# A Comparative Study of Intersection-Based Triangle Counting Algorithms on GPUs

Jiangbo Li
*School of MCS*
*Nanchang University*
Nanchang, China
lijiangbo@email.ncu.edu.cn

Zichen Xu*
*School of MCS*
*Nanchang University*
Nanchang, China
xuz@ncu.edu.cn

Minh Pham
*Depart. of CSE*
*University of South Florida*
Tampa, FL, USA
minhpham@usf.edu

Yicheng Tu
*Depart. of CSE*
*University of South Florida*
Tampa, FL, USA
tuy@usf.edu

Qihe Zhou
*Faculty of Data Science*
*City University of Macau*
Macau, China
T20090106129@cityu.mo

***Abstract*—Counting triangles in large graphs, being a crucial problem in graph computing, has attracted significant attention from research communities. There is a large body of work dedicated to algorithmic design and efficient implementation on parallel platforms such as GPUs. Among them, the intersection-based triangle counting algorithm is found to be the most efficient approach and a few GPU implementations have been proposed following this algorithm. However, there remains a gap in understanding how these algorithms perform when confronted with diverse real-world graph datasets. It is a well-established fact that the performance of GPU code is heavily influenced by data characteristics, including graph size and node degree, often leading to issues such as workload imbalances and inefficient memory access. The goal of this study is to systematically evaluate the performance and analyze the behavior of eight recently published intersection-based triangle counting implementations. For that, we developed a unified testing framework that facilitates fast performance assessment of any triangle counting algorithm. Our experiments show that the TRUST algorithm outperforms competitors in most cases. To our surprise, the Polak algorithm, with a simple design, has displayed commendable performance across the board. Notably, it even surpasses the TRUST algorithm when processing small datasets. We conducted an in-depth analysis on the resource consumption patterns of these implementations in relation to their performance and identified key factors that contributed to their behaviors. Based on insights gained from such analysis, we proposed a novel algorithm named GroupTC that delivers outstanding performance under all types of datasets.**

***Index Terms*—GPU, Triangle Counting**

## I. Introduction

Triangle counting (TC), which involves determining the total number of triangles in a graph, holds significant importance in graph data management, finding many applications like k-truss analysis and calculating the clustering coefficient. In recent years, there has been a growing interest in leveraging parallel hardware systems, particularly GPUs, to achieve high performance in processing large-scale graphs. This has resulted in a surge of research efforts, leading to the development of diverse designs and implementations for TC on GPUs.

Three primary approaches are commonly employed in TC: the intersection-based method [17, 6, 3, 11, 5, 9, 15, 8, 16], the matrix multiplication-based method [2, 24, 25], and the subgraph matching-based method [21]. A concise overview of these methods can be found in Section II. Among these approaches, the intersection-based triangle counting (ITC) method is widely regarded as the most effective solution [22, 8, 16]. This is primarily because it can reduce unnecessary computations, leading to improved efficiency over other solutions.

Over the past decade, there has been a continuous stream of GPU implementations of ITC algorithms. Table I provides an overview of the significant proposals in this field. While each publication has presented experimental results, there is still a need to comprehensively assess how these algorithms perform when applied to a variety of real-world graph datasets. As a result, the primary aim of our study is to bridge this gap in understanding by evaluating the performance of these algorithms across diverse real-world graph datasets, using the latest GPU hardware available. This research endeavor seeks to provide a comprehensive and up-to-date perspective on the effectiveness of ITC algorithms in practical scenarios.

The characteristics of the input graph are widely recognized for their significant impact on the efficiency of graph processing algorithms that utilize GPUs [19, 23], and this also holds true for ITC algorithms. Previous work [12] has pointed out that **graph size** and **vertex degree** are salient factors that could change the behavior and performance of ITC algorithms. As the graph size increases, the increasing number of intersections places a higher demand on the algorithm's ability to effectively balance the workload by combining different intersection calculations. This is because real-world graphs often exhibit a power-law distribution of node degree,

[§]J. Li, Z. Xu, and Y. Tu are also associated with Jiaxing Neofelis Technologies Co. Ltd., Jiaxing, Zhejiang, China.

TABLE I
MAJOR ITC ALGORITHMS ON GPUS

| Ref. | Name | Year | Iterator | Intersection | Granularity |
|---|---|---|---|---|---|
| [6] | Green | 2014 | edge | Merge | fine |
| [17] | Polak | 2016 | edge | Merge | coarse |
| [3] | Bisson | 2017 | vertex | BitMap | coarse |
| [11] | TriCore | 2018 | edge | Bin-Search | fine |
| [5] | Fox | 2018 | edge | Merge/Bin-Search | fine |
| [9] | Hu | 2019 | vertex | Bin-Search | fine |
| [15] | H-INDEX | 2019 | edge | Hash | fine |
| [16] | TRUST | 2021 | vertex | Hash | fine |

meaning that larger graphs lead to greater differences in the degree distribution of vertices. Therefore, the performance of the algorithm depends on the ability to satisfy different scales of graphs.

We developed a comprehensive framework that enables effective graph data preparation and performance benchmarking of ITC algorithms on NVIDIA GPUs. Our experiments, conducted on 19 real graph datasets, have yielded intriguing results that deviate from the published data. Intuitively, one might anticipate that newer solutions in Table I generally outperform their older counterparts. On one hand, the most recent TRUST algorithm indeed demonstrates superior performance when processing medium to large-scale datasets. Conversely, the old Polak algorithm, characterized by its straightforward design, emerges as the champion when dealing with smaller datasets (i.e., those with less than 2M edges). All other solutions, despite incorporating innovative concepts in their design, consistently delivered performance that ranged from mediocre to inferior compared to Polak and TRUST. Notably, when processing small datasets, these algorithms can be an order of magnitude slower than Polak and TRUST! In light of these findings, we unequivocally recommend TRUST as the preferred choice for medium to large datasets, while endorsing Polak for small datasets. This, in essence, constitutes the primary contribution of our research.

