Noname manuscript No.
(will be inserted by the editor)

Accelerating Triangle Enumeration on FPGA-CPU Heterogeneous Platforms

Yinnian Lin · Xunbin Su · Lei Zou

Received: date / Accepted: date

Abstract Triangle enumeration seeks out 3-cliques in a given data graph, which is vital for graph analysis. However, its performance often turns into a bottleneck due to set intersections' poor locality and imbalanced workload. Worse, the problem becomes even more complicated when the data graph changes. Luckily, recent research has made progress in boosting triangle enumeration with modern hardware for parallel computing, such as GPU and FPGA.

Therefore, we present TEAF, a triangle enumeration acceleration system optimized for CPU-FPGA heterogeneous platforms. First, we propose a structure-aware adaptive strategy for a crucial operation in triangle enumeration, set intersection. Instead of designing a one-size-fits-all algorithm for set intersections, we use a data-driven method to decide the intersection implementation for different settings. Second, a series of hardware-level optimization, including pipelining, customized cache, and CPU-FPGA co-processing, can reduce computation and memory costs and lessen workload imbalance. Third, we develop a parallel update technique for our graph storage structure to handle triangle enumeration on dynamic graphs. Experiments show that TEAF prevails over the previous systems with significant speedup on average. Also, TEAF outperforms its counterparts even more in terms of energy-averaged performance.

Yinnian Lin
Peking University, Beijing, China
E-mail: linyinnian@pku.edu.com

Xunbin Su
Peking University, Beijing, China
E-mail: suxunbin@pku.edu.com

Lei Zou (corresponding author)
Peking University, Beijing, China
E-mail: zoulei@pku.edu.com

Keywords Triangle enumeration · FPGA · Heterogeneous Computing

1 Introduction

Graphs are becoming increasingly popular in industry and academia due to their ability to model complex relationships between entities. Among various types of graph analysis tasks, triangle enumeration is the listing of all triangles in a graph, which is the basis for computing various graph metrics such as k-truss [49], clustering coefficients, and transitivity ratio [84]. Moreover, many real-world graph analysis applications such as spam detection [7], link recommendation [81], community detection [50], and so on also rely on triangle enumeration. Therefore, researchers have proposed various algorithms and optimizations [8,20,85,64,84,86,25,78,94] covering different settings: single or distributed, sequential or parallel, CPU-only or heterogeneous.

However, with the rapid growth in the size and update rate of graph data, the performance of triangle enumeration is often a bottleneck for graph analysis algorithms [73] in the traditional CPU-only environment. Recently, the advent of modern hardware (such as FPGAs and GPUs) for parallel computing brings hope for efficient triangle enumeration, especially on large and skewed real-world graphs [34,33,46,39,69,94]. FPGAs and GPUs provide massive parallelism, faster IO, and asynchronous co-processing alongside the CPU, while each has unique characteristics. With careful design to harness the power of such hardware, the heterogeneous solution can significantly speed up triangle enumeration.

This paper focuses on a Field Programmable Gate Array (FPGA), which is a potential alternative for ac-

celerating triangle enumeration due to the following advantages: (1) An FPGA allows user-defined hardware-level processing units (PU). The PUs can run in parallel and perform different logic, making the FPGA powerful in a Multiple Instructions Multiple Data (MIMD) manner. (2) An FPGA is equipped with tens of MBs of on-chip memory units for customizing an efficient and application-oriented cache or index. This on-chip memory is as fast as L2 cache on CPUs and shared memory on GPUs, while its size is similar to L2 cache and larger than shared memory. (3) The power consumption of an FPGA-based solution is often an order of magnitude lower than a GPU-based solution. (4) An FPGA can serve as a co-processor alongside a CPU. Thus, the CPU and FPGA can each focus on what they are good at and run asynchronously to achieve the best performance. In the proposed system, we will take advantage of these features of FPGAs to accelerate triangle enumeration.

1.1 Challenges

However, there are some challenges in implementing a CPU-FPGA co-processing system for triangle enumeration. These challenges arise from FPGA architecture, graph data distribution, and algorithm behavior. In addition to demonstrating these challenges, we also discuss how FPGA features are relevant to overcoming these challenges. By default, our discussion is based on the latest Xilinx architecture, but the principles are universal to FPGAs.

The first challenge is to design an appropriate parallel triangle enumeration algorithm. Our goal is to develop a triangle enumeration routine with high parallelism and efficient memory usage to exploit the power of modern hardware. There are three categories of triangle enumeration algorithms: the linear-algebra-based [5,82,88], subgraph-based [98,89,23], and intersection-based [74]. More details are discussed in section 2.3. The linear algebra-based method represents a graph as a matrix and decomposes the triangle enumeration into a series of matrix operations. Matrix computation is regular and parallel-friendly, but the matrix representation is very sparse, especially for highly skewed real-world graphs where a small fraction of vertices dominate the majority of edges. The sparsity is detrimental to the memory efficiency and performance of matrix computation. A possible alternative is to use a sparse matrix representation, but it reduces memory consumption at the cost of introducing irregular data layout and difficulty for updates. The subgraph-based solution takes triangle enumeration as a special case of subgraph matching. Since a triangle is a simple pattern, some filtering and pruning techniques for subgraph matching may need to be revised. Worse, additional space may be required for intermediate results and auxiliary indices.

Existing work [34,33,46,39,69] has concluded that an intersection-based solution has the best performance on heterogeneous platforms. Its basic idea is to iterate on each edge or vertex and find the common neighbors between conjoint vertices (i.e., intersecting neighbor lists). The intersections are independent of each other and can be performed in parallel. The bottleneck is in repeatedly reading the neighbor lists and computing the intersections. With an FPGA, we can launch targeted PUs for fast intersections and use sufficient on-chip memory to store the neighbor lists. Therefore, in this paper, we can focus on optimizing the intersection-based method for FPGAs.

The second challenge is to develop an efficient list intersection strategy for FPGAs, which accounts for most of the computational cost in triangle enumeration [74,36]. Most existing systems tend to design a *one-size-fits-all* intersection method with massive parallelism [39,69]. However, the efficiency of list intersection algorithms depends on the length of the lists involved, so finding a uniform intersection policy is not trivial, especially for graphs with highly skewed degree distributions. Instead, inspired by a recent work LOTUS [25], we can adopt a *divide-and-conquer* design, taking appropriate intersection methods for different workloads. Unlike previous work, which was designed for a CPU-only platform, our system considers the behavior of workloads on different hardware. In addition, we proposed a cost model to optimize workload partitioning instead of using empirical thresholds in LOTUS [25].

The final challenge is triangle enumeration on dynamic graphs. Many real-world graph data, such as social networks, change rapidly, but existing solutions assume that the graph is static [69,39,46,25]. Therefore, a triangle enumeration system that is suitable for both static and dynamic graphs is more desirable. To the best of our knowledge, we are the first to design such a system on FPGA.

1.2 Our contribution

Based on the aforementioned observations and challenges, we develop `TEAF`, a novel Triangle Enumeration Acceleration system optimized for the CPU-FPGA heterogeneous platform. `TEAF` consists of a set of processing units on an FPGA and a host system on a CPU. Our contributions are as follows:

- We propose a structure-aware triangle enumeration algorithm optimized for a CPU-FPGA heterogeneous platform. We follow the *divide-and-conquer* principle, adopting appropriate intersection strategies for certain types of workloads. Unlike existing studies that use an empirical threshold to classify workloads, we propose a cost model that considers both algorithmic behavior and hardware architecture, as well as a heuristic algorithm to find the optimal workload classification.
- To increase the performance of the system, we have come up with FPGA-oriented optimizations such as pipelining, custom caching, and CPU-FPGA co-processing.
- We also discuss how `TEAF` supports triangle enumeration for dynamic graphs. We introduce dynamic graph storage, which allows fast insertion and deletion. We also propose an improved technique for graph updates.
- We conduct evaluations on the effectiveness of `TEAF`. It outperforms previous systems based on FPGA and CPU on average, and beats GPU-based systems in terms of throughput per watt.

The rest of the paper is organized as follows: Section 2 introduces the preliminaries, and section 3 gives an overview of the system. The triangle enumeration algorithm is presented in section 4. Section 5 introduces the hardware implementation and its optimization, followed by the description of our system's dynamic graph support in Section 6. Experimental results are presented in section 7 and section 8 concludes the paper.

2 Background and related work

2.1 Problem definition

Definition 1 (Graph) A graph is denoted as $G = \{V, E\}$, where V is a set of vertices and $E \subseteq V \times V$ is a collection of edges. For convenience, we assume that every edge connects two distinct vertices. $N(v) = \{u|(v,u) \in E(G), v, u \in V(G)\}$ denotes the neighbor list of a given vertex v. The degree a vertex refers to $deg(v) = |N(v)|$.

Definition 2 (Triangle enumeration) A triangle T (u, v, w) (in graph G) is formed by three vertices u, v and w, where $(u, v), (v, w), (u, w)$ are three edges in G. Triangle enumeration is to list all triangles in a given data graph G.

Fig 1 gives the result of triangle enumeration in an example graph.

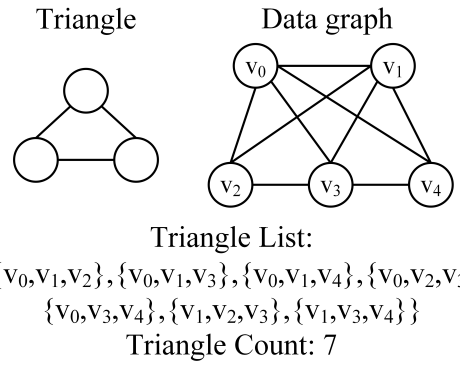


Fig. 1 An example of triangle enumeration. For triangle counting, the triangle number in the example data graph is 7. The system should output the complete list of triangles for triangle listing, as shown in the figure.

A triangle is a fundamental pattern in many graph analysis applications due to its informativeness and practicality in various scenarios. For instance, in a social network, a triangle represents a small community. In online trading data, a triangle is the smallest suspicious fraud cycle. It is worth noting that the graph data in these applications is rapidly changing. For example, Facebook's graph database receives tens of millions of user behavior records every day. Therefore, it is important to use a data structure and corresponding algorithms that support fast triangle enumeration and graph updates, especially for real-world applications. If the data graph is static, counting only needs to be done once.

Definition 3 (Dynamic graph) A dynamic graph is defined by an initial graph $G_i = (V, E)$ and a set of update operations $O = \{o_0, o_1, ...o_t...\}$ consisting of two kinds of operations: edge insertion and deletion, where t denotes the timestamp.

Our proposed framework uses a data structure that supports both fast triangle enumeration and graph updates. When a data graph changes over time, we call it a dynamic graph, as defined in Definition 3. To better leverage the power of parallel computing hardware, existing solutions typically adopt a batch update setting, where the system receives a batch of update operations, performs them in parallel, and outputs the number or list of affected triangles. To ease the burden of concurrent control, we assume in this paper that graph update and triangle enumeration do not overlap.[1] Enumerating the influenced triangles can be done by finding triangles containing a certain set of edges before and after

[1] There is related work focusing on real-time/streaming update processing, but such a strict requirement leads to a high cost of concurrent control, which is beyond the scope of this paper and will be discussed in our future work.

the graph updates [93]. This allows us to focus more on designing efficient graph storage and update techniques.

Table 1 Notations in this paper

Notation	Description
$G(V, E)$	A graph with vertex set V and edge set E
M_h	Classification threshold for heavy vertices
$deg(v)$	The degree of a vertex v
$deg_i(v)$	The in degree of a vertex v
$deg_o(v)$	The out-degree of a vertex v
$N(v)$	The neighbor list of vertex v
$h(N(v))$	The hash table storing $N(v)$
B_c	The size of a hash bucket
tlf	The expected load factor of $h(N(v))$

2.2 FPGA Features

2.2.1 Hardware Architecture

A field-programmable gate array (FPGA) is an integrated circuit that contains arrays of programmable logic devices implemented with look-up tables (LUTs). FPGAs also have on-chip memory (BRAM), which can be controlled by the user and provides fast I/O transport. User-defined functions are translated into a specific distribution of logic blocks and memory elements on the chip, using configurable wires to connect them. FPGAs have a flexible architecture that enables them to implement any function as a digital circuit. This eliminates the need for fetching and decoding instructions, which is essential for CPUs or GPUs. Typically, FPGAs act as co-processors alongside the CPU, providing high parallelism and flexible high-speed memory.

Fig 2 displays a typical architecture for CPU-FPGA heterogeneous platforms. The FPGA has its off-chip memory (DRAM) and communicates with the CPU through PCIE. [2] (1) minimize expensive data movement between the CPU and FPGA, and (2) allow the CPU to dominate the FPGA and start processing units on it. While FPGA-based processing units are running, the CPU host waits, but they can work asynchronously for better performance. Our system utilizes these principles to design our CPU-FPGA co-processing technique.

[2] It is important to note that this is not the only architecture available for CPU-FPGA heterogeneous platforms. Other architectures exist where the FPGA and CPU are packaged together and share main memory. However, the specific hardware we use, the Xilinx Alevo U200, relies on PCIE.

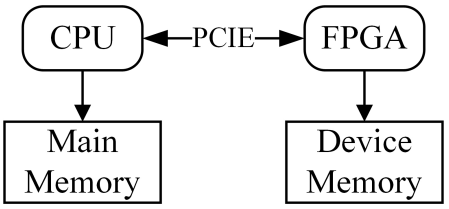


Fig. 2 Organization of a typical CPU-FPGA heterogeneous platform: an FPGA is equipped with off-chip device memory units, usually DDR or HBM. An FPGA can not directly access the main memory. The data movement between the main memory and the FPGA relies on the CPU, which often has limited bandwidth.

2.2.2 High-level programming

FPGA designs are typically written in hardware description languages (HDLs) such as Verilog and VHDL. However, programming with HDLs requires hardware design knowledge and can impede productive development. High-Level Synthesis (HLS) offers a potential solution. The HLS compiler allows designers to write functions in common high-level languages like C++ and Python, which are then translated to HDL. During this transformation, HLS provides several features to accelerate the proposed implementation:

- Loop pipelining and unrolling can be automatically organized as a hardware-level pipeline or unrolled into several parallel modules with certain compiler options.
- Burst read and write. HLS enables optimized sequential reads and writes, resulting in a 16x speedup over random memory access. This burst IO feature relies on the AXI protocol for off-chip memory access of the FPGA computing units.
- The HLS application can be complex and contain multiple modules. Some modules may have data dependencies on others. HLS offers primitives that enable users to instruct the compiler to connect modules with streaming I/O interfaces, reducing the cost of data transfer.

In our design, we make full use of these features to achieve decent performance. We will discuss the details in the following sections.

