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, Qiong Luo. Accelerating Truss Decomposition on Heterogeneous Processors. *PVLDB*, 13(10): 1751-1764, 2020.
DOI: https://doi.org/10.14778/3401960.3401971

1. INTRODUCTION

A truss [14] in an undirected graph G is a subgraph whose cohesiveness exceeds a certain threshold. The cohesiveness is measured by the *support* of each edge in the subgraph, which is the number of triangles in the subgraph each of which contains the edge. A k-truss ($k \geq 2$) is the largest subgraph of G such that the support of each edge in the subgraph is not less than $k-2$. Fig. 1 shows an example graph with the support value of each edge in its *class*: an edge is in the k-class if it is in the k-truss, but not in the $(k+1)$-truss. Truss decomposition is to find the k-class for each edge [38].

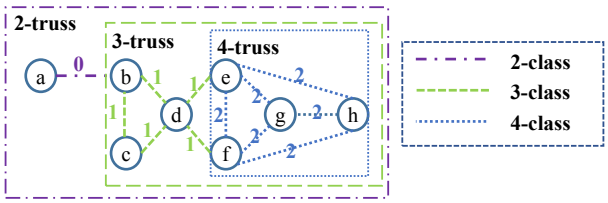


Figure 1: An example graph G

Truss decomposition has various applications in graph mining and social networks, such as community search and personalized recommendation [4, 19, 29, 31]. However, truss decomposition is time-consuming on large graphs due to the high intensity of computation. For example, state-of-the-art in-memory truss-decomposition algorithms [28,38] took tens of hours to complete on a billion-edge twitter graph in our experiment setting. Therefore, we study how to accelerate in-memory truss decomposition on big graphs.

Existing truss decomposition algorithms first initialize the support values of all edges, and follow by several iterative edge peeling phases [28, 38]. The initialization phase computes the common neighbor count $|N_G(u) \cap N_G(v)|$ of each edge $e(u,v)$ as e's support in G, where $N_G(u)$ is the neighbor set of u in G. After that, each peeling phase increments k ($k \geq 2$) and iteratively peels off the edges of which the support values are equal to or less than $k-2$. In phase k, the edges that are peeled off are put into the k-class, and the support values of all remaining edges in the triangles containing those peeled-off edges are updated. The algorithm ends when all edges in G are peeled off.

The time complexity of truss decomposition is linear to that of triangle enumeration in G [38]. For each peeled-off edge $e(u,v)$, triangle enumeration finds all triangles $\triangle_{uvw}$ from the remaining edges. For each triangle $\triangle_{uvw}$, support update is performed on edges $e(u,w)$ and $e(v,w)$. As such, the triangle enumeration operation is expensive due to intensive triangle existence checking, and the support update incurs intensive random memory accesses. As a result, truss decomposition is time-consuming, and researchers have proposed to parallelize the algorithms to speed up.

The state-of-the-art parallel algorithms, i.e., PKT [21] and MSP [34] parallelize the support initialization by creating a directed graph, enumerating each triangle in the directed graph once, and atomically updating the support values for all edges in the triangle. Subsequently, they parallelize the iterations of edge peeling within each phase and synchro-

Proceedings of the VLDB Endowment, Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401971

nize at the end of each iteration. However, these algorithms ignore the data characteristics in graphs and may perform unnecessary triangle existence checking and support update operations. Thus, we propose to design better triangle enumeration and support update procedures based on data characteristics. Specifically, we optimize the processing in the presence of *data skew*. In particular, we define *vertex degree skew* as $d(u) \gg d(v)$ in edge $e(u, v)$.

To avoid atomic operations in the support initialization, we directly count the number of triangles each edge $e(u, v)$ is in, following a bitmap-based all-edge triangle counting algorithm [11]. Specifically, suppose vertex degrees $d(v) < d(u)$ in edge $e(u, v)$, we construct a bitmap for the set of u's neighbors $N(u)$. For each $w \in N(v)$, we probe the bitmap of $N(u)$ to check the existence of triangle $\triangle_{uvw}$. To reduce the number of probes, we pack a neighbor set $N(v)$ into a set of non-zero 64-bit machine words, and utilize these words to perform word-wise look-ups on the bitmap of $N(u)$. In comparison with previous work [11], this packing technique reduces the number of probes significantly when there are many consecutive vertex IDs in $N(v)$.

To accelerate the core component of the iterative edge peeling, i.e., the support update procedure, we dynamically determine whether to (1) recompute all the support values via the direct triangle counting on the remaining edges, or to (2) decrement the support values for affected edges only. When lots of edges are peeled off, we favor the counting based algorithm to avoid intensive atomic operations and random memory accesses; otherwise, we take the support decrementing based procedure to reduce the workload.

To further improve the triangle enumeration procedure, we reduce the number of triangle existence checking operations by (1) designing a pivot-skip merge algorithm to handle data skew, (2) compacting the neighborhood information of each vertex periodically to reduce memory accesses and (3) eliminating the unnecessary enumeration.

At the beginning of a peeling phase, to efficiently filter the edges to peel off, we index the edges of which the support values are in a range $[i, i + rs)$, where i starts from 0 and increments at a step of rs. Specifically, we reconstruct the index structure for a new range every rs phases, maintain the index structure during the support update, and scan only the indexed edges for the filtering. With the index, we avoid scanning the entire set of remaining edges.

Finally, we parallelize our algorithms on both the multicore CPU and the GPU, and design offloading strategies to limit the data transfer between the CPU and the GPU.

We evaluate the effects of individual techniques on real-world billion-edge graphs and compare our optimized solutions with existing work. Experimental results show that (1) it always works best to first compute on the CPUs, compact the intermediate results, and then offload the computation to the GPU; (2) our optimized implementation is up to 68.7x faster than the state of the art [21,30,34], and completes the computation on a 680 million-edge twitter graph within 88 seconds on a computer with two 10-core Intel Xeon CPUs and an Nvidia V100 GPU.

In summary, we make the following contributions.

- We design a word-packing technique to improve an existing bitmap-based triangle counting algorithm [11] on all the edges for the support initialization.
- We design three optimized procedures for the iterative edge peeling: (i) dynamic triangle counting and peeling selection based support update, (ii) triangle enumeration with data skew handling and graph compaction, and (iii) index-based edge filtering.
- We parallelize and optimize our algorithms on both the multicore CPU and the GPU.
- We evaluate the effects of individual techniques on both the multicore CPU and the GPU and show that our implementations on both platforms outperform the state of the art by up to an order of magnitude.

Table 1: Summary of Notations

Notation	Description
G	An undirected graph
V, E, and $\triangle$	Vertex, edge, and triangle sets of G
$N(u)$	Neighbor set of a vertex u in G
Φ_k	The k-class of G
$rptr$, adj	Compressed Sparse Row (CSR) format of G
el	An edge list array of G
eid	An edge mapping array, associated with adj
$sup(e)$, $sup(e, G)$	Support value of an edge e in G
$P(e)$	Processing status of an edge e in G
$B(u)$	Bitmap representation of $N(u)$
$W_I(u), W_C(u)$	Indexes and contents of non-zero words in $B(u)$
$pr(u)$	Pack ratio ($\lvert N(u)\rvert/\lvert W_I(u)\rvert$) of a vertex u in G
Q_C, Q_N	Queues for edge filtering
Q_I	Index structure for Q_C
wpt	Threshold for neighbor set word-packing
ct	Threshold for graph compaction
ept	Estimated peeling throughput
rs	Size of an index range $[beg, beg + rs)$
PP and IEP	Pre-processing and iterative edge peeling
SI and WP	Support initialization and word-packing
SU and TE	Support update and triangle enumeration
EF	Edge filtering
TC and TP	Triangle counting and triangle peeling
DSTCP	Dynamic selection of TC and TP procedures
PSM and VM	Pivot-skip-merge and vectorized block-wise merge
$d(u) \gg d(v)$	Data skew (vertex degree skew)
$d(u)/d(v)$	Degree skew ratio ($d(u) \geq d(v)$)
GC and ES	Graph compaction and enumeration skipping
BMPF and IDX	Bitmap filtering and indexing

2. BACKGROUND AND RELATED WORK

In this section, we describe the problem statement of truss decomposition, categorize related work, and show our profiling results on a representative parallel algorithm.

2.1 Preliminaries

We consider an undirected graph G and denote the vertex, edge, and triangle sets of G by V_G, E_G, and $\triangle_G$, respectively. Given a vertex $u \in V_G$, $N_G(u)$ is the set of neighbors of u, and $d_G(u)$ denotes the degree of u, i.e., $d_G(u) = |N_G(u)|$. We denote the triangle of three vertices $u, v, w \in V$ by $\triangle_{uvw}$.

DEFINITION 1. *(Support) The support of an edge $e(u, v) \in E_G$, denoted by $sup(e, G)$, is defined as $|\{\triangle_{uvw} | w \in V_G\}|$, which can be computed by $|N_G(u) \cap N_G(v)|$.*

DEFINITION 2. *(k-Truss) The k-truss of G ($k \geq 2$), denoted by T_k, is defined as the largest subgraph of G such that $\forall_{e \in E_{T_k}} sup(e, T_k) \geq (k-2)$. The trussness (truss number) of an edge $e \in E$, denoted by $\phi(e)$, is defined as the maximum k of the k-truss that the edge e is in.*

DEFINITION 3. *(k-Class) The k-class of G denoted by Φ_k is defined as $\{e | e \in E \wedge \phi(e) = k\}$. All k-classes of G form a hierarchy. A k-truss of G can be computed by a union of all the i-classes ($i \geq k$) [38].*

Problem Statement. Truss decomposition of graph G is to find all k-classes Φ_k ($k \geq 2$) of G.

We summarize the frequently used notations in Table 1.

Algorithm 1: Truss Decomposition

Input: an undirected graph $G_0 = (V_{G_0}, E_{G_0})$
Output: all the i-classes Φ_i of G_0 $(i \geq 2)$

```
1  G ← G_0, k ← 1
2  foreach e(u,v) ∈ E do sup(e(u,v)) ← |△_uvw|
3  while |E| > 0 do
4      k ← k + 1, Φ_k ← ∅, Q ← {e | e ∈ E ∧ sup(e) = k − 2}
5      while |Q| > 0 do
6          Φ_k ← Φ_k ∪ Q
7          foreach e(u,v) ∈ Q do
8              foreach e' ∈ △_uvw do
9                  sup(e') ← max(k − 2, sup(e') − 1)
10             E ← E \ {e}
11         Q ← {e | e ∈ E ∧ sup(e) = k − 2}
12 return {Φ_i | 2 ≤ i ≤ k}
```

2.2 Related Work

Existing Algorithms. State-of-the-art algorithms [21, 28, 34, 38] perform support initialization, followed by iterative edge peeling phases (Alg. 1). The support initialization (Line 2) is to compute the triangle counts for all the edges, and the peeling phases (Lines 3-11) proceed level by level and find a k-class ($k \geq 2$) at each level. In each level, edges with the support value k are filtered (Lines 4 and 11), which triggers the peeling of triangles containing these edges, which in turn causes the support update of the edges in these triangles. Each triangle is peeled off exactly once for correctness, which is ensured by removing the edge $e(u, v)$ (Line 10) after completing the enumeration of $\triangle_{uvw}$ and support update of $e(v, w)$ and $e(u, w)$ (Lines 8 and 9).