In addition to the evaluation, this paper makes another significant contribution by conducting a comprehensive analysis of the behaviors of the ITC algorithms when confronted with various types of datasets. Going beyond a mere interpretation of our experimental findings, this endeavor provides invaluable insights into understanding the actual challenges and bottlenecks in the design of efficient ITC algorithms on GPUs. Specifically, by examining the consumption patterns of GPU resources of the ITC algorithms, we have identified the following key factors that strongly influence their performance. (1) **Total amount of work**: This is the main reason behind the success of Polak, in which a linear merge is conducted between neighbor lists while other solutions require hash or index-based merge. (2) **Workload imbalance**: Real-world graphs typically exhibit power-law distributions in node degree, resulting in notable disparities in the amount of work assigned to individual threads. This can lead to substantial thread stalls in an SIMD architecture. (3) **Memory access pattern**: TC algorithms are predominantly limited by memory bandwidth. Sub-optimal memory access patterns (e.g., uncoalesced) can result in low utilization of GPU memory bandwidth.

Building upon the insights we gained from our analysis, we introduce a novel GPU-based ITC algorithm with high performance across a diverse range of data characteristics. It is noteworthy that multiple existing algorithms have already incorporated optimization techniques addressing the three aforementioned factors. Our newly proposed algorithm (named GroupTC) leverages the advantageous features found in existing solutions, resulting in superior performance across a wide spectrum of datasets. Remarkably, GroupTC consistently outperforms Polak in 17 of 19 datasets, achieving a speedup of up to 3.83X. Compared with TRUST, GroupTC exhibits significant improvements when processing small to medium-sized datasets, achieving an impressive speedup of up to 2.92X. For large datasets, GroupTC still performs at a comparable level, with a speedup of 0.94-1.01X. These achievements underline the versatility and effectiveness of the GroupTC algorithm in addressing the challenges posed by different data characteristics.

## II. BACKGROUND

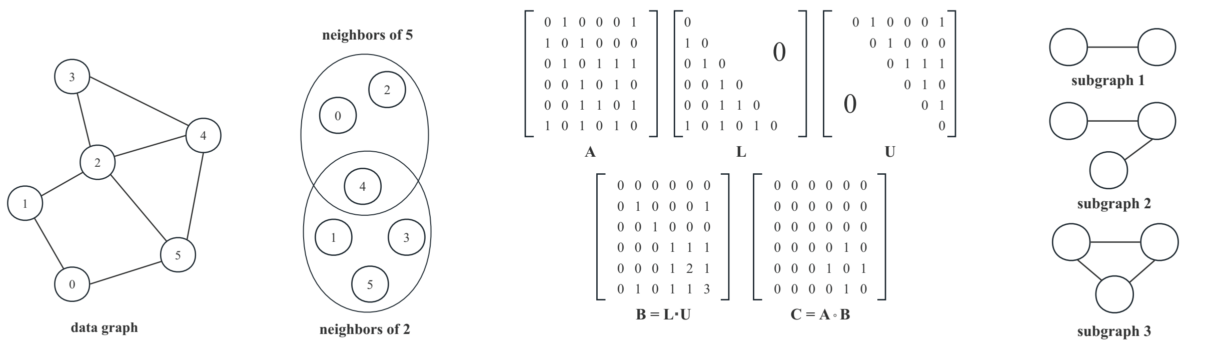
### A. Basic Approaches for Triangle Counting

In this section, we illustrate the main ideas and procedures in the three approaches of triangle counting, namely Intersection, Matrix Multiplication, and Subgraph Matching. Figure 1(a) gives a small graph as an input for the illustrations in figures 1(b), 1(c), and 1(d).

The Intersection approach examines each edge $(u, v)$ in the graph, and computes the intersection $wSet$ between the neighbors of node $u$ and the neighbors of node $v$. For each node $w_i$ in $wSet$, a triangle $(u, v, w_i)$ is formed. The total number of triangles in the graph is the sum of the sizes of all $wSet$ sets. Figure 1(b) shows an example of taking the intersection on the edge connecting nodes *2* and *5*. The neighbors of node *2* are nodes 1, 3, 4, and 5, and the neighbors of node *5* are nodes 0, 2, and 4. The intersection is therefore $\{4\}$. This computation is applied on all edges of the graph.

The Matrix Multiplication approach relies on the matrix $A^n$, ${A^n}_{ij}$ equal the number of walks of length $n$ from vertex $i$ to vertex $j$. Figure 1(c) shows an example where $A$ is the adjacency matrix, $L$ and $U$ is the lower and upper triangular matrix of $A$. We calculate $B = L \cdot U$ where each element $B_{ij}$ represents the number of wedges $i - k - j$. By performing the Hadamard product (element-wise multiplication) $C = A \circ B$, we obtain the number of triangles $i - k - j - i$, and divide the total count by 2 to compensate for the overcounting of each triangle.