2.3 Triangle enumeration Methods

This section introduces the three common triangle enumeration algorithms: intersection-based, matrix-based, and subgraph matching based methods.

2.3.1 Intersection-based methods

Intersection-based methods [74,54,76,25,80,38,55] enumerate the vertices or edges of the data graph and calculate the intersection of the neighbor lists of every two connected vertices. Three intersection options are available: merging, binary search, and hashing.

Merging-based intersection requires the lists to be sorted. It maintains two pointers for the intersecting lists to go through them. During iteration, the program compares the referenced values and moves the pointers of the smaller value backward until one reaches the end of the list to identify common elements. A triangle is formed when the two pointers point to the same value, which represents the same vertex ID. Previous studies [76,29,41] have shown that the merging-based intersection is CPU-friendly due to its ability to be completed in linear time and its better spatial locality, resulting in a higher chance of hitting the cache. However, maintaining the order of neighbor lists during updates can be challenging because they must be sorted.

Binary search-based intersection builds a binary tree with one of the lists and considers the elements in the other list as search targets. For each search target, the classic binary search procedure is performed to see if there is a matching element in the search tree, i.e., a triangle. Unlike the merge-based intersection, the binary search is of higher parallelism because the search of different targets is independent. Some of the existing parallel solutions [44,45,1] choose binary search to implement list intersection. However, binary search is not as cache-friendly as merge, especially if the binary search tree is huge because it must randomly access memory and the data may not be in the cache.

Hash-based intersection constructs a hash table with one list and uses the elements in the other list as search keys to find the common elements. Similar to binary search, hashing search has high parallelism but worse locality. Also, its performance becomes unstable due to hashing collision (i.e., two elements are hashed at the same position). Several techniques have been proposed to handle hash collisions, such as linear probing, double hashing, and separate chaining, all of which introduce unavoidable overhead. Shun [76] implements hashing-based intersection for their multicore solution. They maintain many hash buckets in the hash table to avoid the cost of linear probing, which results in massive memory consumption. Bitmap is another common tool for list intersection, which can be thought of as a hash table with $|V|$ buckets. It sacrifices memory efficiency to eliminate collision costs. There are solutions on GPU and CPU that use bitmap [42,43,34,96,62] to achieve satisfactory performance, but they suffer from high memory consumption and therefore cannot handle larger graphs. Han et al. [36] and Qu et al. [71] propose a compressed format for the bitmap (they are similar in data structure, but use different intersection methods). To reduce memory consumption, it divides the bitmap into blocks of fixed length. Then only the blocks with non-zero bits are stored, along with their index. For intersection, it uses a merge-based strategy, comparing first the index and then the blocks. So the blocks are sorted by their indices. However, such a design is disadvantageous for updates. To insert or delete an element, it first uses a binary search to find the position of the element. If the element does not belong to any existing blocks, a new block should be inserted, causing data movement to relocate the blocks.

Fig.3 shows how the three intersection options mentioned above work on two neighbor lists from Fig.1, $N(0)$ and $N(1)$. In Fig.3(b), at the first iteration, the first element in $N(0)$ is 1 and smaller than that in $N(1)$. So the pointer to $N(0)$ increases. In the figure, this means that the red arrow points in the horizontal direction. In the second iteration, both pointers increase because they point to 2, a common element in $N(0)$ and $N(1)$, and a triangle is enumerated. This way we can find all triangles. In Fig. 3 (c), we search for every element of $N(1)$ in the binary search tree of $N(0)$. For example, to decide whether 4 is present, the search must go down the binary search tree to the rightmost leaf node. In Fig.3 (d), we construct a hash table with the longer list $N(0)$. When searching for 3, 3 is hashed to an occupied position, so we should use linear probing to check the next position to find 3.

2.3.2 Matrix-based approach

The matrix-based approach [5,88,17,13,14,91,65] stores a data graph in the form of a $|V| \times |V|$ adjacency matrix. Fig. 4 shows the adjacency matrix representation A for the data graph in Fig. 1. We first decompose the adjacency matrix into a lower and an upper triangular matrix, denoted L and U in Fig. 4. Then we compute the standard matrix multiplication to get $B = L \cdot U$. The elements in B count the number of wedges (i.e., 2-hop simple paths). Then we do an element-wise multiplication of A and B to check if the wedges can form triangles. To get the final result, we sum the element in the resulting matrix and divide it by 2. Matrix computation has a regular computation pattern and is therefore suitable for high parallelism. There are many efficient and easy-to-use libraries for matrix computation. However, as the size of the graph grows, memory efficiency becomes a problem. If the matrix is stored as a $|V| \times |V|$ 2D array, the memory consumption of the

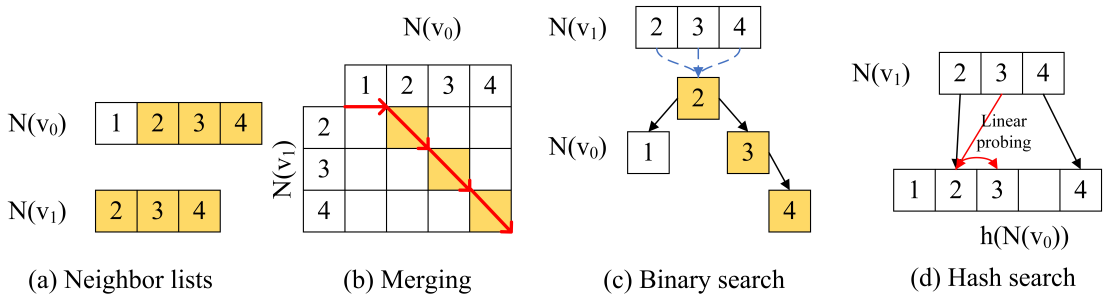


Fig. 3 Three intersection options for intersection-based triangle enumeration. Common elements are demonstrated in yellow. (a) shows the neighbor list of v_0 and v_1 in Fig.1. The red arrows (b) show the merge path when intersecting $N(0)$ and $N(1)$. (c) and (d) build the binary search tree and hash table with the longer list, which is a common choice in previous work.

adjacency matrix grows by the square. Worse, when a new vertex is inserted, the adjacency matrix may need to be reorganized, which is time-consuming.

A possible alternative for enumerating triangles in a matrix-based approach is to use sparse matrix representations such as CSR, COO, and CSC. [92,59,24] These data structures are space-efficient, but their performance is not as good as intersection-based methods, as recent research has shown [69,25]. Sparse matrix multiplication (SpMM) is more challenging to parallelize due to its unstructured data storage compared to dense matrix representation. Currently, there is no existing research on triangle enumeration using sparse matrix representation on FPGA. However, there is growing research interest in accelerating SpMM with FPGAs [66,4,60,70]. However, most of the research in this area focuses on improving SpMM for deep learning and has not been tested on large graphs with millions of vertices and billions of edges. It is important to note that updating a sparse matrix in a compact storage can be expensive.

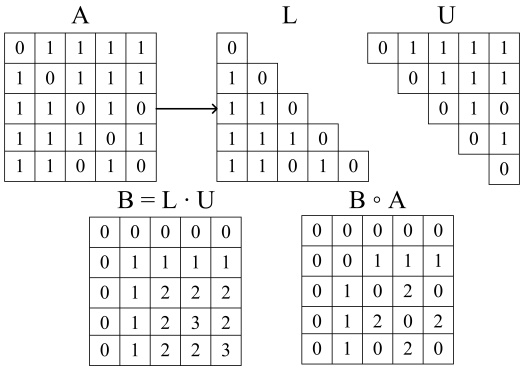


Fig. 4 An example for matrix-based triangle enumeration.The notation · denotes standard matrix multiplication and ∘ is element-wise matrix multiplication.

2.3.3 Subgraph matching based methods

Subgraph matching-based methods take a query graph as input and search for every subgraph that satisfies the query graph pattern. There are two types of mainstream subgraph matching algorithms. One method is based on exploration [37,72,12,82,23,52]. In this approach, the query graph (in this case, a triangle) is first decomposed into a spanning tree and non-tree edges. Subsequently, we identify all subgraphs that match the spanning tree pattern as partial matching results. Finally, we filter the results from the last step by checking the existence of non-tree edges. Other solutions are based on joining [63,11,35,79]. These solutions iterate through the vertices of the query graph in a matching order generated online. At each iteration, they attempt to match the current query vertex to extend the results from the previous iteration.

Triangle enumeration is a specific case of subgraph enumeration. Therefore, subgraph-matching solutions can be easily adapted to triangle enumeration by considering a triangle as the pattern to be matched. However, optimization techniques such as filtering and pruning may not be as effective in this case, as they are intended for a more universal problem.

2.4 Heterogeneous triangle enumeration

GPU is a popular hardware choice for parallel triangle enumeration. Green et al. [34,33] provide an algorithm for workload estimation and allocation to achieve better balancing. TriCore [46] and TC-steam [47] present a binary-search-based list intersection method for efficient and scalable triangle counting. Hu et al. [39] propose a fine-grained workload distribution strategy. Trust [69] proposed a hash-based intersection strategy on GPU for triangle counting.

However, many previous works on GPU have overlooked the dynamic graph scenario, despite it being a

recently popular topic [30]. To support dynamic graph analysis, Gunrock [23] introduces a GPU-friendly hash table technique. Instead, researchers have focused on dynamic graph storage on the GPU [53,32,16,87], enabling users to develop triangle enumeration algorithms.

Many researchers are studying graph computing systems on FPGAs or CPU-FPGA hybrid platforms. Some focus on building a universal graph processing framework [22,21,99,18,98,89]. For example, in ForeGraph [22], graph partitioning and communication in the multi-FPGA system are discussed. [21] presents a streamlined vertex-centric framework based on an interval-shard structure. Other works aim at solving specific graph problems such as BFS [95], shortest path [56], and maximum matching [100]. However, few of these works can be directly adapted to triangle enumeration, and most suffer from performance loss when graphs become large.

Researchers are studying graph computing systems on FPGAs or CPU-FPGA hybrid platforms. Some are focused on building a universal graph processing framework [22,21,99,18,98,89]. For instance, ForeGraph [22] discusses graph partitioning and communication in the multi-FPGA system. FPGP [21] presents a streamlined vertex-centric framework based on an interval-shard structure. Other works aim to solve specific graph problems, such as breadth-first search [95], single-source shortest path [56], and maximum matching [100]. However, few works can be directly adapted to triangle enumeration, and most experience performance loss when dealing with large graphs.

To our knowledge, only one existing work has utilized FPGA to solve the triangle enumeration problem [48]. The forward algorithm [74] was implemented on FPGA in this work. In this system, FPGA accelerators retrieve a pair of neighbor lists from DRAM to BRAM and perform a merge-based intersection. Parallelism is achieved by launching multiple processing units (PEs). Although this work is an early exploration of triangle enumeration on FPGA, it lacks consideration of graph metrics, particularly for large graphs. Furthermore, the authors did not discuss how their system supports dynamic graphs.

2.5 Triangle enumeration in dynamic graphs

Most research on triangle enumeration in dynamic graphs or graph streams [6,15,51,58,77,83] focuses on obtaining an approximate triangle count. The main focus is on designing edge or vertex sampling for more accurate and faster triangle number estimation. Recent topics in this field include edge duplication [77], sliding window [83], and estimation with bounded memory [31]. However, it is important to note that estimating the number of triangles may not be sufficient for all applications [73]. Our proposed system can provide the exact number or list of triangles, filling the research gap.

2.6 Data structure for dynamic graphs

Efficient data structures for dynamic graphs on various hardware have been extensively studied. Typically, researchers aim to adapt common static graph structures to accommodate frequent graph updates without sacrificing their advantages in accessing and scanning graph data. Some works follow the idea of an adjacency list, considering graph data as a collection of neighbor lists so updates on different lists are independent. Thus, researchers propose or adopt data structures that are efficient in updating neighbor lists, such as hash tables [23, 3], sparse arrays [87,16,27], skip lists [28], and key-value storage [19].

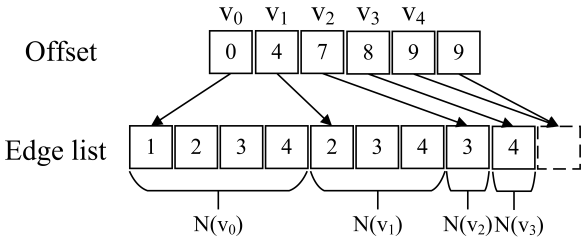


Fig. 5 CSR representation of the data graph in Fig.1.

There are also studies based the Compressed Sparse Row (CSR) structure, which is commonly used for storing large graphs [46,23]. Fig. 5 shows an example of the CSR structure, which consists of offset and edge lists. The neighbor lists are concatenated into a large edge list in the order of their vertex ID. An offset list indicates the beginning position of neighbor lists in the edge list by recording the prefix sum of the degree of vertices in ID order. To find the neighbor list of vertex v_0 in CSR, we should read $offset[0]$ and $offset[1]$ to determine the starting position and length of $N(v_0)$. Adding a new CSR element requires moving elements backward in its edge list, resulting in significant overhead. Previous studies have suggested reserving empty slots for future insertions and using block arrays [32,61, 101] or tree-like structures [57,75,102,68] to maintain balance.

Utilizing Packed Memory Array(PMA [9]) for dynamic graph storage is a successful practice in recent works[75,102,68]. PMA is designed for storing elements serially while reserving empty slots for future insertion

to facilitate fast updates. For an array of N entries, PMA divides the whole array into fixed-length leaf segments. Then, PMA uses a self-balancing tree to make sure that reserved slots are evenly distributed in leaf segments. Non-leaf segments are defined as the integration of their descendant segments. After an insertion or deletion, if the density of a leaf segment violates a given range, PMA attempts to reallocate its elements to its sibling(i.e., leaf segment with the same parent) segment. If its sibling segment does not have enough space, PMA traces back to higher layers to include more segments.

3 System Overview

Fig. 6 presents an overview of our proposed `TEAF` (Triangle Enumeration Acceleration system optimized for CPU-FPGA heterogeneous platform). TEAF is designed to count or list triangles in a data graph and comprises a host system and a set of processing units (PUs) on an FPGA. The host system, located on the CPU, is responsible for offline data preprocessing and constructing the graph data structure. It also launches the processing units and allocates the workload. The computation ordered by the host is conducted by FPGA-based processing units (PUs), which then write the results back to the CPU via a PCIe interface. To enumerate the triangles in a given data graph, TEAF follows a four-step process: preprocess, data structure construction, workload allocation, and computation on the FPGA side.