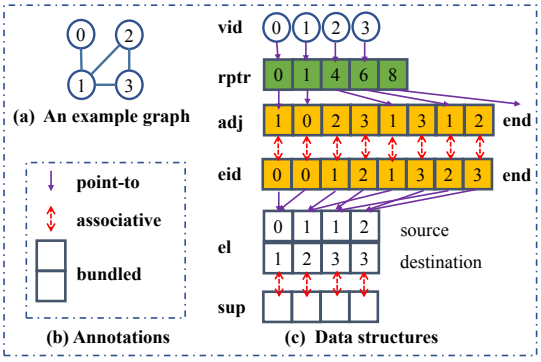


Figure 2: An example graph and its storage

Data Structures. The in-memory storage for the truss decomposition (Alg. 1) consists of adjacency lists, edges, support values, and the mappings from the triangles to the edges. We show the data structures in Fig. 2. Specifically, the adjacency lists are represented in a Compressed Sparse Row (CSR) format [15,21,24], which consists of row pointers and adjacency arrays denoted by *rptr* and *adj*, respectively. For triangle enumeration (TE), each adjacency list is sorted and used for subsequent merge operations [21, 34]. Edges (denoted by *el*) are represented as a list of source and destination vertex pairs, and each edge is associated with a support value denoted by $sup(e)$. To quickly map a TE result (three offsets in the *adj* array) into edge indexes, an *eid* array is introduced and associated with *adj* [21].

Support Initialization (SI). The SI step counts the number of triangles that each edge is in (Line 2 of Alg. 1). There are two approaches to parallelizing SI. The first approach [27,37,39] creates a Degree Oriented Directed Graph (DODG) by turning each undirected edge into a directed one. The direction of each edge is from a smaller degree vertex to a larger degree vertex. Using DODG, each triangle is enumerated exactly once, and the support values of edges in the triangle are updated atomically. In contrast, the second approach [11] directly computes the triangle count of each edge on the original graph and updates the support value of each edge exactly once.

Merge-based, hash-based, and bitmap-based set intersection algorithms for triangle counting (TC) [17,18,36,40] can be applied to SI. Merge-based algorithms scan the two sorted arrays of $N(u)$ and $N(v)$ and compare the elements to find matches, for example, vectorized block-wise-merge (VM) on the CPUs [11,40] and binary-search-based merge (BSM) [17, 18] on the GPUs. In contrast, hash-based algorithms construct a hash table for each $N(u)$. Then for each $w \in N(v)$, they probe the hash table to find common neighbors of u and v. In this category, Tom et al. [36] optimized the hash table construction by splitting $N(u)$ into dense and sparse parts and representing the dense part in an array to reduce the probe cost. Similar to hashing, Che et al. [11] used a bitmap (BMP) to represent $N(u)$ and dynamically construct and clear the bitmap during the all-edge common neighbor counting. Recently, on the GPUs, hardware-conscious memory accesses and load balance strategies have been studied to further improve the performance of BSM [17, 18] and hash-based set intersection algorithms [26] for TC.

When there is data skew ($d(u) \gg d(v)$, given edge $e(u, v)$), hash-based, BMP, and BSM algorithms work better than the VM [11] algorithm because of the $O(d(v))$, $O(d(v))$, $O(log(d(u)) \cdot d(v))$, and $O(d(u) + d(v))$ time complexity, respectively. In practice, $N(u)$ may have dense parts with many consecutive vertex IDs. However, prior work neglects this data characteristic and does not compact the information in $N(u)$ to reduce number of operations in $N(u)$.

Iterative Edge Peeling (IEP) on multicore CPUs. PKT [20,21] and MSP [34] parallelize each edge peeling iteration and synchronize among iterations. They differ in the design of adjacency representation, support update (SU), and edge filtering (EF) procedures. PKT [20, 21] adopts the CSR format and introduces a boolean array to indicate the edge removals, whereas MSP [34] maintains array-based doubly-linked lists and dynamically updates them upon edge removals. For SU, PKT uses atomic operations, whereas MSP expands all the edges in the peeled-off triangles and groups these edges by the source vertex for lock-free computation. For EF (Line 4 of Alg. 1), PKT scans all the edges, whereas MSP indexes the edges by grouping the edges with identical support values into a bucket. MSP maintains the bucket index during SU in $O(|\triangle|)$ time and space.

Existing triangle peeling based SU algorithms (TPSU) [21, 34] take triangle enumeration results ($N(u) \cap N(v)$) as the input and run $O(|\triangle|)$ SU operations on the unprocessed edges of enumerated triangles (Line 9 of Alg. 1). In these algorithms, some iterations may involve a large number of triangles due to a lot of connections among the set of filtered edges Q. For example, an isolated clique with thousands of vertices contains billions of triangles. However, in such cases, SU for the edges in the clique is useless since those edges will be removed right after the edge peeling iteration. Also, the triangle peeling involves intensive random memory accesses on the edges of enumerated triangles.

IEP on GPUs and Clusters. On GPUs, Vikram et al. [15, 24] proposed to first identify the edges with affected support values via triangle enumeration, and then recompute the support values of the affected edges. They imple-

mented on architectures with both CPUs and GPUs. Specifically, they adopted the unified memory interface of GPUs and CPUs, divided up the tasks in a single edge peeling iteration, and distributed the tasks to CPUs and GPUs. Due to the random memory access pattern of affected edges, their method incurred intensive memory page swaps among processors. As a result, increasing the number of processors even slowed down the execution [15, 24]. To address the drawback of intensive data transfer, Mohammad et al. [5] proposed to evaluate k-truss finding tasks of different k values in parallel across GPUs. However, this approach inevitably incurred more edge peeling operations, since edge removal status was not shared across GPUs. Also, load imbalance occurred between different k-truss finding [5] tasks.

Other Algorithms. Recently, Sariyuce et al. [30] proposed to extend an h-index-based algorithm for core decomposition [8, 23, 25] to truss and nucleus decomposition [30]. However, their implementation was up to 10x slower than the peeling-based MSP [34] due to the higher time complexity. Wu et al. [39] proposed to optimize the memory usage of truss decomposition algorithms [30, 38] by compressing the adjacency lists of a graph in a WebGraph [7] framework and using the CSR format to represent a sorted edge list.

Recently, truss maintenance on dynamic graphs has been studied. Specifically, Zhang et al. [41] and Huang et al. [19] have studied how to track the trusses given edge insertions and deletions. However, in edge insertion cases, these algorithms do not have a polynomial time complexity bound in terms of the input and output change size [41]. As such, for a situation with intensive edge insertions, truss decomposition may be a better solution than truss maintenance.

2.3 Analysis

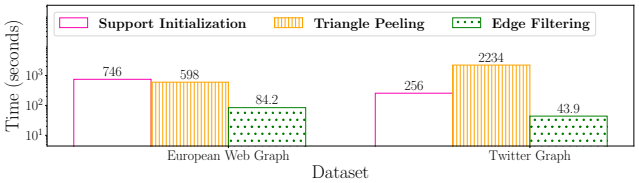


Figure 3: Time breakdown of PKT

To study the performance bottleneck of truss decomposition, we profile the parallel algorithm PKT [21] on two billion-scale graphs, namely the European web (WE) and a twitter graph (TW) from WebGraph [6, 7]. We use the same dataset, hardware, and PKT implementation as in Sect. 5. We show the time breakdown results in Fig. 3. First, the support initialization (SI) takes hundreds of seconds on the two graphs, which is 4x-10x slower than the state-of-the-art all-edge common neighbor counting algorithms [11]. The inefficiency lies in the $O(|\triangle|)$ atomic update operations. Second, the triangle peeling, which consists of triangle enumeration, support update, and queue maintenance procedures, consumes 8.7x more time than SI on TW. Third, the edge filtering of PKT takes tens of seconds to scan all edges and generate the queue (Line 4 of Alg. 1). The filtering takes less time than the other two components.

3. DESIGN

In this section, we describe the design of our optimized truss decomposition, which consists of (1) a pre-processing (PP) stage to initialize an edge list *el* and an edge mapping *eid* array (Alg. 2), (2) a support initialization (SI) stage to count the number of triangles that each edge is in (Alg. 3), and (3) an iterative edge peeling (IEP) stage (Alg. 4).

Our PP stage is different from previous work PKT-PP [21] and consists of three loops, each of which is efficiently parallelized. We eliminate the loop-carried dependencies in PKT-PP and utilize a prefix-sum parallel primitive.

In the SI stage, we follow the bitmap-based direct triangle counting algorithm [11], and further introduce a word-packing technique to reduce the bitmap-probe workload.

In the IEP stage, we record the number of accumulated removals by n_{rm} and periodically compact the adjacency lists to reduce the memory accesses in the triangle enumeration (TE). We further dynamically select triangle counting and peeling-based support update procedures to utilize the data and computation characteristics of support update. Moreover, we design a pivot-skip merge algorithm to handle the data skew in TE and safely skip unnecessary enumeration.

To accelerate the edge filtering in the peeling process, we use two queues Q_C and Q_N to store the edges to process in the current and next iterations, respectively. Moreover, we introduce a queue Q_I to index the edges by their support values. The three queues enable us to track edges with identical support values efficiently. We use a boolean array to indicate the queue occupation status in a given range, i.e., $[0, |E|)$. Checking the existence of an element in a queue runs in a constant time complexity. We will show the details of queue maintenance in Sections 3.3 and 3.5.

3.1 Pre-Processing (PP)

The edge list and mapping (*el* and *eid*) initialization in PKT [21] carries loop dependencies and results in sequential execution. It took minutes to finish the sequential execution on the twitter graph on an Intel Xeon CPU server. In contrast, our three-loop PP algorithm $PP(G)$ (Alg. 2) without any dependency can exploit parallelism. The input is an undirected graph stored in the CSR format with the row pointer and sorted adjacency arrays *rptr* and *adj*. Let us denote the set $\{v|v \in N(u) \wedge u < v\}$ by $N^+(u)$. We show the data structures of Alg. 2 in Fig. 4, including the CSR, vertex-related auxiliary structures, and the output edge list and mapping. We describe the three loops as follows.