The Subgraph Matching approach works by searching for a small query graph within the input data graph, as depicted in Figure 1(d). Initially, a query subgraph that contains a single edge, named $subgraph1$, is searched on the input graph. All candidates in the input data graph that match $subgraph1$



(a) Data graph (b) Intersection (c) Matrix multiplication (d) Subgraph matching

Fig. 1. Three approaches of triangle counting algorithms.

are then obtained. Next, the $Join$ operation is performed to obtain additional candidates that satisfy $subgraph2$, which is a wedge. Similarly, subgraphs that satisfy $subgraph3$ (triangle) are identified.

It is widely recognized that the Intersection approach is superior to the Matrix Multiplication and Subgraph Matching approaches [22, 8, 16], because Matrix Multiplication and Subgraph Matching both perform unavoidable redundant work. Wang et al. [22] performed a comparative study among these three approaches, providing experimental support.

## B. Implementation Choices in Intersection-based Methods

In this subsection, we discuss different methods and implementation choices within the intersection-based approach. To implement the intersection-based approach, one needs to make decisions about the choice for generating the neighbor lists and the method for performing the intersection between two neighbor lists. Furthermore, in a parallel implementation, one also needs to consider the execution granularity, i.e., scheduling the works for a thread or a block of threads.

**Iterator**. There are two ways to generate neighbor lists in the Intersection-based approach: by iterating through vertices or through edges. The vertex-centric method finds triangles by taking the intersection between each vertex's neighbors and the neighbors of its neighbors. This is illustrated in Figure 2 (a). On the other hand, the edge-centric method iterates through edges in the graph and takes the intersection between the two neighbor lists of the two connected vertices. This is illustrated in Figure 2 (b).

**Intersection method**. To find the intersection between two lists, there are four methods: Merge, Binary Search, Hash, and BitMap. The Merge approach uses two pointers, one on each sorted neighbor list. In each iteration, the pointer that points to the smaller value is incremented. When the two values being pointed to are equal, an element of the intersection set is found. The other three methods Binary Search, Hash, and BitMap follow a similar principle. They construct an index from one list and use the other list as the search keys. In the Binary Search method, the index is a binary search tree. In the Hash method, the index is a hash table. In the BitMap method, the index is a bitmap. Finding a search key on the index means we find an element of the intersection.

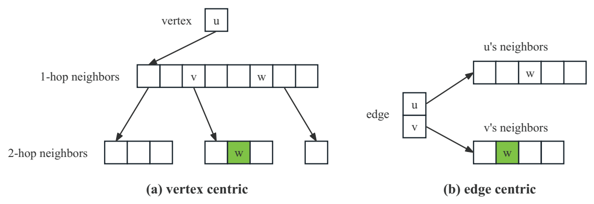


Fig. 2. Iteration method, calculating the number of triangles by iterating from each vertex or iterating from each edge.

**Execution granularity**. In a parallel implementation, it is crucial to determine the amount of work assigned to each thread. Processing a vertex can be viewed as processing multiple edges, we categorize implementations that assign a thread to process neighbor lists around an edge as coarse-grained, and those that employ multiple threads for processing neighbor lists around an edge as fine-grained.

**Pre-processing**. To accurately count the triangles and enhance computational efficiency, selecting or designing a suitable graph pre-processing method is important. Common pre-processing methods include ordering based on node IDs, degree, k-coreness, random ordering, etc. [1, 10]. However, it is not our focus due to page limits.

## C. GPU Architecture

The CUDA (Compute Unified Device Architecture) programming model, developed by NVIDIA, is designed for harnessing the power of GPUs in parallel computing. A warp is a group of 32 threads executed in a SIMT (Single Instruction Multiple Threads) fashion, which serves as the fundamental hardware execution unit. When threads within a warp are given different instructions due to conditional branches or uneven workload distribution, some threads become idle waiting for other threads to finish which leads to suboptimal utilization of GPU resources. A block is a larger computing unit that comprises multiple warps, all threads in a block can be synchronized.

In NVIDIA GPU devices, the *shared memory* acts as a programmable L1-level cache accessible by all threads within the same block. Meanwhile, the *global memory* (also known as device memory) functions as the primary memory source, facilitating communication between threads across different blocks. Notably, compared to typical CPU memory, the global memory bandwidth is significantly higher, reaching up to 900 GB/s in devices such as the NVIDIA Tesla V100. To reach this peak memory bandwidth, programs must utilize *coalesced memory access*, consolidating multiple memory accesses into a single cache line lookup.

## III. Existing ITC Algorithms

In this section, we present major ITC algorithms designed for GPUs, organized chronologically based on their publication dates.

### A. Polak

As a parallel version of the CPU-based Forward algorithm [13, 18], Polak [17] is a merge-based edge-centric triangle counting algorithm that uses one thread to process an edge.

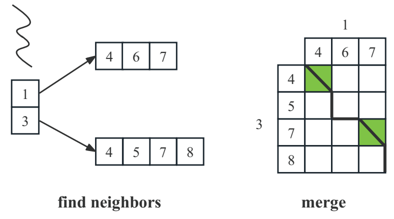


Fig. 3. Overview of the Polak algorithm.

Figure 3 illustrates the processing of a single edge in Polak, which can be divided into two steps. First, the thread index $tid$ is mapped to an edge $(u, v)$, and the neighbor lists of $u$ and $v$ are retrieved; Second, the neighbor lists of $u$ and $v$ are merged in a sequential manner. When the pointers of the two neighbor lists point to the same element (Figure 3, where $4$ and $7$ are marked in green in the merge path), a local variable storing triangle count is incremented by one. The merge operation is accomplished when either of the two neighbor lists are traversed.