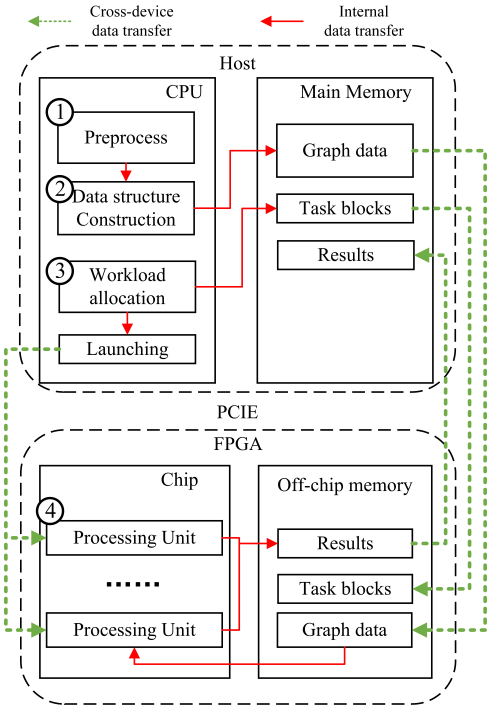


Fig. 6 The `TEAF` Architecture

3.1 Preprocess

The first step is to load the graph into main memory for preprocessing. This involves vertex reordering, vertex classification, and edge filtering. Each vertex is typically assigned a unique integer ID, and the distribution of these IDs can impact the performance of triangle enumeration [39]. Therefore, optimization may be necessary, such as vertex reordering. In this case, we sort the vertices by their degrees and use their ranks as IDs. For vertices with the same degree, they are ranked by their original IDs. This allows for the comparison of two vertices' degrees by their IDs and iteration of the vertices in descending order of their degrees, which facilitates later computation. Following the reordering, all vertices are partitioned into two groups based on their degrees: heavy vertices and light vertices. Targeted data structures and intersection methods are employed for different types of vertices. As the vertices have already been sorted by their degrees, a parameter M_h can be used to distinguish between the heavy and light vertices. M_h is equal to the largest ID of the heavy vertices. The selection of M_h affects the total workload, so a greedy algorithm has been developed to search for a good enough M_h in linear time. The details are discussed in Section 4. Following this, the neighbor lists are filtered by the host, deleting neighbors with a smaller ID. For the convenience of discussion, for an edge (u, v), if $deg(u) > deg(v)$, we say u is the source vertex of (u, v) and v the destination vertex (i.e., (u, v) is an out-edge of u and an in-edge of v. To avoid duplicate results in triangle enumeration, it is effective to use this trick [74] because a triangle can only be induced once by one of its edges that connects the two vertices with a higher degree.

3.2 Data structure construction

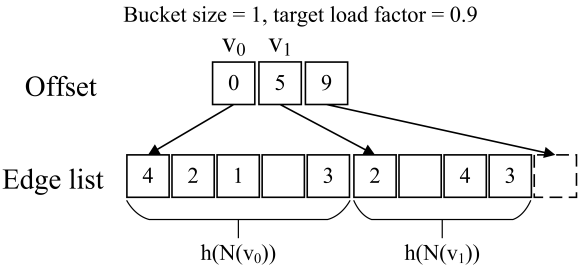
To adhere to the principle of *divide-and-conquer*, we utilize distinct data layouts for heavy and light vertices. These layouts are based on Compressed Sparse Row [2](CSR). This structure allows for fetching a neighbor list with only two random memory accesses and two series of sequential ones. Such an access pattern is beneficial for triangle enumeration, which requires repeated reading of neighbor lists.

For light vertices, we directly use CSR for static graphs and apply an update-friendly array called Packed

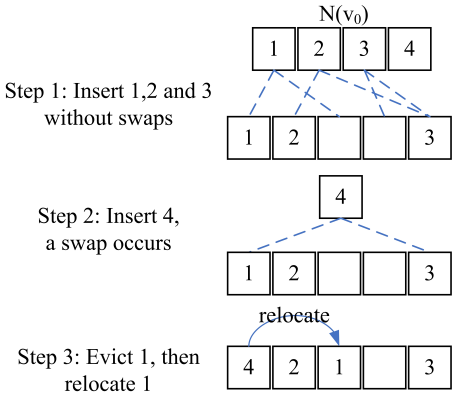
Memory Array [9] for dynamic graphs. For heavy vertices, as shown in Fig. 7(a), we build a cuckoo hash table [67] for every heavy neighbor list and then organize the neighbor hash tables as CSR. The idea is straightforward since we can simply replace the neighbor lists with hash tables and recompute the offsets. A cuckoo hash table is efficient for point lookup(i.e., searching for a single element). It uses k hash functions to generate k candidate slots(or buckets) for a to-be-inserted element v. Then, it randomly selects one available slot to insert v. If all candidate slots are full, a swap operation is triggered. This means that one of the elements in these candidate buckets is moved to its other vacant candidate slots to make space for v. If there are still no available slots, the swap operation continues. This guarantees that any element will remain in one of its candidate buckets, ensuring constant search time (i.e., checking k candidate buckets) and enabling fast hash-based intersection.

Fig. 7(b) gives an example of building a cuckoo hash table from $N(v_0) = \{1, 2, 3, 4\}$, where each element is mapped to two candidate slots. Inserting the first three elements needs no swap operation. However, to insert 4 whose candidate slots are all occupied, a randomly chosen element 1 has to be evicted and moved to another candidate slot.

In summary, cuckoo hash tables prioritize faster point lookup over space consumption and update speed. On one hand, to manage the memory consumption of hash tables, we introduce an empirical parameter called the target load factor (tlf). When constructing the hash table $h(N(v_h))$ for a heavy vertex (v_h), we allocate $\lceil \frac{|N(v_h)|}{tlf \times B_c} \rceil$ hash buckets for $h(N(v_h))$, where $tlf < 1$. On the other hand, one major issue with a cuckoo hash table is that updates can be slow and may even fail, particularly when the load factor increases, due to an excess of swap operations. In Section 6, we will discuss an improved solution for parallel updates for cuckoo hash tables.



(a) The data structure for the heavy neighbor lists in Fig.1.



(b) Building $h(N(v_0))$

Fig. 7 Data structure in `TEAF`. The light blue dash lines indicate the candidate slots of an element.

3.3 Workload allocation

With vertices classified into heavy and light, edges are divided into three groups: edges connecting two light vertices, two heavy vertices, and one light vertex with a heavy vertex. The host scans all edges and prunes those with conjoint vertices without valid neighbors. The remaining edges are taken as tasks and are stored as a set of fixed-length task blocks. We have a customized cache, including a typical LRU cache and a static buffer. Therefore, when filling the task blocks, we collect the edges with common conjoint vertices, particularly high-degree ones, and group them into the same block to increase the cache hit rate.

In our design, it is easy to identify edges that have common conjoint vertices. For a graph $G = (V, E)$, the edges that connect a vertex v and its neighbors can be expressed as $\{(v, u_i) | u_i \in V, (v, u_i) \in E\}$. Referring to the data graph in Fig. 1, we can see that there are four edges $\{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_0, v_4)\}$ connecting to v_0, which have a common conjoint vertex v_0. It is important to note that the neighbor lists of the vertices are stored, and for a vertex v, its neighbors are connected to it by an edge. To clarify, the edges with v as a common vertex can be easily identified by enumerating $N(v)$.

3.4 FPGA processing

Once the graph data structure and task blocks are prepared and sent to the FPGA off-chip memory, the host can initiate the FPGA processing units to conduct parallel triangle enumeration. It is important to note that the host does not need to be blocked and wait for the FPGA processing to finish. Instead, the host can share a small workload for better performance. The implementation of the processing units is introduced in Section

5. The final results are written to off-chip memory and then to the main memory.

The framework above addresses two critical issues related to performance tuning. The task assignment among the processing units on FPGAs is determined by the partition strategy of the heavy and light vertices. Therefore, we propose an analytic model to select the best partition to minimize workload imbalance in Section 4.2. Secondly, the PU design should achieve both massive parallelism and an efficient memory access pattern. Thus, we propose a pipeline strategy, a caching mechanism, and a CPU-FPGA co-processing design to enhance our performance as described in Section 5.3.

4 Structure-aware triangle enumeration

This section discusses the algorithm and parameter tuning of our proposed system. Our triangle enumeration method is structure-aware because we choose the most appropriate intersection strategies based on the data graph's structure. Additionally, since the proposed system is intended for a CPU-FPGA heterogeneous platform, we consider the features of different hardware and an algorithm's behavior on CPUs and FPGAs to optimize our design. `TEAF` enumerates triangles by iterating on edges and calculating the intersections between the neighbor lists of an edge's two conjoint vertices. This method is space-saving and has good computational complexity. However, performing triangle enumeration on large and power-law-distributed graphs remains challenging due to the expensive computational cost of conducting the intersection. Our algorithmic design focuses on addressing this issue.

4.1 Algorithm

In contrast to previous heterogeneous solutions that implement a *one-size-fits-all* intersection strategy [46,69, 36], `TEAF` applies different intersection methods based on the sizes of the two intersecting neighbor lists. In our problem, there are three kinds of intersection: one between two light neighbor lists, one between two heavy neighbor lists, and one between light and heavy neighbor lists. We use LLI (light-light intersection), HHI (heavy-heavy intersection), and HLI (heavy-light intersection) for short in the later discussion.

For LLI, intersecting neighbor lists are both short and a typical merge-based intersection turns out to be suitable. Other methods either have higher complexity (e.g. binary search) or heavier overhead of building auxiliary structure (e.g. hash-based search). Specifically, we maintain two pointers to the neighbor lists respectively and compare the elements they refer to. If they are equal, a triangle is found and both pointers are advanced together. Otherwise, we move forward the pointer to the smaller elements by one step. Merging requires sorted neighbor lists. However, since most of the light neighbor lists are small (usually only hundreds or tens of elements), sorting them is not very costly and can be done offline. `TEAF` conducts multiple merging in parallel to accomplish high parallelism. However, we do not implement intra-intersection parallelism because it takes logarithmic time to split the lists, which is not worth especially for short lists. Based on the better locality in merge-based intersection [46], when the pointers move forward, it is likely that the next elements are already present in the cache due to the continuous storage of intersecting lists.

For HLI, a search-based solution appears promising because we can search for every element in one intersecting list in the other without data dependency. This reduces the burden of concurrent control. If the parallelism is sufficiently high, the impact of complexity and overhead of search-based solutions can be reduced. In addition, random memory access is a bottleneck for search-based methods on the CPU, resulting in a lower cache hit rate. However, FPGAs enable users to manage on-chip memory, which operates at a speed comparable to that of L2/L3 Cache on CPUs, with user-defined logic. Therefore, the entire search indices (i.e., search trees or hash tables) or auxiliary structures can be stored in on-chip memory to minimize memory access to slow off-chip memory units.

Two types of search algorithms are available: binary search and hash search. Hash search requires additional time and space to build the hash table and may encounter hashing conflicts. Fortunately, the cuckoo hash table is available for use.

Algorithm 1: N-way Cuckoo-hashing-based Intersection for HLI

Input: light neighbor list $N_L(v)$ and hashed heavy neighbor list $N_H(u)$, hash function h_1,h_2...h_n
Output: number of common element C

$C \leftarrow 0$;
foreach $v' \in N_L(v)$ ***in parallel*** **do**
  calculate $h_1(v'), h_2(v')...h_n(v')$;
  foreach $b \in \bigcup_{i=1...n} N_H(u)[h_i(v')]$ ***in parallel*** **do**
    if $v' \in b$ **then** $C \leftarrow C + 1$;
  end
end

A k-way cuckoo hash table typically maintains a group of hash slots, or buckets, and k corresponding hash functions. The hash functions map an element to k slots, and the element is stored in one of them. If there are no available slots when inserting a new element, one of the elements already in these slots is chosen and relocated to another one among its other $n-1$ hash slots. This ensures that a search will scan k buckets at most and end in constant time.

`TEAF` builds M_h cuckoo hash tables for each heavy neighbor list and compacts them as CSR in the preprocessing phase. The PUs then perform a parallel search in the cuckoo hash table to obtain the HLI results. Algorithm 1 illustrates this process using a k-way cuckoo hash table. Note that the search for different vertices and the scanning of the k potential buckets are performed in parallel.

Similarly, in terms of HHI, the merge-based solution is less friendly towards fine-grained parallelism, while the hash-based search can utilize the pre-built hash table to achieve high parallelism. Since the time cost of parallel search in the cuckoo hash table depends on the bucket size rather than the list size, `TEAF` selects the elements in the smaller hash table as search targets. The challenge lies in iterating through a hash table instead of a sorted list. Empty slots in a hash table can lead to performance loss. However, a cuckoo hash table can run at a load factor of around 0.9 [26] by allocating an appropriate number of hash slots or buckets. Therefore, the overhead of empty positions is acceptable in our system.

The above algorithm employs suitable strategies for various list intersection scenarios. However, it is important to consider the hardware-related design and optimization of our system, which will be discussed in the following sections.

4.2 Optimizing M_h

To enhance the performance of our proposed system, it is crucial to determine an optimal boundary M_h that distinguishes heavy and light vertices. This boundary directly affects the allocation of the three workloads: LLI, HLI, and HHI. Workload allocation is significant since `TEAF` concurrently deals with three workloads, and the slowest workload dominates its final running time. To quantify M_h's influence, we need to design a model that estimates `TEAF`'s performance with a given M_h. However, we face two challenges with this requirement:

First, we have developed specific strategies for three different workloads that accommodate the storage structures for the neighbor lists of heavy and light vertices. Cuckoo hash tables are utilized for the neighbor lists of heavy vertices, while sorted arrays are maintained for light vertices. As a result, the workloads exhibit various behaviors. For LLI, two sorted neighbor lists are scanned, the elements are compared, and the process proceeds according to the comparison result. HLI searches for elements in a sorted neighbor list within cuckoo hash tables. For HHI, we enumerate a hash table to find valid search targets and check their existence in the other hash table. Therefore, workloads should be estimated based on their respective behaviors.

Second, since our system is running on a CPU-FPGA heterogeneous platform, it is important to share the three workloads between the CPU and the FPGA to avoid wastage of computing power. However, workload estimation can be challenging due to the completely different architectures of FPGAs and CPUs. CPUs have sophisticated pipelines designed to execute instructions from a specific Instruction Set. Every program is translated into a series of instructions. However, an FPGA kernel is implemented using lookup tables that can simulate any user-defined logic on a hardware level without the overhead of fetching and decoding instructions. A CPU program typically runs at a higher frequency than a typical FPGA kernel, but an FPGA kernel can achieve higher parallelism. We can not overlook the difference in hardware when estimating and allocating the workloads.