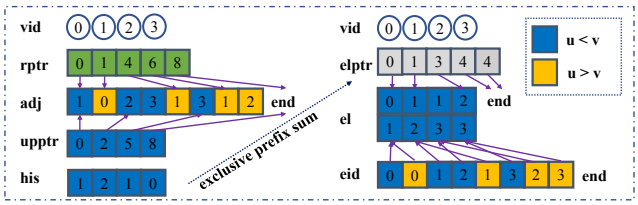


Figure 4: An example showing data structures in $PP(G)$

Firstly, we create auxiliary arrays *upptr* and *his* to record the start position of $N^+(u)$ in the *adj* adjacency array and $|N^+(u)|$ for each vertex u, respectively. Each $upptr(u)$ is found by a search on the sorted adjacency list of $N(u)$. Secondly, we compute an exclusive prefix sum on the *his* to get the write locations in the edge list *el* (denoted by *elptr*) for the edges with u as its source vertex. Thirdly, we go through each element in the *adj* array in a two-level loop, and record the position of $v \in N(u)$ as o_{uv}. When we process a vertex $v \in N^+(u)$, we (1) create the mapping $eid(o_{uv})$ by adding $elptr(u)$ to o_{uv}'s relative offset from the start position $upptr(u)$, and (2) assign $e(u, v)$ to the edge list (Lines 13-15). Otherwise, we invoke a binary search on

Algorithm 2: Pre-Processing ($PP(G)$)

Input: an undirected graph $G = (V_G, E_G)$
Output: an edge list el and an edge mapping eid

1 $el \leftarrow$ an array of size $|E|$, $eid \leftarrow$ an array of size $2|E|$
2 $his, upptr \leftarrow$ arrays of size $|V|$
3 $elptr \leftarrow$ an array of size $|V|+1$, $elptr(0) \leftarrow 0$
4 **foreach** $u \in V$ **do**
5 $\quad upptr(u) \leftarrow SearchGreater(adj, rptr(u), rptr(u+1), u)$
6 $\quad his(u) \leftarrow rptr(u+1) - upptr(u)$
7 **foreach** $u \in [0, |V|)$ **do** $elptr(u+1) \leftarrow elptr(u) + his(u)$
8 **foreach** $u \in V$ **do**
9 $\quad$ **foreach** $v \in N(u)$ **do**
10 $\quad\quad$ **if** $u > v$ **then**
11 $\quad\quad\quad o_{vu} \leftarrow SearchEqual(adj, rptr(v), rptr(v+1), u)$
12 $\quad\quad\quad eid(o_{uv}) \leftarrow elptr(v) + (o_{vu} - upptr(v))$
13 $\quad\quad$ **else**
14 $\quad\quad\quad ei \leftarrow elptr(u) + (o_{uv} - upptr(u))$, $eid(o_{uv}) \leftarrow ei$
15 $\quad\quad\quad el(ei) \leftarrow e(u, v)$
16 **return** the edge list el and the mapping eid

$N(v)$ to locate o_{vu} and create the mapping $eid(o_{uv})$ by using the edge with v as the source vertex (Lines 10-12). The first and last loops (Lines 4-6 and 8-15) perform the independent vertex-related computation, and the second loop (Line 7) can be parallelized by a two-pass prefix sum algorithm [33]. Thus, all three loops can be efficiently parallelized.

Time Complexity of $PP(G)$. Let d_{max} denote the maximum degree of G. Time complexity of $PP(G)$ is $O(|V| \cdot log(d_{max}))$ for $upptr$ and his initialization, $O(|V|)$ for prefix sum computation on $elptr$, and $O(|E| \cdot log(d_{max}))$ for the edge mapping and edge list creation in the third for-loop, where $O(log(d_{max}))$ is the complexity of a binary search on a sorted neighbor set (Lines 5 and 11 of Alg. 2).

3.2 Support Initialization (SI)

In this stage, we extend a bitmap-based all-edge triangle counting algorithm [11] (BMP-TC), to pack the bitmap of $N(u)$ into indexed non-zero 64-bit machine words and make use of the indexed words to perform word-wise multiple bitmap look-ups with a single operation.

Word Packing (WP). The neighbor set of a vertex u can be represented in a bitmap $B(u)$ of cardinality $|V|$, by setting each v'th bit ($\forall v \in N(u)$) in the $B(u)$ and leaving the other bits as zeros. In practice, many words of $B(u)$ contain all-zeros, especially for a u with a small $|N(u)|$ value. Therefore, we index the non-zero words of the bitmap $B(u)$, via storing a set of indexes for the non-zero words in $B(u)$, and keeping the associated set of the non-zero word content. We denote the word index and content of a vertex u by $W_I(u)$ and $W_C(u)$, respectively. The $W_I(u)(i)$'th word of $B(u)$ has the non-zero word content $W_C(u)(i)$. For example, given a system with 3-bit words, a packed structure of vertex 1, $W_I(1) = \{0, 1\}$ and $W_C(1) = \{0b101, 0b100\}$, means that vertex 1 has the neighboring vertices 0, 2 and 3.

As illustrated in Alg. 3, before BMP-TC, we pack the words of $B(u)$ for each vertex u and prepare the packed words $W_I(u)$ and $W_C(u)$ (Lines 1-4). Initially, we reset all the word indexes and contents for each vertex, and then test whether its neighborhood is dense enough for the word packing. Specifically, we compute a pack ratio $pr(u)$ of a vertex u as $|N(u)|$ divided by $|W_I(u)|$. Intuitively, the pack ratio represents the average percentage of non-zero bits in the packed words. To reduce the memory consumption, we allow users to input a neighbor set word-packing threshold wpt and only pack the neighbor set of a vertex u with a high pack ratio ($pr(u) > wpt$) into indexed non-zero words.

Triangle Counting (TC). For completeness, we give the

Algorithm 3: Support Initialization ($SI(G, wpt)$)

Input: an undirected graph $G = (V_G, E_G)$ and a neighbor set word-packing threshold wpt
Output: a support array sup

1 W_I, $W_C \leftarrow$ arrays of size $|V|$ for indexes and words
2 **foreach** $u \in V$ **do** $W_I(u) \leftarrow empty$, $W_C(u) \leftarrow empty$
3 **foreach** $u \in V$ ***and*** $|N(u)|/|W_I(u)| > wpt$ **do**
4 $\quad W_I(u), W_C(u) \leftarrow PackWords(N(u))$
5 **foreach** $u \in V$ **do**
6 $\quad B \leftarrow$ a bitmap of cardinality $|V|$
7 $\quad$ **foreach** $v \in N(u)$ **do** $SetBit(B, v)$
8 $\quad$ **foreach** $v \in N(u)$ ***and***
9 $\quad$ $((d(u) > d(v))$ ***or*** $(d(u) = d(v)$ ***and*** $u < v))$ **do**
10 $\quad\quad$ **if** $W_I(v)$ ***is*** $empty$ **then**
11 $\quad\quad\quad sup(e(u, v)) \leftarrow CountMatch(B, N(v))$
12 $\quad\quad$ **else**
13 $\quad\quad\quad sup(e(u, v)) \leftarrow CountMatch(B, W_I(v), W_C(v))$
14 $\quad$ **foreach** $v \in N(u)$ **do** $ClearBit(B, v)$
15 **return** the support array sup and the elapsed time

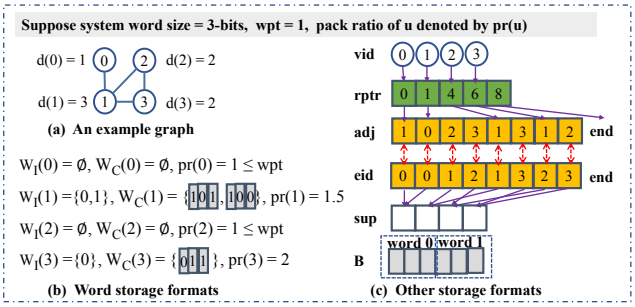


Figure 5: An example showing our word-packing technique

existing direct triangle counting algorithm [11] in Lines 5-14 of Alg. 3. Our word-packing-based triangle counting optimization is plugged into the algorithm at Line 13.

In BMP-TC, we loop over each vertex $u \in V$ and compute $|N(u) \cap N(v)|$ for each vertex $v \in N(u)$. In order to compute each $sup(e)$ exactly once, we add a degree-based ordering constraint (Line 9). For each vertex u, we (1) dynamically construct a bitmap B by setting v's bits for each $v \in N(u)$, (2) reuse the bitmap for the intersections $|N(u) \cap N(v)|$ ($\forall v \in N(u)$), and (3) reset the bits as all-zeros after finishing the computation on the vertex u.

Time Complexity of BMP-TC. The bitmap construction and clearing cost of $N(u)$ is in amortized constant time, since the bitmap is constructed once and reused for each $|N(u) \cap N(v)|$ ($v \in N(u)$) totaling $|N(u)|$ times. Thus, the computational cost is mainly from iterating each $w \in N(v)$, looking up $N(u)$'s bitmap, and counting the matches (Line 11). A match is a vertex w that is in both $N(v)$ and $N(u)$. Given the degree-based ordering (Line 9), for each $|N(u) \cap N(v)|$ computation, we perform $min(d(u), d(v))$ bitmap probe operations. Given the constant time complexity of bitmap operation cost, we have an $O(min(d(u), d(v)))$ time complexity for computing $|N(u) \cap N(v)|$.

In our WP-based BMP-TC (Line 13), we use the packed words to perform a single operation for multiple look-ups. Fig. 5 shows an example to illustrate this technique. To compute $sup(e(1, 3))$, we go through the indexed packed words of vertex 3, and directly look up the 0th word in the bitmap $B(1)$. We then perform a word-wise logical-and operation between $0b101$ (0th word of $B(1)$) and $0b011$ ($W_C(3)(0)$) to get $0b001$, and pop the count of 1-bits in the word to get the match count 1. After that, we assign the count 1 to $sup(e(1, 3))$. Without the packed words, we have to perform two look-ups and bit-masking operations to check the

existence of an element in the bitmap $B(1)$. This example shows that the WP technique reduces the workload.

3.3 Support Update (SU)

In the iterative edge peeling (IEP) stage, we operate on a mutable graph G and find each k-class Φ_k (Alg. 4). We use $P(e)$, which is initially $false$, to mark the removal status of an edge $e \in G$. Given l as the current level ($l = k-2$), for each remaining edge e, SU is to compute $max(l, sup(e, G - Q_C))$, assign it to the $sup(e)$, and maintain the edges to process in the queues Q_N and Q_I. We put an updated edge e into Q_N for then next iteration, once $sup(e)$ decrements to the current level l but is not in Q_N yet.

As discussed in Sect. 2.2, existing TPSU algorithms have two weaknesses: (1) unnecessary support update operations for the edges to be removed and (2) intensive random memory accesses on the edges of enumerated triangles. To address these weaknesses of TPSU, we propose to dynamically select triangle counting and peeling procedures (DSTCP) based on their estimated time (Lines 13-15 of Alg. 4).