Clearly, the computational workload of each thread depends on the length of the neighbor lists. The total amount of work for a thread is $d(u) + d(v)$ where $d()$ is the out-degree of a vertex. Different threads within the same warp will perform merge tasks of varying sizes, and the overall execution time will be determined by the thread with the largest workload. This results in low warp execution efficiency. Additionally, during the merge operation, a single thread accesses adjacent data in the neighbor lists sequentially (rather than multiple threads in the same warp accessing adjacent data simultaneously), leading to lower memory access efficiency.

### B. Green

Green [6] is another edge-centric triangle counting algorithm, which uses a block of threads to process an edge. Green utilizes an efficient parallel merge algorithm extracted from another paper [7]. As compared to Polak, the design focuses on merging large lists to improve the warp execution efficiency.

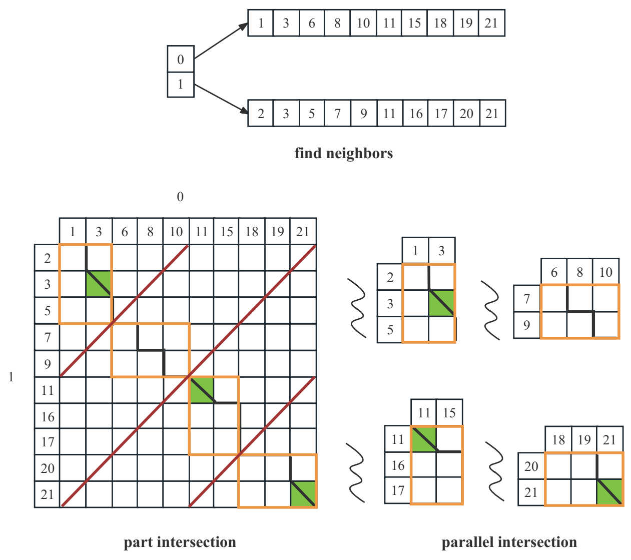


Fig. 4. Overview of the Green algorithm.

Figure 4 shows the process of merging two neighbor lists of the edge of interest in the Green algorithm. First, parallel partition lines (red lines in Figure 4) are defined to divide the merge task into multiple pairs of smaller lists (orange boxes). Such partitioning ensures that the work of merging each pair of small lists is similar. After the partitioning is completed, multiple threads are used for the parallel execution of merging pairs of small lists.

The number of threads dedicated to each edge is empirically set by the user, and such a number is the same for all edges.

However, in real-world graphs, the number of neighbors for each edge is unpredictable, resulting in the coexistence of both large merge lists and small merge lists. This can cause a serious waste of thread resources in case of small lists.

### C. Bisson

Bisson [3] is a vertex-centric triangle counting algorithm that utilizes bitmap for quick lookup operations to find triangles. As illustrated in Figure 5, for each vertex $u$, the algorithm first creates a bitmap that identifies members in the 1-hop neighbor list of $u$, denoted as $N(u)$. Naturally, the length of the bitmap would be the same as the total number of vertices in the graph. For each vertex in $N(u)$, the corresponding bit in the bitmap is set via an atomic operation. After building the bitmap, the neighbor list of each member in $N(u)$ (i.e., a 2-hop neighbor list of $u$) is processed by one thread. Specifically, each match between the 2-hop list and the 1-hop list (the one stored in bitmap) is considered a triangle.

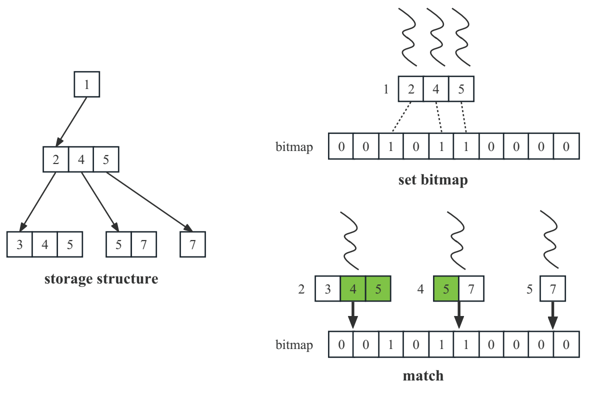


Fig. 5. Overview of the Bisson algorithm.

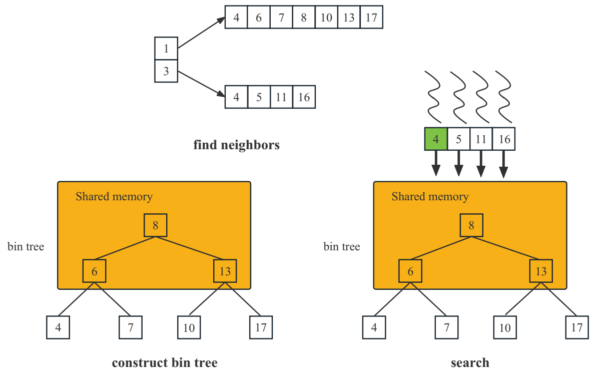


Fig. 6. Overview of the TriCore algorithm.

In addition, Bisson changes the workload assignment based on graph sparsity. When the average out-degree of the vertices is greater than 38, a thread block is used for processing a single vertex. When the degree is between 3.8 and 38, Bisson uses one warp to process one vertex. Otherwise, only one thread is used per vertex. In the first case, when allowed by size, Bisson will store the bitmap in shared memory.