To address the challenges, we begin by breaking down the running time into two components: the number of operations and the latency of each operation. We estimate these factors separately and then approximate the running time by multiplying them. Regarding the first challenge, we observe that although the algorithms for processing LLI, HLI, and HHI differ in behavior, they perform the same type of operation: pairwise comparison. The distinction lies in the selection of elements to compare and the order in which the neighbor lists are accessed. Therefore, we can approximate the total number of comparisons for the three workloads based on their behavior. We represent the estimated comparison counts for LLI, HLI, and HHI as W_{LL}, W_{HL}, and W_{HH}.

For the second challenge, we first determine the appropriate hardware to assign different workloads. Our observation suggests that merging two sorted arrays is better suited for the cache mechanism of a CPU because it scans the arrays serially. In contrast, hash-based search introduces random memory access. Besides, hash-based search can achieve higher parallelism thanks to the independence among multiple searches, but parallel merging requires an extra process to ensure correctness. In conclusion, a CPU is suitable for

LLI, while an FPGA is better equipped to handle HLI and HHI. We estimated the latency of pairwise comparison on an FPGA and a CPU by conducting a pairwise comparison between two randomly chosen elements one billion times and recording the average latency. The estimation is denoted as L_C and L_F. Additionally, we considered parallelism and found that the growth pace of performance is not linear but instead gets slower as the number of concurrent threads or PUs increases. For instance, increasing the number of threads from one to four may result in a speedup of 2.8x. However, if the number of threads is further increased to eight, the speedup may decrease to 4.8x instead of 5.6x. To avoid the time-consuming task of testing the speedup trend for every data graph, we use sampling to reduce the data scale.[3] The speedup of systems with different numbers of threads or processing units (PUs) is tested against the serial implementation on the sampled graph. The resulting data points are then fitted using a polynomial curve. The polynomial function obtained is denoted as $f_c(n_c)$ for a CPU and $f_f(n_f)$ for an FPGA, where n_c represents the number of threads and n_f represents the number of PUs.

As previously stated, our objective is to minimize the running time, which is equivalent to minimizing the time of the slowest workload. Now that we have estimated the number of comparisons for W_{LL}, W_{HL}, and W_{HH}, as well as the comparison latency of CPUs and FPGAs, L_C and L_F, and the parallel speedup, $f_c(n_c)$ and $f_f(n_f)$, we can express our objective as follows:

$$\arg\min_{M_h} max(\frac{L_C}{f_c(n_c)}W_{LL}, \frac{L_F}{f_f(n_f)}W_{HL}, \frac{L_F}{f_f(n_f)}W_{HH}) \quad (1)$$

The calculation of W_{LL}, W_{HL}, and W_{HH} is dependent on the intersection strategies employed. For LLI, a merge-based method is utilized, which involves maintaining two pointers to the neighbor lists. At each iteration, the pointed elements are compared, and at least one of the pointers is increased. This procedure terminates when one of the pointers reaches the end of the list. Therefore, the average of the sizes of the two intersecting lists will provide a suitable approximation. The system applies cuckoo-hash-based intersections for HLI and HHI. It searches for each element from one list within the other, stored as a cuckoo hash table that guarantees $O(1)$ time for hash search. Therefore, workload can be measured as:

$$W_{LL} = \sum_{(u,v)\in E_{LL}} \frac{N(u)+N(v)}{2} \quad (2)$$

[3] We use a sampling algorithm in [30] that ensures the sampled subgraph has a similar distribution with the original graph

$$W_{HL} = \sum_{(u,v)\in E_{HL}} |N(v)|B_c \quad (3)$$

$$W_{HH} = \sum_{(u,v)\in E_{HL}} \frac{|N(u)|}{tlf}B_c \quad (4)$$

where E_{LL}, E_{HL} and E_{HH} represent three types of edges, while tlf denotes the load factor of the cuckoo-hashed neighbor lists, and B_c is the size of hash buckets. It is assumed, without loss of generality, that $|N(u)| > |N(v)|$. The workloads of merge-based and cuckoo-hash-based intersections are measured by the number of pairwise comparisons. It is important to note that we consider the load factor in W_{HH} because empty slots will idle the hash search modules in our design. A fixed tlf of 0.9 is used, which performs best in our experiments.

To achieve a global optimal using this cost model, one could enumerate every possible M_h. However, this brute solution has a time complexity of $O(|V||E|)$ due to the $O(|E|)$ time cost of calculating the workload, making it too slow. To address this issue, we have developed a heuristic approximation algorithm, as demonstrated in Algorithm 2. At each step, the algorithm attempts to designate the light vertex with the highest current degree as a heavy vertex (Line 4). It then computes the alterations in the workload estimation (Line 5) and terminates if no further reduction in W is achieved (Lines 3 and 6).

Algorithm 2: Optimizing M_h

Input: Sorted vertex list $V = \{v_1, v_2 ... v_{|V|}\}$
Output: The largest heavy vertex ID M_h

```
1 M_h ← 0;
2 calculate exact W_LL, W_HL, W_HH;
3 while true do
4 |   M_h ← M_h + 1;
5 |   Update W_HH, W_HL, W_LL with Algorithm 3;
6 |   if Δ_W < 0 then break ;
7 end
```

The key to Algorithm 2 is to estimate the changes in W_{LL}, W_{HL}, and W_{HH}. Algorithm 3 describes our solution in detail. Note that we only keep vertices with larger IDs in a neighbor list. When a light vertex v turns into a heavy one, all light-light edges connected to v are transferred to heavy-light edges(Lines 3-4). Meanwhile, $deg_i(v)$ heavy-light edges become heavy-heavy ones(Line 4-5), where $deg_i(v)$ indicates the number of heavy vertices that share an edge with v because we suppose the direction of an edge to go from high-degree vertices to low-degree vertices. $deg_i(v)$ can be recorded in the edge filtering phase. The only unknown values here are the sizes of neighbor lists of light vertices connected with v. Since the sizes of light neighbor lists are

all small, we considered them uniformly distributed. We can approximate their sizes with the median of light neighbor lists' sizes(Line 2).

Algorithm 3: Estimation of workload changes

Input: Sorted vertex list $V = \{v_1, v_2...v_{|V|}\}$, workload estimation W_{HL}, W_{HL}, W_{HL}, boundary vertex v_{M_h}

Output: the changes in workload estimation Δ_W, updated W_{HL}, W_{HL}, W_{HL}

1 $W_{old} \leftarrow max(\alpha_1 W_{HL}, \alpha_2 W_{HL}, \alpha_3 W_{HL})$;
2 $mid = \lceil (M_h + |V|)/2 \rceil$;
3 $W_{LL} \leftarrow W_{LL} - \frac{|N(v_{M_h})|}{2}(|N(v_{M_h})| + |N(v_{mid})|)$;
4 $W_{HL} \leftarrow W_{HL} + B_c|N(v_{M_h})|(|N(v_{mid})| - deg_i(v_{M_h}))$;
5 $W_{HH} \leftarrow W_{HH} + \frac{B_c}{l_T}|N(v_{M_h})|deg_i(v_{M_h})$;
6 $W_{new} \leftarrow max(\alpha_1 W_{HL}, \alpha_2 W_{HL}, \alpha_3 W_{HL})$;
7 $\Delta_W \leftarrow W_{old} - W_{new}$;

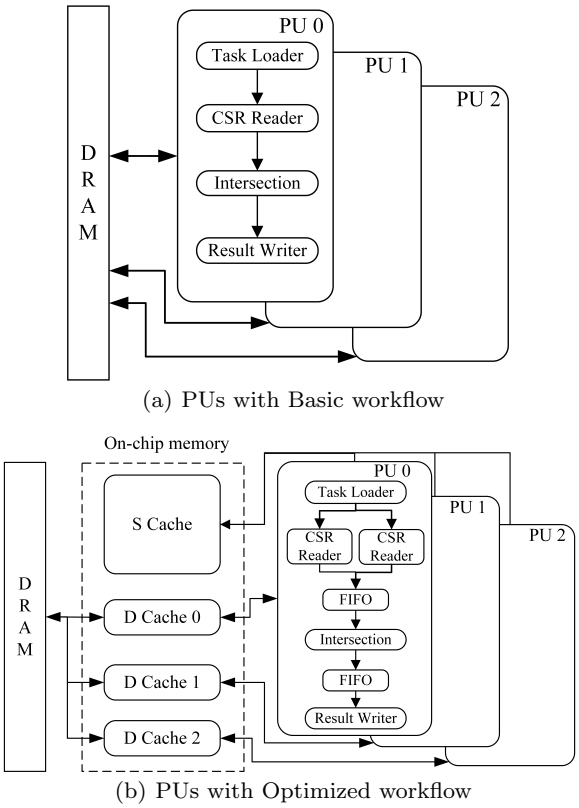
5 Processing unit design

In this section, we introduce the processing unit(PU) design for the three types of intersection: the light-light intersection(LLI), heavy-light intersection(HLI), and heavy-heavy intersection(HHI). We also analyze the bottleneck and propose optimization techniques, including pipelining, caching, and co-processing of the CPU and FPGA.

5.1 Basic workflow

All PUs share a similar basic workflow as shown in Fig. 8(a), which consists of four modules: Task Loader, CSR Reader, Intersection Module, and Result Writer. These modules operate in a serial fashion. First, the Task Loader fetches a task block from DRAM to BRAM with a burst read. Then, the CSR Reader moves the required neighbor lists from CSR in off-chip memory to on-chip buffers. Next, the Intersection Module is started. Each intersection module contains a set of working submodules that perform the actual intersection computation. For LLI, one submodule merges two neighbor lists. For HLI and HHI, a submodule performs a hash-based intersection on a cuckoo-hashed neighbor list. Finally, the result writer collects the output from the intersection module and sends it back to the host.

There are two major problems with the basic workflow design. First, parallelism relies on running multiple PUs, but the modules within a PU run serially. Other modules remain idle while one is working, wasting a lot of computing resources. Second, memory requests



(a) PUs with Basic workflow

(b) PUs with Optimized workflow

Fig. 8 The organization of Processing Units

can face severe contention when multiple CSR readers access CSR in off-chip memory. Therefore, we develop two targeted optimizations: workflow pipelining and caching.

5.2 Pipelining

In the basic workflow, the modules process a task block in a row, without pushing intermediate results to the next stage as soon as they're generated, so the following modules must wait. Although we can improve throughput by increasing the number of parallel modules, logic resources, and memory ports are limited. However, thanks to the dataflow optimization technique mentioned in Section 2.2, we can build a pipeline with FIFO IO streams to concatenate the modules.

As shown in Fig. 8(b), the Task Loader continuously fetches edges from the off-chip memory and pushes them into the FIFO stream to the CSR readers in the optimized PU. When the CSR reader finishes reading the required neighbor lists into the on-chip buffer, it sends their location to the Intersection Module via a FIFO stream. Upon receiving the data, the intersection immediately begins and the results flow to the Result Writer for later transfer. This greatly reduces the idle

time of each module. A module now continues its job when the result of the previous task is transferred to the FIFO stream. In addition, our design uses the on-chip BRAM to build shared buffers between the CSR reader and the intersection module, thus avoiding unnecessary data transfer. Note that the pipelined PU consists of hardware-level logic modules without the overhead of fetching and decoding instructions that is unavoidable on CPU and GPU.

According to our experiments, the key to improving the pipeline is to speed up the slowest module, which often turns out to be the CSR reader. There are two main reasons for its inefficiency. First, most of the off-chip memory transactions occur in this module to fetch the required neighbor lists. Although they are stored sequentially, and the FPGA's burst read feature can speed up such continuous read requests, they are still much slower than the on-chip memory operation because of the different storage media. Second, there are many CSR readers initiating read requests, but the IO ports of the off-chip memory (DRAM) are limited, so the concurrent IO bandwidth is also limited. A common solution to this problem is to reuse data as much as possible with caching techniques such as LRU (Least Recently Used Cache) and LFU (Least Frequently Used Cache). However, the user-controllable on-chip memory units of FPGAs prompt us to design a graph-structure-aware caching mechanism.

5.3 Graph-structure-aware Caching

Caching is an important way to reduce the memory cost of the system. It stores frequently used data in faster memory units. Fortunately, FPGAs provide flexible on-chip memory that allows us to design customized caches for a specific application. Our solution is inspired by an idea from [90] that different memory management benefits graph mining performance. Similar to graph mining, triangle enumeration also requires repeated reading of neighbor lists. Unlike traditional caching techniques, we consider the metric of the data graph to exploit the data locality in triangle enumeration.

We call ours a graph-structure-aware caching strategy. First, the static cache (S cache) maintains a fixed set of neighbor lists of frequently accessed vertices, so it is read-only during execution; the dynamic cache (D cache) is a typical LRU that is updated as we count triangles. The CSR readers first access the S cache and the D cache concurrently, and our design ensures that their contents do not overlap. Only if both caches miss, off-chip memory is accessed and the D cache is updated.

Fig.8(b) illustrates the memory hierarchies with an example of 3 PUs. The S cache is shared to reduce concurrency control overhead because it is read-only during execution, and the D cache is private for each PU to reduce concurrency control overhead. Both S Cache and D Cache are resident in on-chip memory, so the available on-chip memory limits their size. On a Xilinx Alevo U200 card used in the experiments, we allocate 4MB for the S cache and 256KB for each D cache. If the graph is larger, with more than 100M edges, we expand the S cache to 16MB and the D cache to 512KB.

The crucial point here is to decide which vertices are frequent. To estimate the memory access cost $M(v)$ of vertex v, we focus on the amount of data transfer it could bring and propose the following equation:

$$M(v) = (|N(v)| + deg_i(v)) \times |N(v)| \tag{5}$$

The value of $M(v)$ is dominated by $|N(v)|$, which is stored in the CSR structure. In practice, we iterate over the vertices in ID order, which is sorted by degree in descending order, and add them to the S cache until it's full. In this way, we ensure that the vertices in the cache have consecutive IDs, so that the CSR reader can check the existence of a vertex by comparing vertex IDs.

5.4 CPU-FPGA Co-processing

Another important way to improve performance is to use more computing resources. In the original `TEAF` design, after the processing units are started, the host on the CPU waits for the PUs to finish. The CPU side can share some tasks instead of sitting idle.

The key is to decide on the workload-sharing strategy. In `TEAF`, we adopt merge-based and cuckoo-hash-based intersections. Comparing these two intersection strategies, merging provides CPU-friendly sequential memory access, while cuckoo hashing introduces random access. On FPGAs, we can store the neighbor list on BRAM with higher random access bandwidth, but not on the CPU. Moreover, the cuckoo hash-based intersection can achieve higher parallelism within a single task. Therefore, we can conclude that merging is better suited to the CPU while hashing is more friendly to FPGA-based acceleration, so we assign the LLI workload to the CPU host. Also, we can still trust our algorithm to get M_h by modifying the α parameter.