The triangle peeling (TP) procedure, in Lines 20-24 of Alg. 4, loops over each $e(u,v) \in Q_C$ to perform triangle enumeration $N(u) \cap N(v)$ and update support and queues for each triangle $\triangle_{uvw}$. After the loop, all the edges in Q_C are marked as processed. Different from Alg. 1, the delayed edge removals incur a challenge to the design of a correct peeling procedure, in which each triangle $\triangle_{uvw}$ is peeled off exactly once, since both $e(u,v) \in Q_C$ and $e(u,w) \in Q_C$ can enumerate the same triangle $\triangle_{uvw}$ and trigger the support update of $e(v,w)$. For an edge $e(v,w)$, we decrement its support value only once for a given triangle $\triangle_{uvw}$.

The support update for each enumerated triangle is in Lines 25-34. Suppose a triangle exists after checking the flag P (Line 28). There are three conditions in peeling a single triangle. We denote the three edges in the triangle by e_1, e_2, and e_3, and consider three cases in peeling this triangle. (1) If all these edges are in Q_C, no support update is needed, since all three edges are to be removed and already in the k-class Φ_k (Line 27). Thus, we save the checking of both P and sup. (2) If we have $e_1 \in Q_C$ only and $e_2, e_3 \notin Q_C$, then the support update will be applied to both e_2 and e_3 (Line 30). (3) If we have two edges $e_x, e_y \in Q_C$ and $e_z \notin Q_C$, we choose the edge with the smaller edge ID in e_x and e_y to apply the update for e_z (Lines 31-34). In summary, the algorithm avoids the $P(e)$ and $sup(e)$ checking for an edge $e \in Q_C$ if all edges in the triangle are in Q_C, and ensures each triangle is peeled off exactly once.

The triangle counting (TC) procedure (Lines 35-39) peels the edges, compacts the adjacency lists and then invokes a similar procedure to the support initialization $SI(G, wpt)$ except (1) the support value is set to $max(sup(e), l)$ and (2) the queues Q_N and Q_I are updated. When a large number of triangles are peeled off in an iteration, the TC procedure is likely to have less workload than the TP procedure, because the edges of those triangles may mostly belong to Q_C, and the graph compaction in the TC procedure avoids unnecessary update for these edges. Also, the WP technique (discussed in Sect. 3.2) further reduces the workload of TC.

We estimate the time of the TC procedure via multiplying the support initialization time T_{SI} by the remaining portion of edges $(|E_G| - |Q_C|)/E_{G_0}$, and the time of the TP procedure via dividing the estimated triangle peeling workload $|\triangle_{Q_C}| = l \cdot |Q_C|$ by the estimated triangle peeling throughput ept: $T_{TC} = T_{SI} \cdot (|E_G| - |Q_C|)/E_{G_0}$, and $T_{TP} = l \cdot |Q_C|/ept$. If $T_{TC} > T_{TP}$, we choose the TP procedure; otherwise, we choose the TC procedure.

Algorithm 4: Optimized Truss Decomposition

Input: an undirected graph $G_0 = (V_{G_0}, E_{G_0})$, a neighbor set word-packing threshold wpt, a graph compaction threshold ct, an estimated peeling throughput ept, and an index range size rs
Output: all the i-classes Φ_i of G_0 ($i \geq 2$)

```
1  G ← G_0, el, eid ← PP(G), sup, T_SI ← SI(G, wpt)
2  k ← 1, n_rm ← 0, foreach e ∈ E do P(e) ← false
3  while |E| > 0 do
4      k ← k + 1, Φ_k ← ∅, l ← k − 2
5      if (l mod rs) = 0 then
6          Q_I ← {e | e ∈ E ∧ sup(e) ∈ [l, l + rs)}
7      Q_C ← {e | e ∈ Q_I ∧ sup(e) = l}, Q_N ← ∅
8      while |Q_C| > 0 do
9          Φ_k ← Φ_k ∪ Q_C
10         if |Q_C| = |E| then break
11         if n_rm > ct then CompactAdj(G, P), n_rm ← 0
12         if k = 2 then PeelEdges(P, Q_C, n_rm)
13         else if T_SI · (|E_G| − |Q_C|)/E_G0 < l·|Q_C|/ept then
14             CountTri(G, P, Q_C, Q_N, n_rm, wpt)
15         else PeelTri(G, P, Q_C, Q_N, n_rm)
16         Swap(Q_C, Q_N), Q_N ← ∅
17 return {Φ_i | 2 ≤ i ≤ k}
18 Procedure PeelEdges(P, Q_C, n_rm)
19     foreach e ∈ Q_C do P(e) ← true, n_rm ← n_rm + 1
20 Procedure PeelTri(G, P, Q_C, n_rm)
21     foreach e(u, v) ∈ Q_C do
22         foreach w ∈ N(u) ∩ N(v) do
23             PeelSingleTri(△_uvw, P, Q_C, Q_N, Q_I)
24     PeelEdges(P, Q_C, n_rm)
25 Procedure PeelSingleTri(△_uvw, P, Q_C, Q_N, Q_I)
26     e_1 ← e(u, v), e_2 ← e(u, w), e_3 ← e(v, w)
27     if (e_2 ∉ Q_C or e_3 ∉ Q_C) then
28         if not P(e_2) and not P(e_3) then
29             if e_2 ∉ Q_C and e_3 ∉ Q_C then
30                 Op(e_2, sup, Q_N, Q_I), Op(e_3, sup, Q_N, Q_I)
31             else if e_3 ∈ Q_C and e_1 < e_3 then
32                 Op(e_2, sup, Q_N, Q_I)
33             else if e_2 ∈ Q_C and e_1 < e_2 then
34                 Op(e_3, sup, Q_N, Q_I)
35 Procedure CountTri(G, P, Q_C, n_rm, wpt)
36     PeelEdges(P, Q_C, n_rm), CompactAdj(G, P), n_rm ← 0
37     SI(G, wpt)
38     foreach e ∈ G do
39         sup(e) ← max(sup(e), l), UpdateQ(Q_N, Q_I, sup)
```

3.4 Triangle Enumeration (TE)

In our TP-based support update (Lines 20-24 of Alg. 4), we conduct TE on each edge $e(u,v) \in E$ in the current queue Q_C to find the edges whose support values are changing. We propose three techniques to reduce the number of triangle existence checking operations in TE: (1) using a pivot-skip merge algorithm to handle the data skew, (2) compacting the graph periodically to reduce false-positives and save the P checking cost, and (3) skip the unnecessary enumeration.

Pivot Skip Merge (PSM). TE in truss decomposition is different from a traditional one. Traditionally, for an edge $e(u,v)$, we enumerate each triangle $\triangle uvw$ that $e(u,v)$ is in by recording a triplet (u,v,w). In truss decomposition, to facilitate SU, we further find the offsets of w in the adjacency lists $N(u)$ and $N(v)$, and map the offsets to the edge list domain using the edge mapping array. In our neighbor set intersection, we store the offsets related to $e(u,w)$ and $e(v,w)$ instead of the vertex ID w. PSM of two sorted arrays $N(u)$ and $N(v)$ is shown with an example in Fig. 6, where u and v have a large and small degree, respectively.

In PSM, we initialize the pivot values w and w' in $N(v)$ and $N(u)$ to be the first element of the corresponding adja-

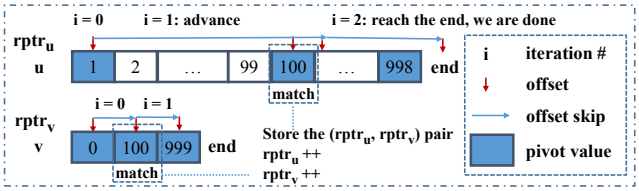


Figure 6: An example showing our pivot-skip merge

cency list. Our merge iterates in three steps. Firstly, we use the pivot value w in $N(v)$ and skip in $N(u)$ to locate the first element $w' \geq w$, which is the next pivot value in $N(u)$. The skip is implemented via a galloping search, which skips the offset $rptr_u$ at increasing sizes $0, 2^0, 2^1, ..., 2^i, 2^{i+1}$ until we reach the end of an element $tmp > w$, and then locates the exact position with a binary search in the skip area $[2^i, 2^{i+1})$. Secondly, we use the pivot value w' in $N(u)$ and skip in $N(v)$ to locate the first element $w \geq w'$. Thirdly, we test whether there is a match of the pivot values of the two lists. If a match occurs, we store the offset pair $(rptr_u, rptr_v)$ into a result array and increment the offsets $rptr_u$ and $rptr_v$.

Time Complexity of PSM. Suppose two vertices u and v and $d(v) < d(u)$, the skip cost in $N(v)$ is bounded by $|N(v)|$, i.e., $O(d(v))$. Let $s[i]$ denote the skip size in $N(u)$. In each skip step, the galloping skips and the binary search both take the $O(log(s[i]))$ time complexity, and there are at most $2 \cdot d_v$ iterations to advance the offset of v's neighbor set to the end. We sum up the cost of each skip step in $N(v)$, add the total skip cost in $N(u)$, and get the time complexity $O(\Sigma_{i \in [0, d(v))} \log(s[i]) + d(v))$. In practice, the average logarithms of skip size $(\Sigma_{i \in [0, d(v)} \log(s[i]))/d(v)$ is small. Thus, the computational cost is $O(min(d(u), d(v)))$.

Graph Compaction (GC). After many edges are peeled off, the checking of triangle existence and processing status P on these removed edges is unnecessary, because they do not trigger any support update. To eliminate this checking, we compact the adjacency list and the edge mapping arrays when the accumulated number of edge removals n_{rm} is greater than a user-specified threshold ct. This threshold is set in consideration of the compaction overhead and the benefit of the memory access reduction in TE.

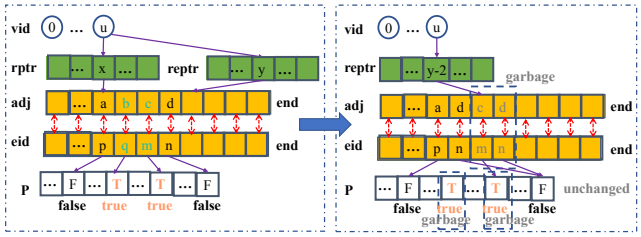


Figure 7: An example showing our graph compaction

We show the idea of GC in Fig. 7. To track each $N(u)$ after the compaction, we introduce an array $reptr$ to record the end-of-row pointer for each adjacency list. The pointers $rptr(u)$ and $reptr(u)$ represent the start and end positions of $N(u)$ in the adjacency array adj. Initially, $reptr(u)$ is assigned $rptr(u+1)$. The $reptr(u)$ is updated after a GC invocation (Line 11 of Alg. 4) whereas $rptr(u)$ remains unchanged. In GC, for a vertex u, we loop over each $v \in N(u)$ and utilize the edge mapping and the processing flag arrays to determine whether a vertex v should be removed from $N(u)$. If v should not be removed, we assign the corresponding neighbor (adj) and edge mapping (eid) values to the next write location nwl, and then increment nwl; otherwise, we take no operation. Finally, we update $reptr(u)$ to nwl, which serves as the new end position of $N(u)$.