## D. TriCore

TriCore [11] is an edge-centric algorithm where a warp of threads is used to process one edge via a binary search strategy. As shown in Figure 6, the neighbor lists of the two vertices $u$ and $v$ of an edge are first identified. Then a binary tree is built containing the IDs of all members of the the neighbor list of one vertex $u$. Finally, for each neighbor of $v$, its ID is used to search the binary tree, and any match found in the search is considered a new triangle.

For each edge, TriCore chooses the vertex with a longer neighbor list to build the binary search tree to reduce computational time. Furthermore, to reduce global memory access time, TriCore stores as many top levels of the binary tree in shared memory as allowed by shared memory size. During the final matching stage, threads in the same warp will retrieve neighbors stored in adjacent memory locations of the neighbor list to achieve coalesced global memory access.

## E. Fox

Fox [5] is an edge-centric solution that can be viewed as a meta algorithm built on key ideas found in other algorithms such as Green and TriCore. Specifically, Fox chooses between merging and binary search in finding the matches between two neighbor lists based on an estimation of the computational load in processing each edge. The workload of the merge operation for an edge $(u, v)$ is estimated at $d(u) + d(v)$. For bin-search operation, the workload is estimated to be either $d(u) \cdot \log(d(v))$ or $d(v) \cdot \log(d(u))$, depending on the sizes of $d(u)$ and $d(v)$. The algorithm converges to the Green algorithm when merging is to be done. When binary search is to be done, Fox is similar to TriCore except the granularity of workload assigned to a thread is different.

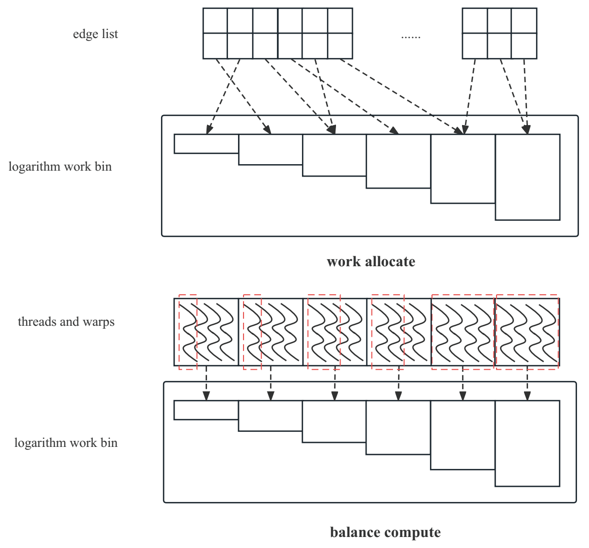


Fig. 7. Overview of the Fox algorithm.

To choose which exact algorithm to use (Figure 7), Fox defines a series of six bins corresponding to exponentially increasing amount of work in processing an edge. Each edge is put into a bin according to an estimation of its workload. For each edge in the $n$-th bin, Fox assigns $2^n$ threads to process it, with a maximum allocation of 32 threads (one warp) to an edge. Each Warp computes the edges within the same work bin, with a workload variation that is often less than 2X (same workload for most cases), resulting in a more balanced workload within the warp.

## F. Hu

Hu's algorithm [9] is a vertex-centric solution that uses one block of threads to process one vertex. The algorithm (Figure 8) utilizes a binary search method for triangle counting and is the first fine-grained (i.e., using multiple threads to process an edge) algorithm with a vertex-centric approach.

For each vertex $u$, the algorithm runs in two steps: 1) *Caching neighbors*: it places as many 1-hop neighbors of $u$ as possible into shared memory; 2) *Fine-grained search*: each thread processes a portion of the 2-hop neighbors of $u$. For each 2-hop neighbor, a binary search is performed to check whether it is found in the 1-hop neighbor list.

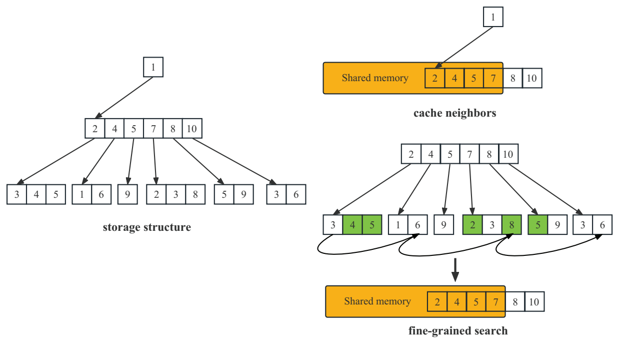


Fig. 8. Overview of Hu's algorithm.

**Algorithm 1:** Fine-grained Search in Hu's algorithm

```
Input: CSR Row: row[], CSR Col: col[], u: A vertex
Output: Triangles of vertex u
1  tc = 0
2  vOffset = threadIdx.x
3  uPoint,uEnd = col[u],col[u+1]
4  while uPoint < uEnd do
5      v = row[uPoint]
6      vPoint = col[v]
7      vDegree = col[v+1] - vPoint
8      // Current v calculation is complete, find next v
9      while uPoint < uEnd and vOffset ≥ vDegree do
10         vOffset -= vDegree
11         v = row[++uPoint]
12         vPoint = col[v]
13         vDegree = col[v+1] - vPoint
14     end
15     if uPoint < uEnd then
16         w = row[vPoint + vOffset]
17         tc += binSearch(w,u,row,col)
18     end
19     vOffset += blockSize
20 end
21 reduce tc in a warp
```

In Hu's algorithm, each thread traverses the 2-hop neighbors with a fixed stride, ensuring that the entire set of 2-hop neighbors are evenly distributed to threads. Certainly, as the 2-hop neighbors are stored in multiple lists, it is non-trivial to implement the idea (see Algorithm 1 for details). Additionally, neighboring threads access adjacent data, leading to efficient memory access. Several other optimizations were introduced, including storing frequently accessed variables in constant memory, using loop expansion to reduce in warp, and determining appropriate block and shared memory sizes, resulting in further performance improvements.