Fig. 9 shows the benefits of CPU-FPGA co-processing. The runtime of preprocesses decreases because the system now sends fewer data and task blocks to the FPGA. Then, the LLI computation occupies the idle time before the co-processing is applied. With more logic and memory resources saved, `TEAF` can start more PUs for HLI and HHI. As for LLI, we can directly adopt existing CPU-based algorithms for fast list intersection.

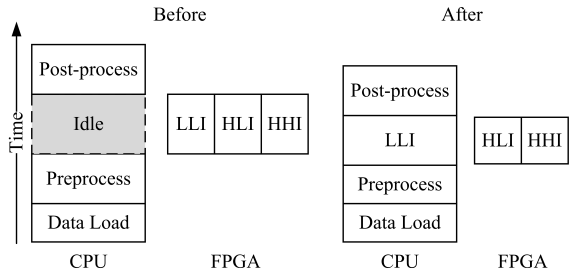


Fig. 9 Timeline of `TEAF` before and after CPU-FPGA co-processing

Therefore, our design can reduce the running time on both CPU and FPGA sides.

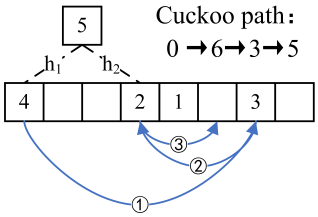
6 Dynamic graph support

There's a research gap on exact triangle enumeration for dynamic graphs on heterogeneous platforms. Current State-of-the-Art solutions focus more on triangle enumeration for static graph [69,39,46]. However, if the graph is static, spending so much effort to boost its performance seems less meaningful and interesting. After all, we only need to calculate once. Besides, recent research on triangle enumeration with graph updates focuses more on approximate algorithms and CPU-only environments. Therefore, we design `TEAF` to support both efficient triangle enumeration and fast graph updates to fill the research gaps. As mentioned in Section 2, the key to efficient dynamic triangle counting is to design a graph data storage technique for the neighbor lists and guarantee fast scan and update throughput. Thus, this section will discuss how `TEAF` stores a dynamic graph and handles batch updates.

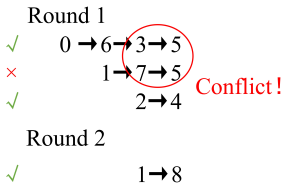
6.1 Dynamic Graph Storage

The widely-used CSR structure might not be a good choice for supporting graph updates. A typical CSR concatenates the neighbor lists and stores them consecutively. Hence, inserting an edge requires moving lots of data to maintain the structure. In the proposed `TEAF`, we hold the light neighbor lists on the CPU side and keep the heavy neighbor lists and light ones connected with a heavy vertex on the FPGA side. Light neighbor lists are sorted lists, and heavy neighbor lists are stored as cuckoo hash tables. To the best of our knowledge, Packed Memory Array(PMA) shows good performance in maintaining dynamic sorted lists on both CPU [10, 9] and GPU [75]. The basic idea of PMA is preserving space for the incoming insertion and managing the space with a balanced binary tree. Therefore, we directly adopt PMA to manage the light neighbor lists.

The heavy neighbor lists are maintained as cuckoo hash tables. Luckily, hash tables are naturally more friendly to insertion and deletion than sorted lists. Therefore, we still use cuckoo hash tables to keep the dynamic heavy neighbor lists. In other words, the data structure of `TEAF` we proposed for triangle enumeration also supports efficient graph updates. The deletion of an edge from a cuckoo-hashed neighbor list is easy with a lazy policy. We can search for the neighbor to be deleted and mark it as invalid. However, recalling the insertion example in Fig 7(b), an insertion may cause numerous rounds of element relocation when the load factor is high. A possible solution is allocating more space to each cuckoo hash table to lower the load factor, but it may lead to an out-of-memory problem for large data graphs. Alternatively, we adopt the two-phase insertion strategy proposed in MemC3 [26], a popular Key-Value storage solution, and improve it for the FPGA hardware.



(a) Finding the cuckoo path



(b) Counter-free update

Fig. 10 FPGA-friendly insertion for cuckoo tables

6.2 FPGA-friendly Two-phase Insertion

Our improvement focuses on better updates. When inserting an element into a cuckoo hash table, the system keeps looking for a vacant position and swapping the elements. The two-phase insertion strategy separates the location detecting and data moving into two phases. It first searches for an available position without moving the elements. As shown in Fig 10(a), to insert a new element 5 into $N(0)$, the system tries to move the element

at positions 0,6, and 3 successively and finally discovers an empty position 5. Such a search path is named a cuckoo path. With the cuckoo path, the system moves the element in reverse order. The system first swaps elements in positions 3 and 5, swaps those in positions 6 and 3, exchanges elements in positions 6 and 3, and finally writes the new element into position 0.

In the original design of the Two-phase Insertion, we should keep a version counter for every element to implement optimistic locking. However, these counters require extra memory cost on FPGA. Therefore, we propose a counter-free updating strategy demonstrated in Fig 10(b). Generally speaking, it is a multi-round update. We first generate all cuckoo paths in parallel at each round and store them on BRAM. Then, we start from the end of every path and check for conflicts. We postpone the execution of the shorter conflicting cuckoo path to the next round. In this way, we sacrifice certain parallelism to avoid locks and version counters. Since we can store the hash tables on BRAM, its fast IO bandwidth can somewhat amend the parallelism loss.

Fig 10(b) gives a running example of the data moving phase. Three insertions are required, and the system generates three cuckoo paths in parallel. Then, it starts the conflict check in reverse order of the cuckoo path search phase. In our example, a conflict is detected between two paths that try to fill position 5 with elements from different locations. The path $1 \leftarrow 7 \leftarrow 5$ is delayed to the next round because it's shorter.

7 Experiment

7.1 Experimental Setup

The experiments were conducted on Linux servers, each with sufficient main memory to store the entire graph data. Our FPGA PUs are implemented on a Xilinx Alevo-U200 FPGA board, which has 64GB of DRAM and 35MB of on-chip memory (7MB of BRAM and 28MB of URAM). The PUs are all implemented using the Xilinx Vitis Development Kit[4]. The performance evaluation implementation consists of 12 PUs. We employ 12 PUs, although more PUs can be used because we found that latency becomes unstable on large graphs. Our GPU-based systems are tested using Nvidia V100 and CUDA 11.3 to compile the GPU projects. To evaluate the CPU-based systems, we use two Intel XEON Gold 6326 CPUs. A default target load factor of 0.9 is applied. The S cache size is set to 4MB for small graphs (RE, FN, PA, LJ, KR18) and 16MB for large graphs (KR21, Orkut, TW). The D cache size for each PU is 256KB for small graphs and 512KB for large graphs.

[4] https://www.xilinx.com/products/design-tools/vitis.html

Table 2 Details of Datasets

dataset	$\|V\|$	$\|E\|$	triangles	type[a]
roadNet(RN)	1.9M	5.5M	120,676	r
flickr(FE)	0.1M	4.6M	107,981,213	r
patent(PA)	6M	17M	7,515,023	r
com-lj(LJ)	4M	34M	177,820,130	r
kron18(KR18)	3M	25M	281,814,846	s
kron21(KR21)	12M	201M	627,584,181	s
orkut(OK)	3M	117M	1,765,053,740	r
twitter(TW)	62M	1.5B	34,824,916,864	r

[a] Type denote whether the graph data is real-world(r) or synthetic(s)

Our datasets are obtained from SNAP[5] and HPEC graph challenge[6]. The selected datasets are frequently used in previous work[46,39,40,48]. They include both real-world and synthetic datasets. The number of edges ranges from millions to billions to assess the scalability of our system. Table 2 provides further information on our datasets. It is important to note that while we compare the overall performance of various systems with each dataset, we only employ PA, LJ, KR18, KR21, and OK for other tests. RN and FE were excluded due to their small size, as most of our optimization designs are intended for larger graphs. TW is omitted except for the scalability evaluation because some compared systems cannot handle a billion-scale graph.

For the baseline, we implement the CPU-only and parallel version of `TEAF`[7], denoted as `TEAF_CPU`. It uses the same data structure and intersection method with `TEAF`. Another baseline is a parallel forward algorithm using the merge-based intersection. Both baselines are compiled with O3 optimization. We also compare `TEAF` with previous CPU, FPGA, and GPU solutions, as displayed in Table 3. All of the compared methods are written in C++ and CUDA. Among them, `LOTUS_CPU` and `Huang_FPGA` are implemented by ourselves because we can't find any available public codes. We obtain the reproduced code of `Tricore_GPU` from other researchers and use the open-source version of `Rapid_CPU`, `Trust_GPU`, and `Gunrock_GPU`. Apart from comparing `TEAF` with other systems, we evaluate our list intersection strategy by testing it against the merging only, binary-search only, and a bitmap-based method proposed in [71], which is the best list intersection implementation on FPGA we can find. Moreover, we compare our update technique with those from MemC3 [26] and

[5] http://snap.stanford.edu/data/
[6] https://graphchallenge.mit.edu/data-sets
[7] https://github.com/pkumod/TEAF

[97]. To prove the advantage we mentioned in Section 2.3.1, we include the evaluation of updating the compressed bitmap proposed in [36].

Table 3 Compared systems and notations

Notation	Method Description
`FW_CPU`	Parallel Forward algorithm [74]
`TEAF_CPU`	`TEAF` implemented on CPU
`LOTUS_CPU`	SOTA CPU-based solution[25]
`Rapid_CPU`	SOTA solution [79] for subgraph matching
`Huang_FPGA`	Huang's work [48] on FPGA
`Tricore_GPU`	Uses binary-search based intersection [46]
`Trust_GPU`	Uses hashing-based intersection [69]
`Gunrock_GPU`	A widely-used graph framework [23]

7.2 Comparison of preprocessing time

Before going to performance comparison, we first discuss the preprocessing time of the compared systems. All compared systems require various preprocessing steps including sorting the neighbor lists, building particular data structures, constructing auxiliary indices, and reordering the vertices. The time for preprocessing is not included in the running time in the following sections.

Table 4 Preprocessing time in ms

data	Sort&CSR*	Lotus	Tricore	Trust	TEAF
PA	41	63	49	45	47
LJ	103	174	151	143	165
KR18	394	475	486	502	432
KR21	1611	2327	2399	2001	2104
OK	592	1087	853	728	731
TW	8264	11775	12534	13463	14872

* Sorting the neighbor lists and building a CSR, as required by `FW_CPU`, `Rapid_CPU`, `Huang_FPGA`, and `Gunrock_CPU`.

Table 4 presents the preprocessing time for `TEAF` and the compared systems. The preprocessing procedure of `FW_CPU`, `Rapid_CPU`, `Huang_FPGA`, and `Gunrock_CPU` is the same: sorting the neighbor lists and constructing a CSR. We denote these operations as Sort&CSR in Table 4. `TEAF` takes 1.23x longer to finish preprocessing than Sort&CSR, and is faster than `LOTUS_CPU` in 5 out of 6 datasets. The preprocessing of `TEAF` is efficient because our design avoids sorting long neighbor lists, and adopts optimized parallel updates to build cuckoo hash tables.

After preprocessing, different data structures are built and stored on various devices (CPUs, GPUs, or FPGAs). Since we use hash tables for heavy neighbor lists, chances are that the memory consumption will increase. Therefore, we compare the memory usage after the construction in Table 5. Compared with a typical CSR, the data structure of `TEAF` consumes more memory of 1.18x. The space consumption of `TEAF` is minor because we set a target load factor of 0.9. We believe such memory overhead is worthy because this data structure enables `TEAF` to support fast intersection and update at the same time.

Table 5 Space consumption of data structures in MBs

dataset	CSR	TEAF	v.s. CSR
LJ	152.4	188.4	1.23x
KR21	852.7	957.1	1.12x
OK	481.2	582.8	1.21x
TW	6248.9	7214.7	1.15x
avg.	-	-	1.18x

7.3 Comparison with baselines and competitive methods

This section analyzes the performance improvement of `TEAF` over baselines and previous systems on CPU, FPGA, and GPU. We compare the runtime of CPU-based and FPGA-based methods. Regarding GPU-based methods, we discuss the absolute and energy-averaged throughput. GPUs are powerful in parallelism but consume a large amount of energy, while FPGAs achieve higher throughput with less power. Furthermore, we report the resource usage and frequency of FPGA-based systems.

By default, performance is calculated as the average of ten runs. Although the methods compared in this subsection are intended for static graphs, we report the performance of `TEAF` using the proposed data storage for dynamic graphs, including PMA and cuckoo-hashed neighbor lists.

Fig. 11 demonstrates that `TEAF` outperforms all other systems on CPUs and FPGAs in terms of end-to-end runtime, including workload allocation, triangle enumeration execution, and cross-device data transportation. It is important to note that the loading time for large graphs into main memory, which can be quite time-consuming, was excluded from end-to-end runtime for all compared systems.