Time Complexity of GC. Suppose G_0 is the input graph, and G is the compacted graph. The time complexity of GC is $O(|E_{G_0}|)$, since we perform GC at most ct (constant) times, and touch $O(|E_G|)$ elements each time.

Enumeration Skipping (ES). We can safely skip the triangle enumeration in the first and last iterations. In the first iteration (Line 12, $k = 2$, $l = 0$), no triangle contains an edge in Q_C, since all edges have a support value of zero. Thus, we mark all the edges $e \in Q_C$ as processed and proceed to the next level without any update of the empty Q_N. In the last iteration, the number of remaining edges $|E|$ is equal to the current queue size $|Q_C|$ (Line 10). It indicates no edge remains after the iteration, so no further support update is required. We can terminate early right after constructing the last k-class Φ_k.

3.5 Edge Filtering (EF)

As discussed in Sect. 3.3, during the support update (SU), we maintain Q_N in both the TC and TP procedures to track the edges with support value l. At the end of an iteration, we put the content of Q_N into Q_C, and then clear Q_N to prepare for the next iteration in level l (Line 16). However, for the first iteration in level l, we must scan all edges (E_{G_0}) to filter the edges with the support value l.

We profile the execution and find that for a specific level, the selectivity ($|Q_C|/|E_{G_0}|$) may be low. Thus, we propose to track the edges with support values $sup(e) \in [beg, beg + rs)$. Initially, beg is 0, and rs is a parameter. The variable beg is incremented by rs, and a reconstruction of the index is required once every rs levels (Lines 5, 6 of Alg. 4). With the index Q_I, for the edge filtering (Line 7 of Alg. 4), we scan each edge $e \in Q_I$ to compute Q_C, and scan $e \in E_{G_0}$ once every rs levels to reconstruct both Q_I and Q_C. SU triggers the update of Q_I. Specifically, when e's support value $sup(e) \in [beg, beg + rs)$, but $e \notin Q_I$, we add e to Q_I.

Time/Memory Complexity of EF. The memory cost of Q_C, Q_N, and Q_I is bounded by $O(|E|)$, since we add each edge to the queues only once. The computational cost of checking $e \in Q_N$ and $e \in Q_I$ is $O(|\triangle|)$, since each enumerated triangle triggers queue update operations. The computational cost of queue update is $O(|E|)$ since each edge is put into the queues only once. When a large number of triangles are peeled-off, our TC procedure (Line 14 of Alg. 4) reduces $O(|\triangle|)$ checking cost ($e \in Q_N$ and $e \in Q_I$) to $O(|E|)$, since we do the checking only once for each edge $e \in Q_C$. When $|\triangle|$ is much larger than $|E|$, most triangles are often peeled off in only a small number of iterations. Thus, for those cases, the time complexity of our index maintenance can be regarded as $O(|E|)$ instead of $O(|\triangle|)$.

Difference from Existing Techniques. MSP [20] indexes the edges by grouping the edges with identical support values into a bucket (MSP-IDX), whereas we only maintain a single bucket of edges, support values of which are in a range. Julienne [16, 32] provides a similar bucketing structure to MSP-IDX. The computational cost of MSP-IDX is $O(|\triangle|)$, since each triangle affects the bucket update of two edges. The memory cost of MSP-IDX is $O(|\triangle|)$, since all triangles are expanded and grouped by edges for lock-free bucket maintenance [20]. Our approach has a memory complexity $O(|E|)$, lower than $O(|\triangle|)$ of MSP-IDX. In practice, the computational cost of checking $e \in Q_I$ and $e \in Q_N$

in our approach is less than that of updating the bucket structure in MSP-IDX. Moreover, our TC procedure further reduces the checking cost for intensive triangle peeling cases.

3.6 Algorithm Analysis

Memory Complexity. An input graph *rptr* and *adj* arrays in the CSR format is in $O(|E|+|V|)$ space. Auxiliary arrays *upptr*, *his*, *elptr*, and *reptr* for pre-processing and graph compaction are in $O(|V|)$ space. Packed words W_I and W_C and a dynamically constructed bitmap B for support initialization are in $O(|E|+|V|)$ space. Edge list, edge mapping, support values, and processing flag arrays *el*, *eid*, *sup*, and P are in $O(|E|)$ space. Queues Q_C, Q_N, and Q_I for edge filtering are in $O(|E|)$ space. Therefore, the total memory complexity of our algorithm is $O(|E|+|V|)$.

Time Complexity. As discussed in Sect. 3.1, the PP stage is in $O((|V|+|E|)\cdot log(d_{max}))$ time. SI's computational cost is dominated by BMP-TC. As described in Sect. 3.2, each $|N(u)\cap N(v)|$ computation is in $O(min(d(u),d(v)))$ time complexity. Given this complexity, BMP-TC is in $O(|E|^{1.5})$ time, as proved in prior triangle listing work [35].

Computational cost of IEP consists of triangle enumeration (TE), $O(|\triangle|)$ support value update (SU), and $O(|\triangle|)$ edge filtering (EF) time. We utilize PSM for TE, which has a time complexity $O(min(d(u),d(v)))$ (proved in Sect. 3.4). Given this $O(min(d(u),d(v)))$ complexity, TE is in $O(|E|^{1.5})$ time [38]. This is because the overall time complexity bound of TE is in $\sum_{e\in E} min(d(u),d(v)) = O(\alpha\cdot|E|)$, where α is the arboricity of the graph [13]. In the worst case, this bound is $O(|E|^{1.5})$ since $\alpha\leq\sqrt{|E|}$ [13]. We further reduce the TE workload via GC and ES techniques in $O(|E|)$ and constant time, respectively. As discussed in Sect. 3.5, with our TC-based SU procedure, when most triangles are peeled off in several iterations, the time complexity of EF is $O(|E|)$ instead of $O(|\triangle|)$. Therefore, the total time complexity of our algorithm is $O(|E|^{1.5})$. Even though the time complexity bound is the same as prior work [38], our optimizations significantly reduce the practical workload.

Extension. TC in SI can be extended to other TC methods, as long as the TC result $|N(u)\cap N(v)|$ is written back to $sup(e(u,v))$. TE $N(u)\cap N(v)$ can also be replaced with other set intersection methods used in TC, provided that the offsets of matched elements in the adjacency array are recorded, which are used for subsequent update operations of Q_N and Q_I. The support re-initialization $SI(G,wpt)$ (Line 37 of Alg. 3) in the TC-based SU procedure (Lines 35-39 of Alg. 3) can also be extended with similar methods to those in SI. However, support value increment and queue update should be performed after or during TC (Line 39 of Alg. 3) for the correct support value update and indexing.

4. IMPLEMENTATION

In this section, we describe the parallelization of our algorithms, techniques to offload some computation to the GPU, and more optimization techniques on the CPUs.

4.1 Parallelization on the CPUs

We exploit the parallelism in the outermost foreach-loop of nested loops in Alg. 2-4 and use OpenMP [3] to implement them. We have described the parallel pre-processing in Sect. 3.1. In the support initialization, the word-packing and triangle counting loops carry no dependency; we allocate a local bitmap for frequent reuse in a single thread. In the graph compaction, we parallelize the independent computation on vertices whose neighbor sets are updated. The update status of a vertex is recorded with a boolean array and is updated in the edge peeling related to the vertex.

To handle concurrent support update and dynamic queue maintenance, we adopt gcc built-in atomic primitives [2]. Specifically, we adopt an atomic compare-and-swap (CAS) primitive, which compares the value in a memory address with the target value and swap the value in the memory with a new value only when the compared and target values are equal. We also use atomic fetch-and-add and fetch-and-subtract primitives, each of which consists of read, modify, and write steps as an atomic operation.

Update of *sup* and Q_N. Given the atomic primitives, to ensure the correct support value update and unique enqueueing in Q_N for an edge e (Lines 30 and 39 of Alg. 4), we further introduce a concept *token* and a roll-back operation of $sup(e)$. The thread who reads the value $l+1$ in an atomic fetch-and-subtract operation of $sup(e)$ is regarded as having the token, and only this thread with the token enqueues e to Q_N. However, other threads may decrement $sup(e)$ to produce a $sup(e)<l$. To roll back the over-subtractions and increment $sup(e)$ to l, we execute atomic-add instructions for those threads. To enqueue edge e into Q_N, we atomically increment a global variable sz storing the size of Q_N, and write e into the sz position of Q_N. Recall that we have a boolean array of size $|E|$ to record the occupation status of Q_N. We directly set e's flag in the array without contention, as only the thread with the token can update e's status.

Update of Q_I. When $sup(e)$ is in the range $[beg, beg+rs)$, we update the index structure Q_I. However, given $rs>1$, multiple threads may update Q_I for the same edge e concurrently. Therefore, we adopt the CAS operation on the memory address of the occupation flag for $e\in Q_I$, with the target value $false$ and the new value $true$. The thread that successfully sets the flag as $true$ is regarded as having the token and can further perform an enqueue operation on Q_I. The enqueue operation of Q_I is similar to Q_N. We implement the same operation of Q_I in both the support update and edge filtering (Line 6 of Alg. 4) procedures.

4.2 Offloading Techniques

The Nvidia GPU has tens of Streaming Multiprocessors (SMs), on which hundreds of thread blocks can be executed simultaneously. Because the SI and IEP stages are computation-intensive, we offload them to GPU by launching CUDA (Compute Unified Device Architecture) kernel programs. A kernel runs in a grid of thread blocks, which are scheduled on the SMs. Recently, a unified memory technique (UM) is supported, which provides a unified virtual memory address space for both CPUs and GPUs. Memory pages are transferred on-demand on page faults in UM.

Implementation Overview. We utilize UM to allocate the data structures addressable by both CPUs and GPUs, and design kernels to parallelize the loops in the SI and IEP stages. In addition, we use the parallel primitive library CUB [1], including histogram, prefix sum, and selection kernel function templates, in GC and EF implementation.

In the SI stage, we map each word-packing task to a GPU thread, set the block size to 4 warps (128 threads) to achieve full occupancy (16 concurrent blocks on an SM). The number of thread blocks in the grid is $\lceil|V|/128\rceil$ thread blocks.

In the TC loop of SI, we allocate SM local storage to store

the bitmaps and use a flag array to mark the occupation of each bitmap. We map TC tasks $sup(e(u,v))$ computation between a vertex u and each $v \in N(u)$ to a thread block, and each $|N(u) \cap N(v)|$ computation to a warp in the thread block. The bitmap is acquired and released by a single thread in the block via CAS operations to get the token for the corresponding bitmap. After the bitmap acquisition, the bitmap is constructed by all the threads in the block using atomic-or operations. We synchronize the threads in the same block for the bitmap acquisition, construction, and release. There are in total $|V|$ thread blocks for TC.

In TP-based SU, we adopt binary search instead of galloping search to eliminate loop-carried dependencies since we exploit fine-grained parallelism within each SM. We map each processing task of an edge $e(u,v) \in Q_C$ into a thread block. Threads in the same block loop over $v \in N(u)$ $(d(v) < d(u))$ and perform binary search on each v.