## G. H-INDEX

H-INDEX [15] is an edge-centric triangle counting algorithm, which uses a thread block or a warp to process an edge. Similar to TriCore, H-INDEX performs triangle counting via searching against a (fixed-size) hash table. The hash table contains two parts: len array and element array, $len(i)$ represents the number of elements in the i-th hash bucket, while $element(i)$ represents all elements in the i-th hash bucket (Figure 9). For each edge $(u, v)$, the algorithm runs in two steps. 1) *Hash table construction*: A hash table is built in a parallel manner to store all the neighbors of vertex $u$. 2) *Search*: a block or a warp of threads will access the neighbor list of vertex $v$ in the edge, and check whether each neighbor of $v$ exists in the hash table.

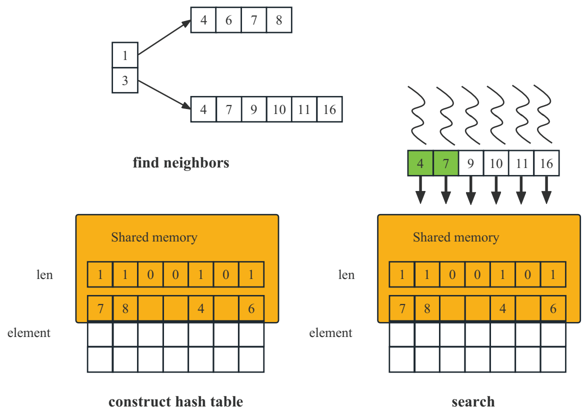


Fig. 9. Overview of the H-INDEX algorithm.

A key problem that H-INDEX tackled is to reduce collision in the hash table. For that, H-INDEX uses the shorter neighbor list to construct the hash table and uses members of the longer list as queries. To achieve faster search, H-INDEX stores the first few elements in each bucket in shared memory. Furthermore, the hash table is stored in a "row-order" manner, i.e., the $i$-th element of all buckets is stored in continuous memory address. This increases the chance of coalesced memory access.

## H. TRUST

TRUST [16] is a vertex-centric triangle counting algorithm that combines the strengths of Hu's algorithm and the H-INDEX algorithm. It uses a fine-grained approach similar to Hu's algorithm in using all 2-hop neighbors as queries to find matches in the 1-hop list. On the other hand, it constructs a hash table to store the 1-hop neighbor list and performs hash lookups similar to H-INDEX (Figure 10).

TRUST uses a heuristic strategy to resolve the computation workload imbalance between vertices. Specifically, vertices

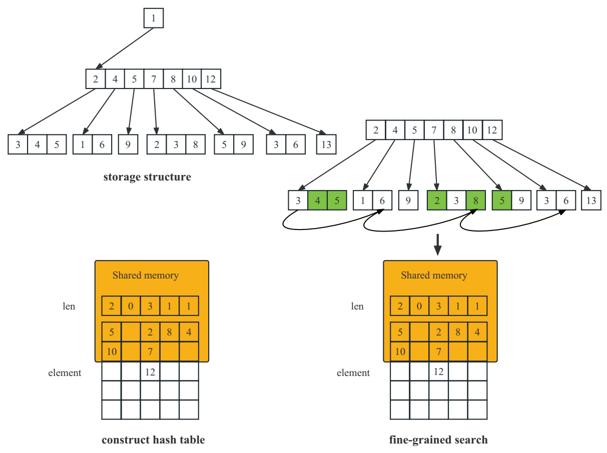


Fig. 10. Overview of the TRUST algorithm.

TABLE II
SIZES OF DATASETS EXTRACTED FROM THE SNAP WEBSITE.

| dataset | vertices | edges | avg degree |
|---|---|---|---|
| As-Caida | 16K | 43K | 5.2 |
| P2p-Gnutella31 | 33K | 119K | 7 |
| Email-EuAll | 39K | 151K | 7.7 |
| Soc-Slashdot0922 | 53K | 475K | 17.7 |
| Web-NotreDame | 163K | 928K | 11.3 |
| Com-Dblp | 273K | 1M | 7.3 |
| Amazon0601 | 391K | 2.4M | 12.4 |
| RoadNet-CA | 1.6M | 2.4M | 2.9 |
| Wiki-Talk | 626K | 2.8M | 9.2 |
| Web-BerkStan | 645K | 6.6M | 20.4 |
| As-Skitter | 1.4M | 10.8M | 14.7 |
| Cit-Patents | 3.1M | 15.8M | 10.2 |
| Soc-Pokec | 1.4M | 22.1M | 30.1 |
| Sx-Stackoverflow | 1.9M | 27.5M | 28 |
| Com-Lj | 3.2M | 33.8M | 21.1 |
| Soc-LiveJ | 3.7M | 41.7M | 22 |
| Com-Orkut | 3M | 117M | 77.9 |
| Twitter | 39M | 1.2B | 60.4 |
| Com-Friendster | 51M | 1.8B | 69 |

with different out-degrees are associated with different number of thread resources. For vertices with an out-degree higher than 100, TRUST uses a block of a fixed size of 1024 threads, and allocates a hash table with 1024 buckets; for vertices with an out-degree between 2 and 100, TRUST uses a warp of 32 threads, and allocates a hash table with 32 buckets. Note that vertices with out-degree lower than 2 are ignored.