Compared to the baselines, `TEAF` achieves an average speedup of 2.48x against `TEAF_CPU` (from 1.96x to 3.07x). The performance gap is even more significant between `TEAF` and `FW_CPU`, ranging from 4.56x to 6.36x. Our optimizations on the intersection strategy, such as cuckoo hashing, can also be applied to CPUs to boost `TEAF_CPU`. Additionally, we found that the previous State-of-the-Art solution on FPGA outperforms `FW_CPU` but is outperformed by `TEAF_CPU` and

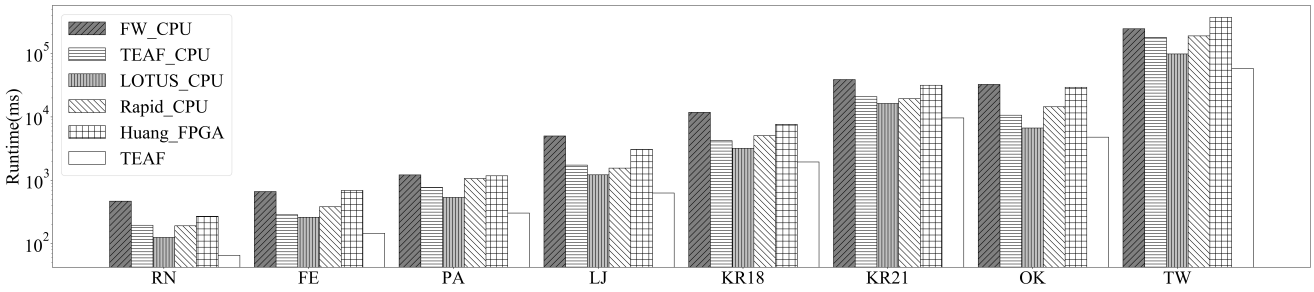


Fig. 11 TEAF v.s. CPU and FPGA solutions in running time

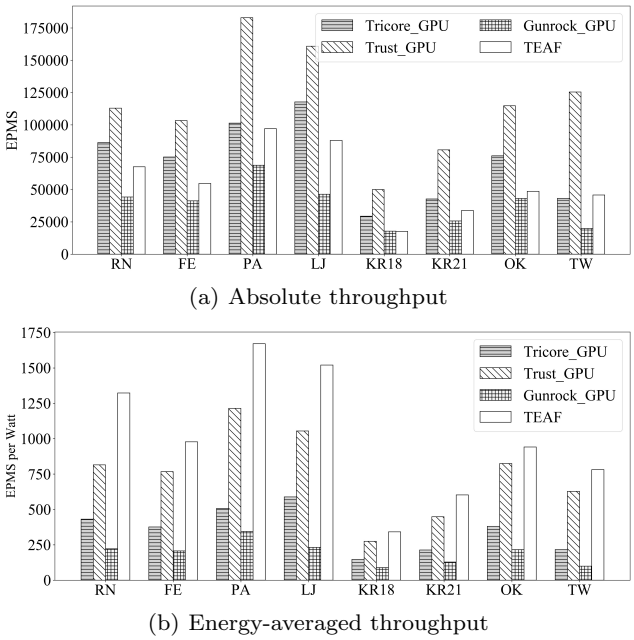
`LOTUS_CPU`. The running frequency of an FPGA is usually much lower than that of a state-of-the-art CPU. If massive parallelism and efficient memory access patterns cannot be achieved, a hybrid solution may be inferior to a CPU-only approach. However, `TEAF` leverages the potential of FPGAs as co-processors for CPUs and therefore outperforms all CPU-only baselines.

Table 6 Resource utilization of FPGA-based methods with the best performance

Method	LUT	Register	BRAM	Frequency
TEAF	36.40%	17.58%	61.09%	255MHz
Huang	28.17%	12.94%	53.72%	241MHz

Compared to FPGA-based systems, `TEAF` is up to 6.14x faster than `Huang_FPGA`, and 4.47x faster on average. `Huang_FPGA` has more difficulty dealing with larger graphs, such as TW and OK, due to its merge-based intersection design struggling with huge neighbor lists. In contrast, `TEAF` utilizes an efficient CPU-FPGA co-processing and caching mechanism to save a significant amount of time-consuming off-chip data transfer. In addition, we utilize the cuckoo-hash-based intersection method, which is better suited for intersecting large neighbor lists and achieving high levels of parallelism. Once the hash table is constructed, the search complexity for an element becomes independent of the list size. Additionally, the search operations for different targets are completely independent.

Apart from evaluating the running time, we also analyze the resource usage and implemented frequency of the FPGA-based methods. For `Huang_FPGA`, we use the data from an implementation of our own because the original code is not open-source, and the required data were not reported in [48]. As shown in Table 6, both `TEAF` and `Huang_FPGA` are mainly bounded by the size of BRAM because they load the intersecting neighbor lists to on-chip memory. However, `TEAF` makes better use of BRAM by designing customized caches for neighbor lists. This accounts for the increase in LUT and BRAM utilization. Besides, the achieved frequency of `TEAF` is higher because `Huang_FPGA` is influenced by more I/O conjunction.



(a) Absolute throughput

(b) Energy-averaged throughput

Fig. 12 Absolute and energy-averaged throughput v.s. GPU solutions

As for CPU-based systems, `LOTUS_CPU` is an up-to-date system optimized for CPU-only systems and exploits targeted triangle enumeration strategies for different vertices. However, `TEAF` still has a superiority of 1.71x on average against `LOTUS_CPU`. Besides, `TEAF` shows more advantages on graphs with higher skewness, thanks to massively parallel FPGA-based processing units carrying the hash-based intersection. In fact, with our CPU-FPGA co-processing mechanism, the performance optimization techniques on CPU and FPGA are orthogonal. We can adopt any CPU-optimized solution to handle the LLI workload. However, for fairness, we conduct parallel merge on the CPU side in our experiments.

Fig 12 illustrates the comparison with GPU systems regarding absolute and energy-averaged throughput by Edges Processed Per Millisecond (EPMS). We can calculate this metric by dividing the number of edges by the product of execution time (ms). The power consumption of the GPU is attained from the Nvidia System management interface (nvidia-smi) provided by its manufacturer. The power consumption of FPGA comes from the Vitis design reports. When running the evaluation of power efficiency, we only consider the power consumption of co-processors (the FPGA and GPU) for two reasons. First, though we can attain the overall power consumption of the CPU or memory, it is not easy to decide what percentage of the power consumption is from the evaluated program, because many processes are running on the server. Second, we ensure the workloads assigned to the FPGA or GPU kernel are the same. Specifically, we slightly modify the workload assignment logic of the GPU-based solutions, instructing them to count triangles derived from HHI and HLI only. In this way, we make sure that our comparison is fair.

FPGA-based systems may not be faster than GPU-based SOTA regarding absolute performance. However, it still achieves higher throughput than `Gunrock_GPU`. For energy-averaged throughput, `TEAF` prevails the GPU systems on all datasets, with an average advantage of 1.51x against Trust_GPU and 2.22x against Tricore_GPU. We believe the comparison of throughput per watt in our experiments is worthwhile. Reducing power consumption is drawing increasing attention, especially from the industry.

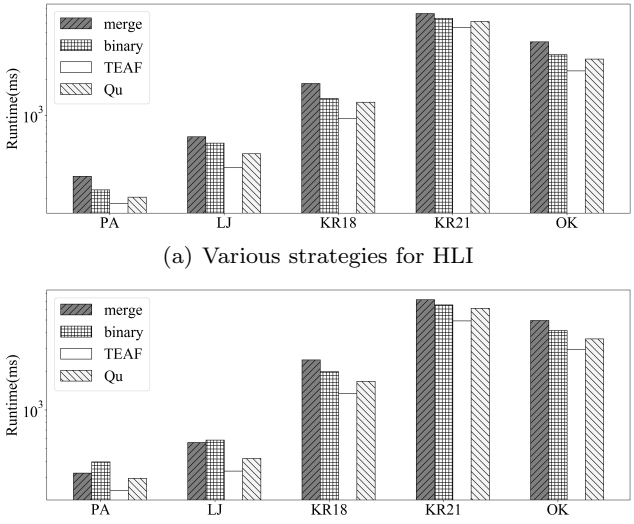
7.4 Evaluation on system design choices

This section compares `TEAF` under different settings to prove the effectiveness of our system design choices. We focus on the effect of intersection strategies and the vertex classification algorithm.

7.4.1 Impact of the intersection strategy.

The results in Fig 13 illustrate that our intersection strategy is better for CPU-FPGA heterogeneous platforms. We compare the runtime of the processing units using different intersection algorithms for HHI and HLI. The runtime of `TEAF` includes the time for building cuckoo hash tables. We omit the LLI experiments since we moved this workload to the CPU.

Compared with the baselines, for HLI, `TEAF` outperforms the system using merely merging by 72% on average and binary search by 39%. For HHI, our lead remains 59% against merging and 52% against binary search. Our advantage is slightly lower because empty elements in the cuckoo hash table do affect our performance to an acceptable extent. The gap between merging and binary search is narrower on HHI than on HLI because merging is more suitable for intersecting two lists of similar sizes. However, the large lists still limit its parallelism. Merging features sequential memory access pattern and linear complexity, making it suitable for dealing with LLI on CPU.



(a) Various strategies for HLI

(b) Various strategies for HHI

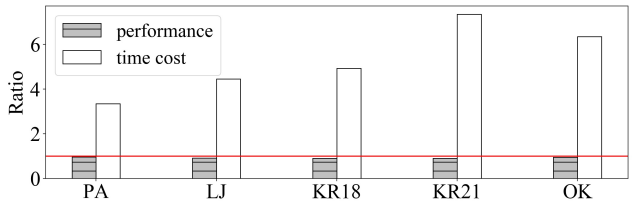
Fig. 13 Runtime evaluation for intersection strategies

Compared with the list intersection solution on FPGA by Qu et al., `TEAF` is 1.46x faster for HLI and 1.23x for HHI. Qu's solution first converts the neighbor list into a bitmap with $|V|$ bits and partitions the bitmap into fixed-sized blocks. The blocks where all bits are zero are removed. The remaining partitions and an ID as identification are stored as a CSR. The parallelism of Qu's solution is not as high as ours because only inter-intersection parallelism is implemented. Worse, since the compressed bitmaps are stored in a CSR, it's costly to insert a new edge when it does not belong to any existing partitions. By contrast, our design supports parallelism between and within different intersection tasks. Moreover, our solution requires less preprocessing time than our competitor.

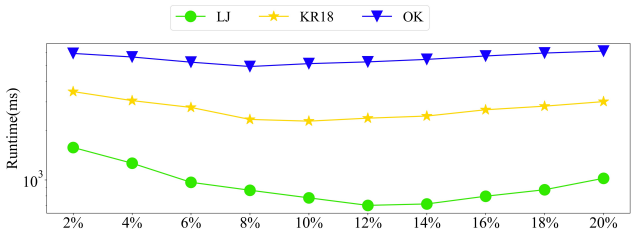
7.4.2 Effect of optimization methods for M_h.

In Section 4, we mention that the selection of M_h significantly influences system performance. Fig 14(b) demonstrates its effect. We gradually increase the value of M_h and record the end-to-end runtime of `TEAF` on three of our datasets. KR18 and OK are more irregular, and the M_h reaches the optimal earlier. The x-coordinate

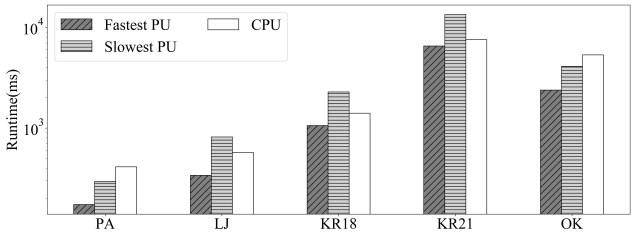
in Fig 14(b) is the ratio between $|V|$ and M_h. At first, M_h is a small number. When it increases, at first, the runtime decreases because more long neighbor lists are stored as cuckoo hash tables, and the workload is more balanced. However, when M_h continues growing, many short neighbor lists are also considered heavy. It causes a severe imbalanced workload. What's more, we notice that on different datasets, the best choice of M_h also varies, telling us it is necessary to use an adjustable value of M_h attained by data-driven algorithms instead of a fixed value.



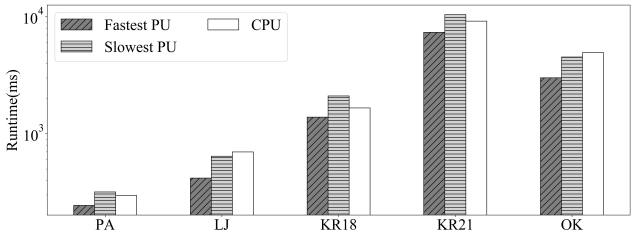
(a) Brute search v.s. our algorithm. We use the brute search as a baseline and calculate the ratio of speedup of preprocessing time or the loss of performance.



(b) Runtime with various M_h. The x-coordinate is the ratio between $|V|$ and M_h and the y-coordinate is the exact runtime.



(c) Workload balance using fixed $M_h = 0.1|V|$



(d) Workload balance using flexible M_h with our proposed method

Fig. 14 Evaluation on the impact and selection of M_h

So the critical factor becomes whether it is worth using a brute search to get closer to the global optimal M_h. According to Fig 14(a), the answer is negative. We compare the two methods to determine M_h in terms of performance gain and time cost. We calculate two ratios: performance ratio and time cost ratio. The former is the ratio between the system runtime achieved by brute search and our heuristic algorithm. The latter is the division of the actual runtime for the two algorithms to finish. We use our proposed algorithm as a denominator for both ratios. Even though the brute force algorithm can get a better M_h, the performance gap is minimal, which means the proposed heuristic algorithm can get a good enough result. However, the time cost reduced by the heuristic algorithm is significant, and the advantage is more evident on larger graphs. In practice, we can use the brute search solution for small graphs and switch to the heuristic algorithm to handle large graphs. Fig. 14(c) and Fig. 14(d) compare the runtime gap among the fastest PU, the slowest PU and the CPU processing. With the proposed flexible M_h estimation, this gap decreases by 14%.

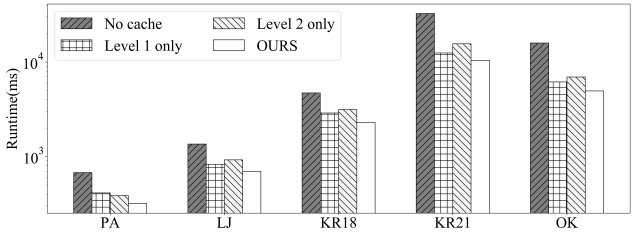
7.5 Evaluation on hardware-related optimization

This section focuses on the three optimizations proposed in Section 5: graph-structure-aware caching, CPU-FPGA co-processing, and pipelining. These techniques consider the features of the FPGA to boost the performance of the processing units on the heterogeneous device. Therefore, we can say they are heterogeneous-friendly optimizations.

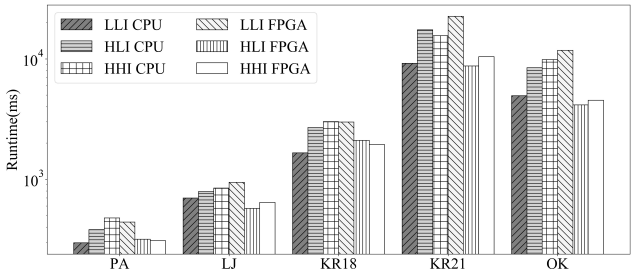
7.5.1 Impact of graph-structure-aware caching.

Fig 15(a) presents the performance of `TEAF` with various cache settings. Experimental results indicate that `TEAF` is 2.49x faster than the system without caching. The advantage of larger graphs like KR21 and OK is higher than 3x because memory cost contributes more to total runtime when the graph is large. The hybrid caching technique is better than using only one caching mechanism. We are 1.35x faster than only the S Cache and 1.58x than only the D Cache. Comparing the cache hit rate in Table 7, we find that the S cache is more useful on graphs of higher skewness because the high-degree vertices bring much more workload.