In the edge filtering, we parallelize the initialization of the Q_I slot occupation flag array, and then invoke a selection CUB kernel to filter the edges in a support value range $[beg, beg + rs)$. We perform Q_C filtering from Q_I similarly.

CPU-GPU Co-Processing. When data structures exceed the GPU memory capacity size, the intensive random memory accesses in the IEP stage cause many page swaps and result in the under-utilization of SMs. Thus, we propose two CPU-GPU co-processing strategies to tackle the problem. First, we can offload the edge peeling iterations to the GPU at the beginning, periodically compact adjacency lists (CSR), edge mapping, edges list, support values, and queues, and record the mapping to the original edge offsets (E_{G_0}). Alternatively, we can perform the edge peeling phases of the first few levels on the CPU, compact all the data structures once on the CPU, and offload the remaining peeling iterations to the GPU at the beginning of a k-class finding phase when the compacted storage is within the GPU memory capacity. Subsequently, we compact only the CSR and edge mapping arrays on the GPU.

4.3 Optimizations

First, we dynamically select the pivot-skip merge (PSM) and a vectorized block-wise merge (VM) [11] based on a data skew ratio $d(u)/d(v)$ (suppose $d(u) > d(v)$). The VM procedure incurs more comparison operations but processes more data in each CPU cycle [11]. We exploit AVX-512 instructions in VM via intrinsic functions on the Intel Xeon CPU. Second, we introduce local write buffers for the queue maintenance to reduce the number of atomic-add instructions at the cost of copying the content from the local buffers to the global queues. An alternative concurrency control mechanism is hardware transactions; unfortunately the contention is high on these global queues and therefore makes hardware transactions unsuitable. Last, we store the processing flag array P (described in Sect. 3.3) in a bitmap that supports the atomic set and unset operations to enable the bitmap-based filtering (BMPF) word by word, which reduces the workload of edge filtering when most edges are processed.

5. EVALUATION

In this section, we evaluate the effectiveness of individual techniques on the CPUs, the offloading techniques on both CPUs and GPUs, and scalability of our algorithms on large datasets. We compare our optimized algorithms with others and summarize experimental findings.

Table 2: Statistics of real-world and synthetic graphs

Dataset	$\|V\|$	$\|E\|$	$\|\triangle\|$	max-$d(u)$	max-$\phi(e)$
orkut (OR)	$3.1 \cdot 10^6$	$1.2 \cdot 10^8$	$6.3 \cdot 10^8$	$3.3 \cdot 10^4$	78
web-uk (WU)	$1.9 \cdot 10^7$	$1.5 \cdot 10^8$	$2.2 \cdot 10^9$	$1.7 \cdot 10^5$	944
web-eu (WE)	$1.1 \cdot 10^7$	$1.9 \cdot 10^8$	$3.4 \cdot 10^{11}$	$1.8 \cdot 10^5$	9,667
webbase (WB)	$1.2 \cdot 10^8$	$5.3 \cdot 10^8$	$6.9 \cdot 10^9$	$8.0 \cdot 10^5$	1,226
web-it (WI)	$4.1 \cdot 10^7$	$5.8 \cdot 10^8$	$2.4 \cdot 10^{10}$	$1.2 \cdot 10^6$	3,210
twitter (TW)	$4.2 \cdot 10^7$	$6.8 \cdot 10^8$	$2.4 \cdot 10^{10}$	$1.4 \cdot 10^6$	1,517
s22-16	$4.2 \cdot 10^6$	$6.4 \cdot 10^7$	$2.1 \cdot 10^9$	$1.6 \cdot 10^5$	543
s23-16	$8.4 \cdot 10^6$	$1.3 \cdot 10^8$	$4.7 \cdot 10^9$	$2.6 \cdot 10^5$	718
s24-16	$1.6 \cdot 10^7$	$2.6 \cdot 10^8$	$1.0 \cdot 10^{10}$	$4.1 \cdot 10^5$	936
s25-16	$3.4 \cdot 10^7$	$5.2 \cdot 10^8$	$2.3 \cdot 10^{10}$	$6.4 \cdot 10^5$	1,203
s26-16	$6.7 \cdot 10^7$	$1.1 \cdot 10^9$	$4.9 \cdot 10^{10}$	$1.0 \cdot 10^6$	1,522
s27-16	$1.3 \cdot 10^8$	$2.1 \cdot 10^9$	$1.1 \cdot 10^{11}$	$1.6 \cdot 10^6$	1,913
s28-16	$2.7 \cdot 10^8$	$4.2 \cdot 10^9$	$2.3 \cdot 10^{11}$	$2.5 \cdot 10^6$	2,359
s29-16	$5.4 \cdot 10^8$	$8.5 \cdot 10^9$	$5.0 \cdot 10^{11}$	$3.8 \cdot 10^6$	2,879

5.1 Experimental Setup

On the CPUs, we evaluate our individual techniques in the three stages of the truss decomposition: pre-processing (PP), support initialization (SI), and iterative edge peeling (IEP). Our baseline is the parallel PKT implementation [21].

We first evaluate our parallelization (**P**) of the PKT-PP stage, and then evaluate the following techniques to improve the PKT-SI stage: (i) the direct triangle counting (**DTC**) and (ii) our word-packing (**WP**) based DTC. We next evaluate our techniques for the most time-consuming IEP stage. Specifically, we start from the PKT-IEP and enable our techniques one-by-one in the following order: (i) the pivot-skip merge (**PSM**), vectorized merge (**VM**), graph compaction (**GC**) and enumeration skipping (**ES**) in the triangle enumeration, (ii) the dynamic selection of triangle counting and peeling procedures (**DSTCP**) in the support update, and (iii) the bitmap filtering of P (**BMPF**) and the indexing of Q_I (**IDX**) in the edge filtering. After that, we further evaluate the scalability to the number of threads n_t and break the time of our optimized truss decomposition into five components: the PP and SI stages, and three components of the IEP stage (i) graph compaction (GC), (ii) support update (SU), and (iii) edge filtering (EF). Moreover, we discuss the effect of four parameters: (i) wpt for WP, (ii) ct for GC, (iii) ept for DSTCP and (iv) rs for EF.

On the heterogeneous processors, we first evaluate the effect of offloading the SI stage to the GPU (**OFF-SI**). We then analyze the difference between our two strategies in the IEP stage: (i) offloading the entire IEP stage to the GPU (**OFF-EIEP**) and (ii) computing the first few k-classes, compacting the storage once on the CPU and then offloading the remaining computation to the GPU (**OFF-RIEP**).

Finally, we evaluate the scalability of algorithms on large datasets and compare our optimized implementations on the CPUs (OPT-CPU) and heterogeneous processors (OPT-HPU) with (1) PKT [21], (2) MSP [34], (3) H-IDX [30], and (4) our enhanced H-IDX (H-IDX+) with the data skew handling and edge mapping techniques on six graphs.

Environments. We conduct experiments on a Linux server with an Nvidia V100 GPU. The server has two 10-core 2.4GHz Intel Xeon Gold 5115 CPUs. The L1, L2, L3 cache, and DRAM sizes of the server are 64KB, 1024KB, 13.75MB, and 256GB, respectively. The Nvidia GPU has 80 SMs and 64 cores per SM. We obtain the implementations of PKT [21], MSP [34], H-IDX [30] from the authors and implement our algorithms and H-IDX+ on the CPUs in C++. We compile all the algorithms with g++ 7.3.1. We implement our GPU algorithms in CUDA 9.2 and compile them with nvcc 9.2.88 with -O3 option. The source code for the experiments is publicly available [12].

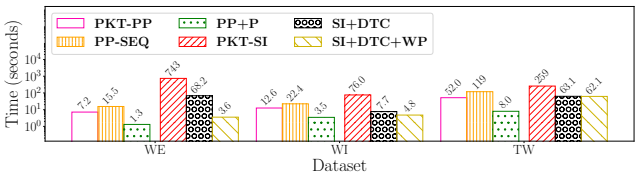


Figure 8: Effect of the techniques for the PP and SI stages

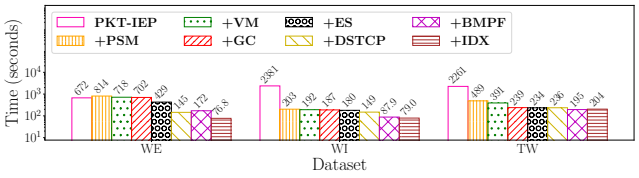


Figure 9: Effect of the techniques for the IEP stage

Datasets and Parameters. We select six representative real-world graphs from SNAP [22] and WebGraph [6,7]. We remove self-loops and multi-edges of directed graphs from WebGraph to obtain undirected graphs for truss decomposition. We generate synthetic RMAT graphs via open-source tools [9,10]. In RMAT graph generation, we set the average degree to 32 and a scale factor $s \in [22, 30)$ to vary $|E|$ from 64 million to 8 billion. Statistics of real-world and synthetic RMAT graph are listed in Table 2. By default, we set the input parameters as follows for best performance: number of CPU threads $n_t = 40$, word-packing threshold $wpt = 2$, graph compaction threshold $ct = 100$, edge-peeling throughput $ept = 2 \cdot 10^9$, and range size $rs = 16$.

Metrics. We run each experiment three times and report the average in-memory processing time. The time variance from the average is less than 5%. In the time breakdown of the IEP stage, we report the accumulated time of all the iterations for GC, SU, and EF. In the evaluation of our offloading strategies, we use an nvprof tool to profile the amount of data transfer and the transfer time between processors.

5.2 Evaluation of Individual Techniques

Effect of the Techniques for the PP and SI Stages. We first compare our parallel pre-processing PP+P with the baseline PKT-PP. Our PP+P has $O(|V| + |E|)$ search overhead in the computation of the histogram and reverse edge finding. However, the benefit of parallel execution offsets the overhead. Results (the three bars on the left in Fig. 8) show that we reduce the PP time by up to 6.5x.

We then evaluate the effect of direct triangle counting (DTC) and word-packing (WP) in the support initialization (SI) stage. Results (the three bars on the right in Fig. 8) show that DTC achieves speedups of 10.9x, 9.9x, and 4.1 over the baseline PKT-SI on WE, WI, and TW, respectively. DTC speeds up less on TW than on the web graphs because triangle enumeration takes more time for mismatches than that for matches on TW. Our word-packing further improves SI-DTC by performing multiple word-wise look-ups in a single operation, which achieves a speedup of 18.9x on WE. The large speedup is because WE has dense local structures for WP. The effect of WP is less significant on WI and TW than on WE because of its relatively low pack-ratio, i.e., the ratio of $|N(u)|$ over $|W_I(u)|$ (discussed in Sect. 3.2).

Effect of the Techniques for the IEP Stage. We evaluate the techniques for the triangle enumeration, support update, and edge filtering components in Fig. 9.