## IV. Evaluation

We developed a comprehensive testing framework that allows us to evaluate all eight aforementioned implementations. The source code for these implementations was retrieved from Github repositories provided by the authors of the corresponding publications. Additionally, we have developed data transformation tools that preprocess data into different formats required by the algorithms. such formats include text edge lists, binary edge lists, CSRs, etc. This ensures a convenient feeding of datasets from different sources to different ITC implementations. The complete code for the framework is available on our GitHub site[1].

**Platform:** All performance tests were conducted on a workstation with an Intel(R) Xeon(R) Gold 6126 CPU, 256GB of DDR4 RAM, an NVIDIA Tesla V100, and an RTX 4090 Founder's Edition GPU card. The Tesla V100, with a Volta GPU engine, has 16 GB of memory and 80 multi-processors, each processor contains 64 CUDA cores and a maximum of 48 KB of shared memory. The RTX 4090, with an Ada engine, has 24 GB of memory and 144 multi-processors, each processor contains 128 CUDA cores and a maximum of 128 KB of shared memory. Our code is CMAKE-based and was compiled using GCC 7.3.1 and NVIDIA CUDA 11.3 with O3-level optimizations enabled.

**Datasets:** As shown in Table II, multiple real datasets are obtained from SNAP [20] and cover various types of graphs including social networks, citation graphs, web graphs, and communication networks. To facilitate algorithm testing, we conduct basic data cleaning operations. These include removing vertices that are not connected to any edges, eliminating self-loop edges, and resolving duplicate edges within the graph. It is important to note that these transformations do not alter the number of triangles within the graph.

**Program configuration:** We test each program under different sets of parameters, and report the ones that yield the best performance. In particular, for Polak, Hu, TriCore and TRUST, we report results under their default configurations. For Green implementation, we set gridSize to one-tenth the total number of edges in the graph, blockSize to 512, and use 32 threads for each intersection - this configuration has been found to be optimal in most cases. For Bisson, we use the graph compaction operation mentioned in [4], which leads to higher performance. For the Fox implementation, as described in [5], the intersection method based on Bin-Search is faster than the implementation based on Merge in most cases, so we only report results for adapting Bin-Search. For the H-INDEX implementation, using the block-based configuration produced incorrect results, so we only use the warp-based configuration.

**Metrics:** We evaluated all implementations using GPU kernel running time as the primary metric. Additionally, we employed the CUDA profiling tool nvprof [14] to analyze various performance metrics, including number of memory accesses, warp execution efficiency, and memory access efficiency. The specific metrics observed are as follows:

1) **global_load_requests** refers to the total number of global memory load requests from the multiprocessors.
2) **warp_execution_efficiency** refers to the ratio of the average active threads per warp to the maximum number of threads per warp. A higher warp execution efficiency indicates a more balanced workload within a warp.
3) **gld_transactions_per_request** refers to the average number of global memory load transactions performed for each global memory load request. A lower value

[1]https://github.com/good-ncu/TC-Compare

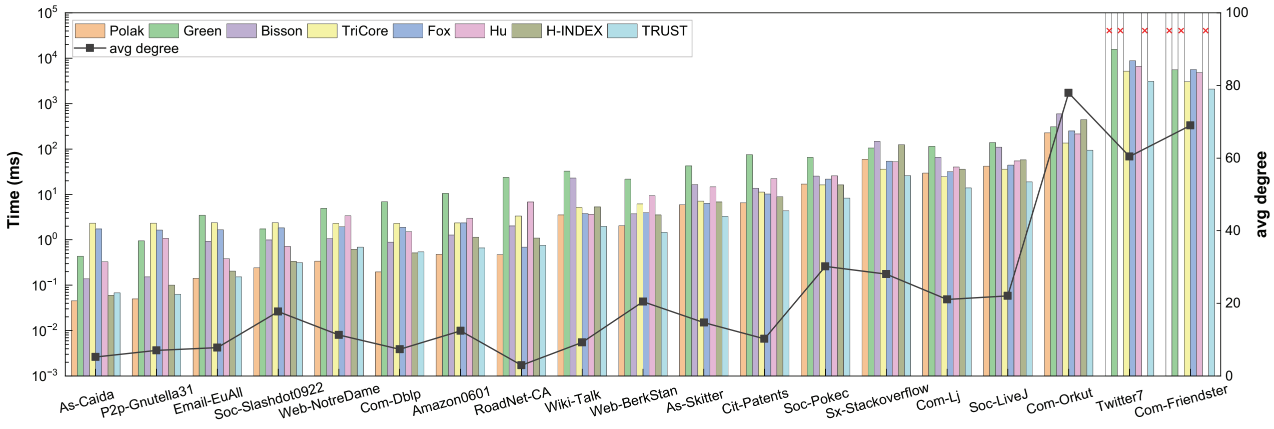


Fig. 11. Total running time of different ITC code under different datasets as well as average vertex degree of the datasets. The datasets are presented in the order of increasing graph size (edge number). Cases marked with a red cross represent events in which the code failed to run.