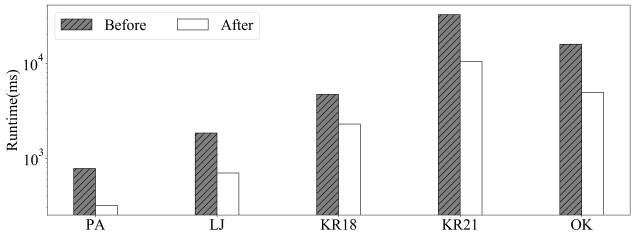
Note that the S Cache maintains a consistent set of the largest neighbor lists, and every S Cache hit saves a lot of off-chip memory access. However, the number of lists in the S Cache is limited. When two neighbor lists are required in an intersection, the performance may be bounded by the neighbor lists absent in the S Cache.



(a) Evaluation of different cache settings



(b) Running time of HHI, HLI, and LLI on the CPU and FPGA



(c) Evaluation of pipelining

Fig. 15 Evaluation on hardware-related optimization

The D Cache and workload allocation mechanism can alleviate this problem.

Table 7 Cache hit rate on various datasets

Dataset	S cache	D cache
PA	29%	36%
LJ	54%	27%
KR18	63%	31%
KR21	65%	29%
OK	59%	44%

7.5.2 Impact of CPU-FPGA co-processing.

Fig 15(b) compares the running time of HHI, HLI, and LLI on a CPU and FPGA, respectively. From the results, we have two observations. First, LLI on CPUs is faster than FPGAs in every dataset, with an average advantage of 1.89x. However, FPGAs outperform CPUs for HHI (1.58x) and HLI(1.61x). The gap between the two devices is wider for larger datasets. This is because we use merge-based intersection for LLI and hashing-based intersection for HLI and HHI. Merge-based intersection scans two lists in order, but hash-based intersection jumps between hash buckets to check the existence of elements. The different memory access pattern makes merge-based intersections a better fit for CPUs. When the graph is large, the locality of doing hash searches on a CPU is worse. Instead, the customized caches and parallel hash search we proposed help the FPGA-based PUs handle hash-based intersection better. Second, the proposed coprocessing technique can improve the overall performance by 2.11x on average. The best improvement is achieved on the largest graph. Our workload allocation strategy for CPU-FPGA co-processing is coarse-grained, simple, but effective. We don't split the workload of LLI, HLI, and HHI because our selection algorithm for M_h can help to balance the workload between the CPU and FPGA.

7.5.3 Impact of pipelining.

As shown in Fig 15(c), applying the pipelining optimization to the processing units can boost our system by 2.69x. The performance gain is more evident on larger graphs because it can significantly reduce the idle time of the modules in a processing unit.

7.5.4 Memory bandwidth utilization

In this subsection, we discuss the achieved memory bandwidth utilization of our implementation to verify the bottleneck of our system.

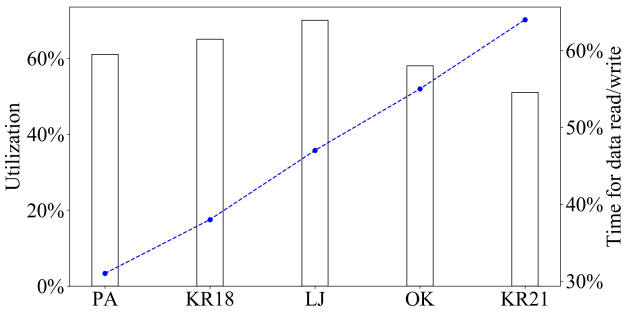


Fig. 16 Memory bandwidth utilization and time cost for reading/writing the memory

Since the profiling and monitoring tools are not as detailed as those for CPUs, we measure the peak memory bandwidth by hand using an example code provided by the hardware manufacturer [8]. Then, we acquire the averaged transfer rate of TEAF from the Vitis report and our own calculation, which is dividing the amount of data we read/write during execution by the running

[8] https://github.com/Xilinx/Vitis_Accel_Examples

Table 8 Evaluation on update strategies

	50-50 Insertion					Pure Insertion					70-30 Insertion				
	serial	M3	Han	Zhang	ours	serial	M3	Han	Zhang	ours	serial	M3	Han	Zhang	ours
PA	90	214	173	235	**304**	74	183	158	216	**267**	84	199	166	222	**281**
LJ	85	208	159	217	**272**	72	179	136	201	**214**	79	189	139	209	**252**
KR18	64	196	110	204	**248**	48	166	112	153	**229**	56	171	140	177	**233**
KR20	61	184	99	198	**236**	40	157	121	168	**204**	46	181	153	189	**217**
OK	73	169	104	171	**199**	42	151	135	156	**182**	59	214	161	169	**186**

time. We choose the smaller value of these two numbers. Besides, we also demonstrate the percentage of running time TEAF spends on reading/writing data with the blue polyline for our bottleneck analysis.

As shown in Fig. 16, we achieve an average memory bandwidth utilization of 50.2%. We have two observations on these data: (1) At first, when the scale of data graphs increases (from PA to KR18 to LJ), the memory bandwidth utilization goes up. This is because the amount of data to be accessed grows rapidly as $|V|$ increases. However, the proposed cache optimization and co-processing design lessen the IO pressure upon DRAM, so the ratio of IO time against the total time is less than 40%. (2) When the data scale keeps on growing (LJ and KR21), the memory bandwidth utilization decreases. This is because the size of on-chip memory is limited so more heavy neighbor lists will be resident on DRAM for larger data graphs. Accessing these lists incurs random memory access (checking hash buckets). According to our tests, our device's random access bandwidth is only 24% of the peak bandwidth. Besides, the blue polyline in Fig. 16 shows that the percentage of IO time against the total time is always going up. In other words, the larger the datasets, the more time we spend on reading and writing the data. This observation accords with the conclusion from previous work that triangle enumeration is IO-bounded especially when the data graph is large. [40].

7.6 Evaluation on dynamic graph support

This section focuses on the performance of `TEAF` when dealing with dynamic graphs. We still use the five datasets in subsection 7.4 and 7.5. We are most interested in two metrics: (1) the overall runtime to output the affected triangles after each batch of graph modification operations. (2) The update efficiency of the dynamic graph data structure. For the first metric, note that we evaluate the runtime of `TEAF` using dynamic graph data storage while the compared works are only for static graphs. However, TEAF still prevails over its competitors.

For the second metric, we measure it in the following routine. First, we build the PMA structure and hash tables with the entire data graph. Then, we generate a batch of 10^7 operations randomly and conduct it on the data graph. We launch local triangle enumeration before and after the graph update. Then, we use their sum as the runtime of this round. We repeat the update process a thousand times to get the average values of the two metrics. The results are reported in table 8. The measure of update speed is kilo operations per second (kops). We compare `TEAF` with four competitors: (1) Serial update. (2) the update technique proposed in MemC3 [26], denoted as `M3`

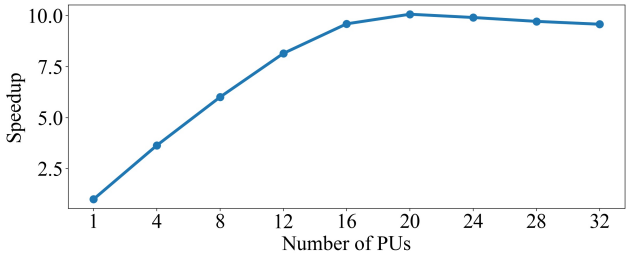
For update speed evaluation, we compare our two-phase update method combining PMA and cuckoo hash table with the serial update strategy and the update technique proposed in MemC3 [26], denoted as `M3`. We record the update speed under two different settings of graph update workload: one consists of half insertion and half deletion (50-50 Insert), only insertion (Pure Insert), and 70% insertion (70-30 Insertion). Lazy deletion is used in all methods, so insertion is more costly than deletion. Our lead in update speed is obvious, reaching a speedup of 3.34x against serial update, 1.26x against `M3`, and 1.22x against `Zhang` on 50-50 Insertion. For Pure insertion, the speedup is 3.93x against serial update, 1.35x against `M3`, and 1.23 against `Zhang`. As for 70-30 insertion, our advantage becomes 3.97x, 1.29x, and 1.20x. All methods suffer performance loss when the ratio of insertion increases, but the decline is the smallest for our method.

Also, we include the update speed of the data structure proposed in Han's work [36] to support our discussion in section 2. As shown in Table 8, the update throughput of Han's work [36] is beaten by `TEAF` on all update batch settings by approximately 1.75x.

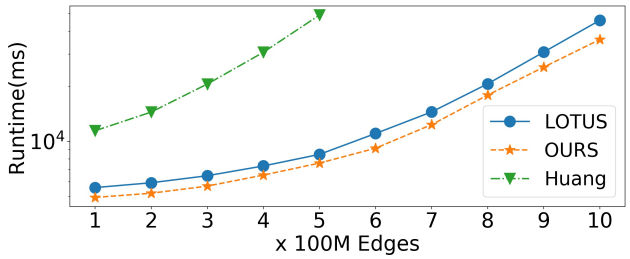
7.7 Scalability

This section tests the scalability of `TEAF` with our largest dataset, TW. Fig 17(a) studies the performance of `TEAF` when the number of processing units (PU)s increases. The number of PUs activated simultaneously for HLI,

and HHI is equal. As we can see in Fig 17(a), at first, the system performance is significantly improved with the increase in PU number. However, performance remains unchanged and is even worse when the growth continues. This is because the memory bandwidth between off-chip memory and on-chip memory limits the number of PUs. Too many running PUs possibly cause stalling memory requests.



(a) Varying the number of PUs



(b) Varying the number of edges

Fig. 17 Scalability evaluation

Fig 17(b) further evaluates the scalability of `TEAF` when the scale of the data graph grows. We randomly sample a fixed number of edges from TW to get a series of data graphs. The edge number begins at 100M and increases by 100M at a time. We record the runtime of `TEAF`,`LOTUS_CPU`, and `Huang_FPGA` on every sampled subgraph of TW. We cut the curve when the runtime exceeds 50,000 ms for a better demonstration effect. `TEAF` and `LOTUS_CPU` achieve a similar trend when the data graph is enlarged, but the curve of `LOTUS_CPU` is always above `TEAF`'s curve. The runtime of `Huang_FPGA` grows much sharper than `Huang_FPGA`.

8 Conclusion

In this paper, we propose a triangle enumeration acceleration system called `TEAF`, which applies an adaptive strategy to handle the computational bottleneck of triangle enumeration, optimized for CPU-FPGA co-processing systems, considering the features of both CPU and FPGA. Furthermore, `TEAF` supports triangle enumeration on both static and dynamic graphs. Extensive experiments on large graphs confirm that `TEAF` outperforms the existing systems on CPU and FPGA and also beat the GPU-based system in terms of energy-averaged performance.

In the future, we will continue to improve triangle enumeration following two potential directions. One is designing or adopting a more meticulous graph data structure for dynamic graphs. The other one is exploring the acceleration in a distributed heterogeneous environment to support efficient triangle enumeration on billion-scaled real-world graphs.

Acknowledgements This work was partially supported by NSFC under grants 61932001 and U20A20174. Lei Zou is the corresponding author of this work.

References

1. Almasri, M., Vasudeva, N., Nagi, R., Xiong, J., mei W. Hwu, W.: Hykernel: A hybrid selection of one/two-phase kernels for triangle counting on gpus. 2021 IEEE High Performance Extreme Computing Conference (HPEC) pp. 1–7 (2021)
2. Ashari, A., Sedaghati, N., Eisenlohr, J., Parthasarathy, S., Sadayappan, P.: Fast sparse matrix-vector multiplication on gpus for graph applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 781–792 (2014)
3. Ashkiani, S., Farach-Colton, M., Owens, J.D.: A dynamic hash table for the gpu. 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS) pp. 419–429 (2017)
4. Ashuthosh, M.R., Krishna, S., Sudarshan, V., Subramaniyan, S., Purnaprajna, M.: Mapparat: A resource constrained fpga-based accelerator for sparse-dense matrix multiplication. 2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID) pp. 102–107 (2022)
5. Azad, A., Buluc, A., Gilbert, J.: Parallel triangle counting and enumeration using matrix algebra. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 804–811 (2015)
6. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: ACM-SIAM Symposium on Discrete Algorithms (2002)
7. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 16–24 (2008)
8. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient algorithms for large-scale local triangle counting. ACM Trans. Knowl. Discov. Data **4**(3), 13:1–13:28 (2010)
9. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious b-trees. SIAM J. Comput. **35**(2), 341–358 (2005)
10. Bender, M.A., Hu, H.: An adaptive packed-memory array. In: S. Vansummeren (ed.) Proceedings of the

Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA, pp. 20–29. ACM (2006)
11. Bhattarai, B., Liu, H., Huang, H.H.: Ceci: Compact embedding cluster index for scalable subgraph matching. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1447–1462 (2019)
12. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1199–1214 (2016)
13. Bisson, M., Fatica, M.: High performance exact triangle counting on gpus. IEEE Transactions on Parallel and Distributed Systems **28**, 3501–3510 (2017)
14. Bisson, M., Fatica, M.: Update on static graph challenge on gpu. 2018 IEEE High Performance extreme Computing Conference (HPEC) pp. 1–8 (2018)
15. Buriol, L.S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. In: ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (2006)
16. Busato, F., Green, O., Bombieri, N., Bader, D.A.: Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), pp. 1–7. IEEE (2018)
17. Che, Y., Lai, Z., Sun, S., Luo, Q., Wang, Y.: Accelerating all-edge common neighbor counting on three processors. Proceedings of the 48th International Conference on Parallel Processing (2019)
18. Chen, X., Tan, H., Chen, Y., He, B., Wong, W.F., Chen, D.: Thundergp: Hls-based graph processing framework on fpgas. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 69–80 (2021)
19. Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., Chen, E.: Kineograph: taking the pulse of a fast-changing and connected world. In: European Conference on Computer Systems (2012)
20. Coleman, J.S.: Social capital in the creation of human capital. American Journal of Sociology **94**, S95 – S120 (1988)
21. Dai, G., Chi, Y., Wang, Y., Yang, H.: Fpgp: Graph processing framework on fpga a case study of breadth-first search. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 105–110 (2016)
22. Dai, G., Huang, T., Chi, Y., Xu, N., Wang, Y., Yang, H.: Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 217–226 (2017)
23. Davidson, Andrew, Yuduo, Wang, Yangzihao, Riffel, Andy, Owens, John, D., Pan: Gunrock: A high-performance graph processing library on the gpu. ACM SIGPLAN Notices: A Monthly Publication of the Special Interest Group on Programming Languages (2015)
24. Davis, T.A.: Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. 2018 IEEE High Performance extreme Computing Conference (HPEC) pp. 1–6 (2018)
25. Esfahani, M.K., Kilpatrick, P., Vandierendonck, H.: LOTUS: locality optimizing triangle counting. In: J. Lee, K. Agrawal, M.F. Spear (eds.) PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022, pp. 219–233. ACM (2022)
26. Fan, B., Andersen, D.G., Kaminsky, M.: Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In: N. Feamster, J.C. Mogul (eds.) Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013, pp. 371–384. USENIX Association (2013)
27. Feng, G., Ma, Z., Li, D., Zhu, X., Cai, Y., Han, W., Chen, W.: Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. Proceedings of the 2021 International Conference on Management of Data (2020)
28. Fuchs, P., Giceva, J., Margan, D.: Sortledton: a universal, transactional graph data structure. Proc. VLDB Endow. **15**, 1173–1186 (2022)
29. Giechaskiel, I., Panagopoulos, G., Yoneki, E.: Pdtl: Parallel and distributed triangle listing for massive graphs. 2015 44th International Conference on Parallel Processing pp. 370–379 (2015)
30. Gou, X., Zou, L.: Sliding window-based approximate triangle counting over streaming graphs with duplicate edges. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pp. 645–657. ACM (2021)
31. Gou, X., Zou, L.: Sliding window-based approximate triangle counting with bounded memory usage. The VLDB Journal (2023)
32. Green, O., Bader, D.A.: custinger: Supporting dynamic graph algorithms for gpus. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2016)
33. Green, O., Fox, J., Watkins, A., Tripathy, A., Gabert, K., Kim, E., An, X., Aatish, K., Bader, D.A.: Logarithmic radix binning and vectorized triangle counting. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), pp. 1–7 (2018)
34. Green, O., Yalamanchili, P., Munguía, L.M.: Fast triangle counting on the gpu. In: Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms, pp. 1–8 (2014)
35. Han, M., Kim, H., Gu, G., Park, K., Han, W.S.: Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1429–1446 (2019)
36. Han, S., Zou, L., Yu, J.X.: Speeding up set intersections in graph algorithms using simd instructions. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1587–1602 (2018)
37. Han, W.S., Lee, J., Lee, J.H.: Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 337–348 (2013)
38. Hoang, L., Jatala, V., Chen, X., Agarwal, U., Dathathri, R., Gill, G., Pingali, K.: Disttc: High performance distributed triangle counting. In: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2019)
39. Hu, L., Guan, N., Zou, L.: Triangle counting on gpu using fine-grained task distribution. In: 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW), pp. 225–232 (2019)
40. Hu, L., Zou, L., Liu, Y.: Accelerating triangle counting on gpu. In: Proceedings of the 2021 International Conference on Management of Data, pp. 736–748 (2021)

41. Hu, X., Qiao, M., Tao, Y.: Join dependency testing, loomis-whitney join, and triangle enumeration. Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (2015)
42. Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: ACM SIGMOD Conference (2013)
43. Hu, X., Tao, Y., Chung, C.W.: I/o-efficient algorithms on triangle listing and counting. ACM Transactions on Database Systems (TODS) **39**, 1 – 30 (2014)
44. Hu, Y., Kumar, P., Swope, G., Huang, H.H.: Trix: Triangle counting at extreme scale. 2017 IEEE High Performance Extreme Computing Conference (HPEC) pp. 1–7 (2017)
45. Hu, Y., Liu, H., Huang, H.H.: High-performance triangle counting on gpus. 2018 IEEE High Performance extreme Computing Conference (HPEC) pp. 1–5 (2018)
46. Hu, Y., Liu, H., Huang, H.H.: Tricore: parallel triangle counting on gpus. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 14 (2018)
47. Huang, J., Wang, H., Fei, X., Wang, X., Chen, W.: Tc-stream: Large-scale graph triangle counting on a single machine using gpus. IEEE Transactions on Parallel and Distributed Systems **33**(11), 3067–3078 (2022)
48. Huang, S., El-Hadedy, M., Hao, C., Li, Q., Mailthody, V.S., Date, K., Xiong, J., Chen, D., Nagi, R., mei Hwu, W.: Triangle counting and truss decomposition using fpga. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), p. 8547536 (2018)
49. Huang, X., Cheng, H., Qin, L., Tian, W., Yu, J.X.: Querying k-truss community in large and dynamic graphs. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1311–1322 (2014)
50. Jabbour, S., Mhadbhi, N., Raddaoui, B., Sais, L.: Triangle-driven community detection in large graphs using propositional satisfiability. In: 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA), pp. 437–444 (2018)
51. Jowhari, H., Ghodsi, M.: New streaming algorithms for counting triangles in graphs. In: International Computing and Combinatorics Conference (2005)
52. Kanewala, T.A., Zalewski, M., Lumsdaine, A.: Distributed, shared-memory parallel triangle counting. In: Proceedings of the Platform for Advanced Scientific Computing Conference, p. 5 (2018)
53. King, J., Gilray, T., Kirby, R.M., Might, M.: Dynamic sparse-matrix allocation on gpus. In: International Conference on High Performance Computing, pp. 61–80. Springer (2016)
54. Kolda, T.G., Pinar, A., Plantenga, T.D., Seshadhri, C., Task, C.: Counting triangles in massive graphs with mapreduce. SIAM Journal on Scientific Computing **36**(5) (2014)
55. Kumar, P., Huang, H.H.: G-store: High-performance graph store for trillion-edge processing. SC16: International Conference for High Performance Computing, Networking, Storage and Analysis pp. 830–841 (2016)
56. Lai, Z., Peng, Y., Yang, S., Lin, X., Zhang, W.: Pefp: Efficient k-hop constrained s-t simple path enumeration on fpga. arXiv preprint arXiv:2012.11128 (2020)
57. Leo, D.D., Boncz, P.A.: Teseo and the analysis of structural dynamic graphs. Proc. VLDB Endow. **14**, 1053–1066 (2021)
58. Lim, Y., Jung, M., Kang, U.: Memory-efficient and accurate sampling for counting local triangles in graph streams. ACM Transactions on Knowledge Discovery from Data (TKDD) **12**, 1 – 28 (2018)
59. Low, T.M., Rao, V.N., Lee, M.K.F., Popovici, D.T., Franchetti, F., McMillan, S.: First look: Linear algebra-based triangle counting without matrix multiplication. 2017 IEEE High Performance Extreme Computing Conference (HPEC) pp. 1–6 (2017)
60. M, M., S, N., S, K.: Bandwidth-efficient sparse matrix multiplier architecture for deep neural networks on fpga. 2021 IEEE 34th International System-on-Chip Conference (SOCC) pp. 7–12 (2021)
61. Macko, P., Marathe, V.J., Margo, D.W., Seltzer, M.I.: Llama: Efficient graph analytics using large multiversioned arrays. In: IEEE International Conference on Data Engineering (2015)
62. Matias, Y., Vishkin, U.: On parallel hashing and integer sorting. J. Algorithms **12**, 573–606 (1991)
63. Mhedhbi, A., Salihoglu, S.: Optimizing subgraph queries by combining binary and worst-case optimal joins. arXiv preprint arXiv:1903.02076 (2019)
64. Milo, R., Shen-Orr, S.S., Itzkovitz, S., Kashtan, N., Chklovskii, D.B., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298 5594**, 824–7 (2002)
65. Ortmann, M., Brandes, U.: Triangle listing algorithms: Back from the diversion. In: Workshop on Algorithm Engineering and Experimentation (2014)
66. Oyarzun, G., Peyrolon, D., Álvarez, C., Martorell, X.: An fpga cached sparse matrix vector product (spmv) for unstructured computational fluid dynamics simulations. ArXiv **abs/2107.12371** (2021)
67. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: F.M. auf der Heide (ed.) Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings, *Lecture Notes in Computer Science*, vol. 2161, pp. 121–133. Springer (2001)
68. Pandey, P., Wheatman, B., Xu, H., Buluç, A.: Terrace: A hierarchical graph container for skewed dynamic graphs. Proceedings of the 2021 International Conference on Management of Data (2021)
69. Pandey, S., Wang, Z., Zhong, S., Tian, C., Zheng, B., Li, X., Li, L., Hoisie, A., Ding, C., Li, D., Liu, H.: Trust: Triangle counting reloaded on gpus. IEEE Trans. Parallel Distributed Syst. **32**(11), 2646–2660 (2021)
70. Pligouroudis, M., Nuno, R.A.G., Kazmierski, T.J.: Modified compressed sparse row format for accelerated fpga-based sparse matrix multiplication. 2020 IEEE International Symposium on Circuits and Systems (ISCAS) pp. 1–5 (2020)
71. Qu, Y., Prasanna, V.K.: Fast online set intersection for network processing on fpga. IEEE Transactions on Parallel and Distributed Systems **27**, 3214–3225 (2016)
72. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. Proceedings of the VLDB Endowment **8**(5), 617–628 (2015)
73. Samsi, S., Kepner, J., Gadepally, V., Hurley, M., Jones, M., Kao, E., Mohindra, S., Reuther, A., Smith, S., Song, W., Staheli, D., Monticciolo, P.: Graphchallenge.org triangle counting performance. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–9 (2020)
74. Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: WEA'05 Proceedings of the 4th international conference on Experimental and Efficient Algorithms, pp. 606–609 (2005)

75. Sha, M., Li, Y., He, B., Tan, K.: Accelerating dynamic graph analytics on gpus. Proc. VLDB Endow. **11**(1), 107–120 (2017)
76. Shun, J., Tangwongsan, K.: Multicore triangle computations without tuning. In: 2015 IEEE 31st International Conference on Data Engineering, pp. 149–160 (2015)
77. Stefani, L.D., Epasto, A., Riondato, M., Upfal, E.: Trièst: Counting local and global triangles in fully-dynamic streams with fixed memory size. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2016)
78. Strausz, A., Vella, F., Girolamo, S.D., Besta, M., Hoefler, T.: Asynchronous distributed-memory triangle counting and lcc with rma caching. 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) pp. 291–301 (2022)
79. Sun, S., Sun, X., Che, Y., Luo, Q., He, B.: Rapidmatch: a holistic approach to subgraph query processing. Proceedings of the VLDB Endowment **14**(2), 176–188 (2020)
80. Tom, A.S., Karypis, G.: A 2d parallel triangle counting algorithm for distributed-memory architectures. In: Proceedings of the 48th International Conference on Parallel Processing, p. 45 (2019)
81. Tsourakakis, C.E., Drineas, P., Michelakis, E., Koutis, I., Faloutsos, C.: Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. Social Network Analysis and Mining **1**(2), 75–81 (2011)
82. Wang, L., Wang, Y., Yang, C., Owens, J.D.: A comparative study on exact triangle counting algorithms on the gpu. In: Proceedings of the ACM Workshop on High Performance Graph Processing, pp. 1–8 (2016)
83. Wang, P., Qi, Y., Sun, Y., Zhang, X., Tao, J., Guan, X.: Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. Proc. VLDB Endow. **11**, 162–175 (2017)
84. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**, 440–442 (1998)
85. Welles, B.F., Devender, A.V., Contractor, N.S.: Is a friend a friend?: investigating the structure of friendship networks in virtual worlds. CHI '10 Extended Abstracts on Human Factors in Computing Systems (2010)
86. Welser, H.T., Gleave, E., Fisher, D., Smith, M.A.: Visualizing the signatures of social roles in online discussion groups. J. Soc. Struct. **8** (2007)
87. Winter, M., Mlakar, D., Zayer, R., Seidel, H.P., Steinberger, M.: faimgraph: high performance management of fully-dynamic graphs under tight memory constraints on the gpu. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 754–766. IEEE (2018)
88. Wolf, M.M., Deveci, M., Berry, J.W., Hammond, S.D., Rajamanickam, S.: Fast linear algebra-based triangle counting with kokkoskernels. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2017)
89. Yang, H., Su, H., Lan, Q., Wen, M., Zhang, C.: Hpgraph: High-performance graph analytics with productivity on the gpu. Scientific Programming **2018**(PT.2), 1–11 (2018)
90. Yao, P., Zheng, L., Zeng, Z., Huang, Y., Gui, C., Liao, X., Jin, H., Xue, J.: A locality-aware energy-efficient accelerator for graph mining applications. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 895–907. IEEE (2020)
91. Yasar, A., Rajamanickam, S., Berry, J.W., Wolf, M.M., Young, J.S., Çatalyürek, Ü.V.: Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments : (update on static graph challenge). 2019 IEEE High Performance Extreme Computing Conference (HPEC) pp. 1–4 (2019)
92. Yasar, A., Rajamanickam, S., Wolf, M.M., Berry, J.W., Çatalyürek, Ü.V.: Fast triangle counting using cilk. 2018 IEEE High Performance extreme Computing Conference (HPEC) pp. 1–7 (2018)
93. Yu, M., Qin, L., Zhang, Y., Zhang, W., Lin, X.: DPTL+: efficient parallel triangle listing on batch-dynamic graphs. In: 37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, pp. 1332–1343. IEEE (2021)
94. Zeng, L., Yang, K., Cai, H., Zhou, J., Zhao, R., Chen, X.: Htc: Hybrid vertex-parallel and edge-parallel triangle counting. 2022 IEEE High Performance Extreme Computing Conference (HPEC) pp. 1–7 (2022)
95. Zhang, J., Khoram, S., Li, J.: Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 207–216 (2017)
96. Zhang, Y., Jiang, H., Wang, F., Hua, Y., Feng, D., Xu, X.: Litete: Lightweight, communication-efficient distributed-memory triangle enumerating. IEEE Access **7**, 26,294–26,306 (2019)
97. Zhang, Z., Zhang, Y., Shi, C.J.R.: High-throughput cuckoo hashing accelerator on fpga using one-step bfs. 2021 IEEE 4th International Conference on Electronics Technology (ICET) pp. 313–317 (2021)
98. Zheng, D., Mhembere, D., Burns, R.C., Vogelstein, J.T., Priebe, C.E., Szalay, A.S.: Flashgraph: Processing billion-node graphs on an array of commodity ssds. USENIX Association (2015)
99. Zhou, S., Kannan, R., Prasanna, V.K., Seetharaman, G., Wu, Q.: Hitgraph: High-throughput graph processing framework on fpga. IEEE Transactions on Parallel and Distributed Systems **30**(10), 2249–2264 (2019)
100. Zhou, S., Prasanna, V.K.: Accelerating graph analytics on cpu-fpga heterogeneous platform. In: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 137–144 (2017)
101. Zhu, X., Feng, G., Serafini, M., Ma, X., Yu, J., Xie, L., Aboulnaga, A., Chen, W.: Livegraph: A transactional graph storage system with purely sequential adjacency list scans. Proc. VLDB Endow. **13**, 1020–1034 (2019)
102. Zou, L., Zhang, F., Lin, Y., Yu, Y.: An efficient data structure for dynamic graph on gpus. IEEE Transactions on Knowledge and Data Engineering **35**(11), 11,051–11,066 (2023)