We first evaluate the techniques for the triangle enumeration (TE). Pivot-skip merge (PSM) speeds up by 11.7x on

Table 3: Effect of PSM

	Number of Invocation		Number of Comparison	
Dataset	VM	PSM	VM-only	VM+PSM
WE	$1.73 \cdot 10^8$	$1.83 \cdot 10^7$	$2.49 \cdot 10^{12}$	$2.16 \cdot 10^{12}$
WI	$4.14 \cdot 10^8$	$1.69 \cdot 10^8$	$2.55 \cdot 10^{13}$	$4.80 \cdot 10^{11}$
TW	$4.64 \cdot 10^8$	$2.21 \cdot 10^8$	$4.05 \cdot 10^{13}$	$2.5 \cdot 10^{12}$

Table 4: Effect of GC (seconds)

	GC Disabled		GC Enabled		
Dataset	T_{GC}	T_{SU}	T_{GC}	T_{SU}	$T_{IEP} \downarrow$
WE	0.0	643.2	1.4	625.5	16.3
WI	0.0	111.3	5.7	104.8	0.7
TW	0.0	347.4	11.6	185.2	150.7

TW and WI but slows down the execution on WE. This different performance impact is because PSM handles data skew ($d(u) \gg d(v)$, given vertices u and v) at the cost of a more irregular memory access pattern; both WI and TW have data skew, whereas WE does not. The PSM optimization is beneficial when a degree ratio $d(u)/d(v)$ is large because of the large reduction on comparison operations. In contrast, vectorized merge (VM) is good for the intersections without data skew, and exploits vectorized instructions, e.g., AVX-512. As a result, VM reduces up to 100 seconds for the WE and TW datasets when there are sufficient invocations. We set the degree ratio threshold to 50 and select PSM for cases with data skew ($d(u)/d(v) > 50$) and VM otherwise. We show the number of invocations of these two functions (VM/PSM) and the number of comparisons with and without PSM in Table 3. The results show that PSM drastically reduces the number of comparisons on WI and TW (> 16x) but not as much reduction on WE. Also, there are more PSM invocations on WI and TW than on WE.

Graph compaction (GC) reduces the memory accesses of the triangle enumeration at the cost of compaction. We show the time of graph compaction (GC), support update (SU), and iterative edge peeling (IEP) in Table 4. The results in Fig. 9 (+GC bar) show that GC achieves a speedup of 1.6x on TW but has less impact on WE and WI. The difference is because, on WE and WI, most edges with a lot of connections are peeled off at later levels, in which case GC takes effect at later several levels on the two web graphs. Moreover, enumeration skipping (ES) eliminates the triangle enumeration of the first and last iterations. In particular, in Fig. 9 (+ES bar), ES shows a large improvement on WE because there is a big clique of 9,667 edges peeled off at the last iteration, which contains many triangles.

We next evaluate the effect of the dynamic selection of triangle counting and peeling procedures (DSTCP). DSTCP achieves a speedup of about 3x on WE and a moderate improvement on WI and TW. The difference is because, on WE, three iterations in total in the two levels ($l = 3700$ and $l = 9584$) take in total hundreds of seconds if we only use the triangle peeling (TP) procedure. With the triangle counting (TC) procedure, less than five seconds are spent on these iterations. This large performance gap comes from three factors. (1) There are trillions of atomic update operations for the three iterations in the TP-only design. (2) The graph compaction in TC eliminates accesses to trillions of triangles, since most of the enumerated triangles are formed by edges $e \in Q_C$. (3) The TC procedure has a workload proportional to our WP-based SI. We profile the number of last level cache (LLC) loads and misses via a perf tool and show the results in Table 5. DSTCP improves performance by reducing the number of LLC loads instead of the cache miss ratio. The LLC load reduction is from (1) the reduc-

Table 5: Effect of DSTCP

	Number of LLC Loads		Number of LLC misses	
Dataset	TP-only	+DSTCP	TP-only	+DSTCP
WE	$1.06 \cdot 10^{12}$	$1.87 \cdot 10^{11}$	$9.27 \cdot 10^{10}$	$2.01 \cdot 10^{10}$
WI	$1.02 \cdot 10^{11}$	$8.23 \cdot 10^{10}$	$2.02 \cdot 10^{10}$	$1.81 \cdot 10^{10}$
TW	$2.20 \cdot 10^{11}$	$2.33 \cdot 10^{11}$	$9.65 \cdot 10^{10}$	$9.81 \cdot 10^{10}$

Table 6: Effect of BMPF and IDX on the CPUs (seconds)

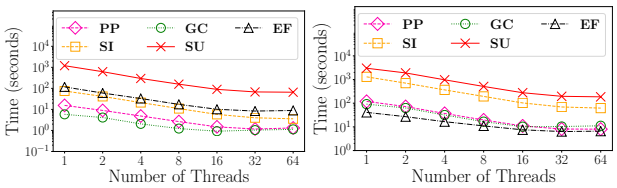
	Both Disabled		BMPF Enabled			BMPF+IDX Enabled		
Dataset	T_{EF}	T_{SU}	T_{EF}	T_{SU}	$T\downarrow$	T_{EF}	T_{SU}	$T\downarrow$
WE	73.1	67.1	98.5	69.2	-27.6	8.9	65.7	**65.5**
WI	70.7	71.6	11.5	70.4	60.4	5.5	67.8	**68.9**
TW	42.2	181.9	10.4	172.8	**40.9**	6.6	185.8	31.7

tion of support value update and queue maintenance operations and (2) workload reduction from our word-packing technique in TC. The performance improvement of DSTCP is significant when a massive number of triangles are peeled off in only a few iterations, such as on the WE dataset.

Next, we evaluate the bitmap filtering (BMPF) and indexing (IDX) optimization for the edge filtering (EF). We show the edge filtering time (T_{EF}), the support update time (T_{SU}), and the overall time reduction ($T\downarrow$) of BMPF and IDX in Table 6. On WE, BMPF slows down the execution; in contrast, on WI and TW, it reduces the EF time by 6.1x and 4.0x, respectively. This is because, on WE, most edges are removed after level 9584, which is close to the maximum truss number of WE; in comparison, on WI and TW, BMPF takes effect much earlier. Finally, IDX improves EF by reducing the number of scanned edges at the cost of maintaining the index Q_I. It performs well on the web graphs but is not the best choice on TW. The reason is that, on TW, the index maintenance cost offsets the time reduction of EF; whereas on the web graphs, the index maintenance cost is reduced to $O(|E|)$ in practice by the DSTCP technique.

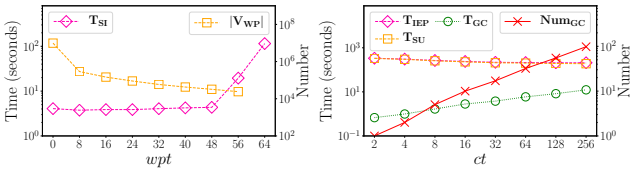
Scalability to Number of Threads. We vary the number of threads $n_t \in \{1, 2, 4, 8, 16, 32, 64\}$ and report the time of the five components of truss decomposition in Fig. 10 on WE and TW. The support update (SU) is the dominant component on both datasets, and its parallelization with 64 threads achieves speedups of 18.2x and 16.1x over the sequential execution on WE and TW, respectively. The parallelization of support initialization (SI) achieves speedups of over 20x on both datasets and takes the second and third-longest time on WE and TW, respectively. The edge filtering (EF) scales better on WE than on TW. This is because a task on TW contains less workload than on WE since more edges are processed at early levels, which makes the dynamic scheduling cost more significant than that on WE. The preprocessing (PP) scales moderately on both datasets, having speedups of up to 13.2x and 14.6x on WE and TW, respectively. The graph compaction (GC) has the lowest speedup due to random memory accesses and load imbalance. Nevertheless, GC takes much less time than SI and SU.

Effect of Parameter Setting. We fix a representative dataset and vary each parameter. In Fig. 11a, we find that even at a very large $wpt = 48$, we can still keep a good performance when packing a smaller but denser group of neighbor sets. This phenomenon suggests that the benefit of word-packing comes from dense neighbor sets, and we can choose a suitable value of wpt, e.g., 4 to limit memory consumption. Fig. 11b shows that (i) as the maximum number of GC invocations ct increases, the overhead of compaction increases and the SU time decreases; (ii) the performance does not improve much after $ct > 128$. Thus, it is good enough to choose $ct \in [100, 200)$ in practice to balance

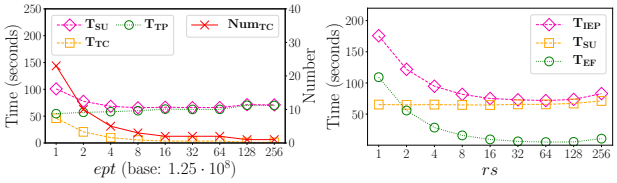


(a) On WE **(b)** On TW

Figure 10: Scalability to number of threads



(a) Varying wpt on WE **(b)** Varying ct on TW



(c) Varying ept on WE **(d)** Varying rs on WE

Figure 11: Effect of parameters

the overhead and benefit. In Fig. 11c, we observe that the underestimation of triangle-peeling (TP) throughput, e.g., $ept < 5 \cdot 10^8$, increases the number of triangle counting (TC) invocations, most of which should be replaced with TP. In contrast, the overestimation of TP throughput, e.g., $ept > 1.6 \cdot 10^{10}$ does not affect the choice of TC over TP. This is because there is a large gap in $T_{TC} \cdot (|E_G| - |Q_C|)/|E_{G_0}|$ and $l \cdot |Q_C|/ept$ for the most time-consuming iteration in the level $l = 9584$ on WE. In Fig. 11d, we observe that the range index size $rs = 32$ works best in our experiment on WE, and the increment of rs incurs more index maintenance overhead than the overhead with a small rs value. However, it is worthwhile to adopt the indexing and choose a relatively large rs, e.g., 16 and 32, because it significantly reduces the overall time of iterative edge-peeling. Much larger rs values may slow down the performance because the index contains more elements to scan and becomes less effective.

5.3 Evaluation of Offloading Techniques

Effect of the SI Stage Offloading. We compare OPT-TC on the CPUs and OFF-TC executed on the GPU. Results (the two bars on the left of Fig. 12) show that the triangle counting performs much better on the GPU and achieves a speedup of up to 8.4x over OPT-TC. This large speedup is because we exploit the warp-level parallelism to fully utilize the computation resources and the high bandwidth of GPU memory.

Effect of the IEP Stage Offloading. We compare the time of OFF-EIEP and OFF-RIEP (discussed in Sect. 5.1) against the CPU-only OPT-IEP (the middle three bars in Fig. 12) and give the time breakdown of OFF-RIEP on the CPU and GPU (the two bars on the right). Results show that OFF-EIEP is less competitive than OFF-RIEP on all the datasets and is even slower than OPT-IEP on WI. To analyze the reason, we profile OFF-EIEP and OFF-RIEP and show the memory size and transfer time of page swaps in Table 7. We find that the amount of page swap and data transfer in OFF-EIEP is orders of magnitude more

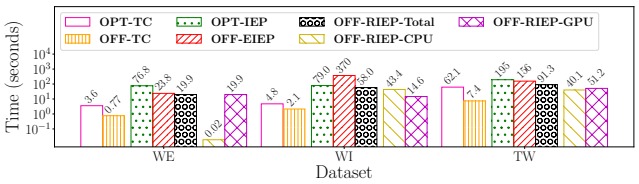


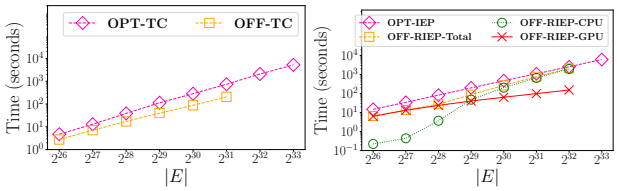
Figure 12: Effect of offloading techniques

Table 7: Effect of offloading strategies for IEP (seconds)

	OFF-EIEP		OFF-RIEP		Benefit of OFF-RIEP	
Dataset	M_{ps}	T_{ps}	M_{ps}	T_{ps}	$T_{ps}\downarrow$	$T_{IEP}\downarrow$
WE	15.0GB	3.6	1.5GB	0.8	2.8	2.6
WI	>2TB	457.7	12.4GB	1.4	456.3	423.0
TW	759.5GB	142.0	13.7GB	2.7	139.3	103.9

than that in OFF-RIEP, even though OFF-EIEP already aggressively compacts all the data structures periodically. Comparing OFF-RIEP with OPT-IEP, we find that the remaining computation on the GPU is 2.4x-3.8x faster than that on the CPU, which indicates it is effective to offload the edge-peeling computation to the GPU when the intermediate results are within the GPU memory capacity.

5.4 Evaluation of Scaling to Large Graphs



(a) SI Stage (TC) **(b)** IEP Stage

Figure 13: Effect of varying $|E|$

We use OPT-TC, OFF-TC, OPT-IEP, and OFF-RIEP to evaluate the effect of varying $|E|$ from 64 million to 8 billion (Fig. 13). In the SI stage, both OPT-TC and OFF-TC scales linearly to $|E|$, and OFF-TC is 1.7x-3.6x faster than OPT-TC. However, when $|E|$ is greater than 2^{31} (2 billion), OFF-TC suffers from intensive page swaps and runs out of time budget (5 hours). In the IEP stage, CPU-only OPT-IEP scales well to $|E|$, whereas the overall improvement of OFF-RIEP over OPT-IEP decreases. We further show the time breakdown of OFF-RIEP, i.e., OFF-RIEP-CPU and OFF-RIEP-GPU. For large datasets, e.g., $|E| > 2^{30}$, we need to first perform iterative computation on the CPU, to make the memory consumption of the remaining compacted graph within the GPU memory capacity. As a result, for those cases, the CPU part takes more time than the GPU part. In the case $|E| = 2^{33}$, OFF-RIEP runs out of memory due to the memory consumption of the mapping from the compacted graph to the original graph and intermediate k-class results Φ_k. Nevertheless, in the remaining IEP computation, OFF-RIEP is 2.3x-4.3x faster than OPT-IEP.

5.5 Comparison of Optimized Algorithms

We show the overall performance comparison of h-index based algorithms: H-IDX, H-IDX+, and peeling-based algorithms: MSP, PKT, our OPT-CPU (CPU-Only), and OPT-HPU (with GPU) in Fig. 14. Our experimental time budget is 5 hours, and execution beyond the limit is overtime. H-IDX is always the worst on all the datasets, and H-IDX+ improves over H-IDX by up to 16.0x because of our data skew handling and edge mapping techniques. However, H-IDX+ is still less competitive than MSP and PKT, especially

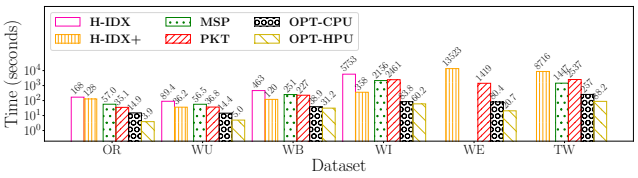


Figure 14: Overall performance comparison of optimized algorithms

on WE and TW, because its time complexity of $O(|\triangle|)$ multiplied by a max-h-index factor can be large.

MSP and PKT have comparable performance. MSP performs better in the presence of data skew (on WI and TW), since its array-based linked list enables quick skips, whereas PKT wins on the other datasets. However, MSP runs out of memory on WE due to $O(|\triangle|)$ memory complexity for the expansion and grouping of enumerated triangles. Our OPT-CPU and OPT-HPU implementations are 2.4x-25.7x and 7.3x-68.7x faster than the state of the art, respectively. In particular, OPT-HPU completes the computation on WE and TW within 21 and 88 seconds, respectively.

5.6 Summary

We start our evaluation on the CPU and use the state-of-the-art PKT as our baseline. Our parallelization of the PP stage and word-packing for the SI stage achieve speedups of 6.5x and 18.9x, respectively. The effect of individual techniques in the IEP stage varies with types of graphs. The DSTCP+IDX+ES, PSM+BMPF, and PSM+GC+VM (techniques ordered by the performance impacts) achieve improvement of 9.1x, 27.1x, 9.6x on the WE, WI and TW graphs, respectively. Our implementation scales well to the number of threads in all five components PP, SI, GC, SU, and EF. Our recommended setting of parameter values is as follows: $wpt = 4$, $ct = 128$, ept at a relatively large value, e.g., $1.6 \cdot 10^{10}$, $rs = 16$. Offloading the SI stage to the GPU improves the performance by 8.4x. The IEP stage always works the best to first compute on the CPU, then compact the intermediate results, and offload the remaining computation to the GPU. It runs up to 3.8x faster on the GPU than on the CPU. Finally, we find that the peeling-based algorithms are faster than the h-index based algorithms because their triangle enumeration and update cost is less. Our OPT-HPU is up to 68.7x faster than the state of the art and completes the computation on TW within 88 seconds.

6. CONCLUSION

To accelerate truss decomposition, we start from state-of-the-art peeling-based algorithms and design better preprocessing, support initialization, support update, triangle enumeration, and edge filtering procedures. In our design, we consider data skew in the real-world graphs and data access patterns in the algorithms. We parallelize and optimize our algorithms on both the multicore CPU and the GPU. Finally, we evaluate the effects of individual techniques and show that our implementations on both platforms outperform the state of the art by up to an order of magnitude.

7. ACKNOWLEDGMENTS

This work was supported in part by Grant MRA11EG01 from Microsoft Research Asia. Yue Wang is supported in part by Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515110473).

8. REFERENCES

[1] Cub documentation. http://nvlabs.github.io/cub/. Accessed in 2020.

[2] Gcc atomic built-ins. https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html. Accessed in 2020.

[3] Openmp documentation. https://www.openmp.org/. Accessed in 2020.

[4] E. Akbas and P. Zhao. Truss-based community search: a truss-equivalence based indexing approach. *PVLDB*, 10(11):1298–1309, 2017.

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

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 587–596, 2011.

[7] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, pages 595–602, 2004.

[8] L. Chang and L. Qin. Minimum degree-based core decomposition. In *Cohesive Subgraph Computation over Large Sparse Graphs*, pages 21–39. Springer, 2018.

[9] Y. Che. Rmat graph format converter code repository. https://github.com/RapidsAtHKUST/KroneckerBinEdgeListToCSR. Accessed in 2020.

[10] Y. Che. Rmat graph generator code repository. https://github.com/RapidsAtHKUST/Graph500KroneckerGraphGenerator. Accessed in 2020.

[11] Y. Che, Z. Lai, S. Sun, Q. Luo, and Y. Wang. Accelerating all-edge common neighbor counting on three processors. In *ICPP*, pages 1–10, 2019.

[12] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo. Source code of accelerating truss decomposition on heterogeneous processors. https://github.com/RapidsAtHKUST/AccTrussDecomposition. Accessed in 2020.

[13] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

[14] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report*, 16:3–1, 2008.

[15] 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*, pages 1–7. IEEE, 2017.

[16] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA*, pages 293–304, 2017.

[17] C. Gui, L. Zheng, P. Yao, X. Liao, and H. Jin. Fast triangle counting on GPU. In *HPEC*, pages 1–7. IEEE, 2019.

[18] Y. Hu, H. Liu, and H. H. Huang. High-performance triangle counting on gpus. In *HPEC*, pages 1–5. IEEE, 2018.

[19] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[20] H. Kabir and K. Madduri. Parallel k-truss decomposition on multicore systems. In *HPEC*, pages 1–7. IEEE, 2017.

[21] H. Kabir and K. Madduri. Shared-memory graph truss decomposition. In *HiPC*, pages 13–22. IEEE, 2017.

[22] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. Accessed in 2020.

[23] L. Lü, T. Zhou, Q.-M. Zhang, and H. E. Stanley. The h-index of a network node and its relation to degree and coreness. *Nature communications*, 7:10168, 2016.

[24] 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*, pages 1–7. IEEE, 2018.

[25] A. Montresor, F. De Pellegrini, and D. Miorandi. Distributed k-core decomposition. *TPDS*, 24(2):288–300, 2012.

[26] S. Pandey, X. S. Li, A. Buluç, J. Xu, and H. Liu. H-INDEX: hash-indexing for parallel triangle counting on gpus. In *HPEC*, pages 1–7. IEEE, 2019.

[27] R. Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *HPEC*, pages 1–4. IEEE, 2017.

[28] R. A. Rossi. Fast triangle core decomposition for mining large graphs. In *PAKDD*, pages 310–322. Springer, 2014.

[29] A. E. Sariyüce and A. Pinar. Fast hierarchy construction for dense subgraphs. *PVLDB*, 10(3):97–108, 2016.

[30] A. E. Sariyüce, C. Seshadhri, and A. Pinar. Local algorithms for hierarchical dense subgraph discovery. *PVLDB*, 12(1):43–56, 2018.

[31] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *WWW*, pages 927–937, 2015.

[32] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, pages 135–146, 2013.

[33] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *European Conference on Parallel Processing*, pages 682–694. Springer, 2007.

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

[35] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

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

[37] C. Voegele, Y.-S. Lu, S. Pai, and K. Pingali. Parallel

triangle counting and k-truss identification using graph-centric methods. In *HPEC*, pages 1–7. IEEE, 2017.

[38] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[39] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo. K-truss decomposition of large networks on a single consumer-grade machine. In *ASONAM*, pages 873–880. IEEE, 2018.

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

[41] Y. Zhang and J. X. Yu. Unboundedness and efficiency of truss maintenance in evolving graphs. In *SIGMOD*, pages 1024–1041, 2019.