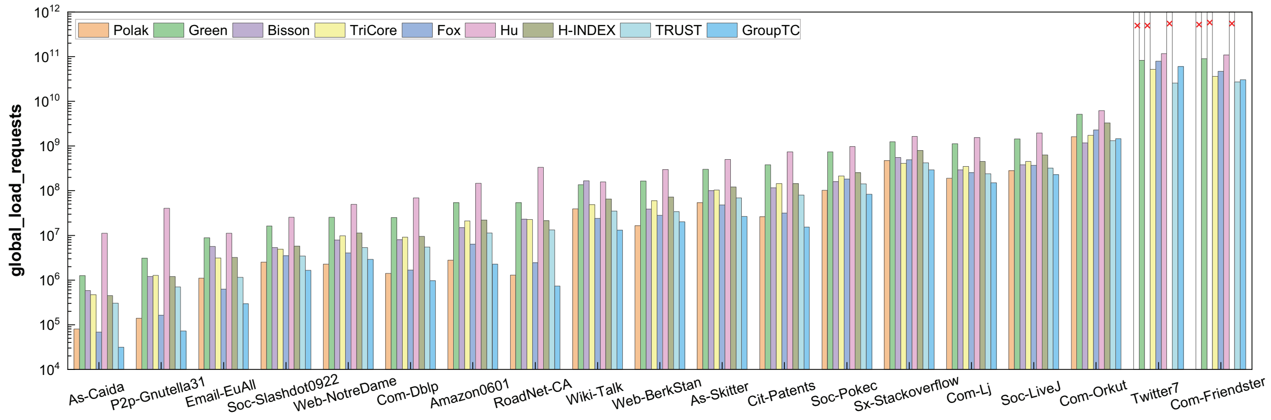


Fig. 12. Global load requests of different ITC code under different datasets. The datasets are presented in the order of increasing graph size (edge number). Cases marked with a red cross represent events in which the code failed to run.

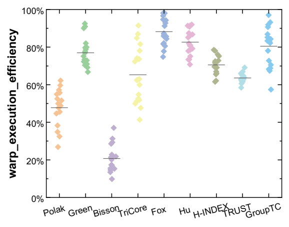
indicates a higher memory efficiency.

## A. Experimental Results and Analysis

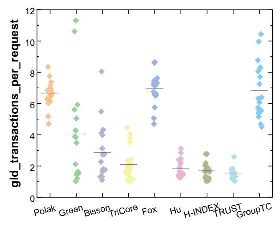
Figure 11 presents the computation time for each implementation in processing all 19 datasets on the V100.[2] A notable trend of the datasets is the average degree seems to increase with the graph size.

Bisson and Green exhibit the worst performance across board, followed by Fox. H-INDEX performed well on small low-degree datasets but show poor performance or even failure on large high-degree datasets. Hu's program performed at an average level among all the implementations. TriCore performs well on large high-degree datasets, but poorly on small low-degree datasets. TRUST shows the best performance in all large datasets starting from the 2.8M-edge 'Wiki-Talk'. The most surprisingly outcome comes from Polak - it shows good performance in all datasets and more notably, is the winner in processing all small-to-medium datasets! With a sophisticated design, the excellent performance of TRUST is understandable. Polak, on the other hand, carries little optimization strategies yet dominated almost half of the datasets.

[2]Results on the RTX4090 are almost the same as in the V100. Here we only present V100 results due to page constraints and the fact that nvprof is not supported in Ada cards.



(a) warp_execution_efficiency



(b) gld_transactions_per_request

Fig. 13. 13(a) is the warp_execution_efficiency of each algorithm under different datasets. 13(b) is the gld_transactions_per_request for each algorithm under different datasets.

Figures 12, 13(a) and 13(b) show profiling results on all implementations and help derive an explanation for each implementation's level of performance.

Polak's warp execution efficiency and memory access efficiency are below average, but its simple design requires much fewer memory accesses than the other methods and therefore excels in small-scale datasets. The Green algorithm is suitable for processing edges that have a significant number of neighbors. However, in real graphs, there are often numerous edges with a small number of neighbors. Therefore, Green's design adds significant overhead and the increased parallelism is not enough to offset this overhead. Bisson, similar to Polak, uses one thread to process the computation around one edge. It exhibits below-average warp execution efficiency and memory access efficiency. Furthermore, the construction of its bitmap adds more synchronization overhead compared to Polak.

TriCore exhibits slower performance on small-scale low-degree datasets due to the requirement of building binary trees. However, it performs well on large-scale high-degree datasets because the tree construction cost can be compensated by numerous efficient lookup operations on the tree. Fox uses the workload estimation method to achieve higher warp execution efficiency and lower memory access times. However, since threads in the same warp are mapped to non-adjacent edges for computing, Fox's memory access efficiency is very low. Hu's fine-grained approach enables high warp execution efficiency and memory access efficiency. However, unlike TriCore, Hu's binary search based on the vertex algorithm cannot benefit from optimization techniques such as flipping the search table and the search key. Consequently, Hu experiences the highest number of memory accesses.

H-INDEX and TRUST utilize hash tables on fast shared memory and employ a fine-grained approach to calculations, distributing the workload evenly among threads within the warp. Consequently, both TRUST and H-INDEX show very high warp execution efficiency and memory access efficiency. TRUST uses a heuristic strategy to allocate threads to process a vertex. However, the implementation of H-INDEX uses a warp to calculate an edge and the number of hash buckets is only 32, which causes H-INDEX to generate too many hash collisions at large-scale high-degree datasets and therefore degrades its performance.

## V. New Algorithm Design

The experimental evaluation has highlighted two standout performers: the Polak algorithm excels when dealing with small low-degree graphs, while the TRUST algorithm shines when confronted with large high-degree graphs.

Polak's good performance for small graphs is largely achieved by its simplicity with fewer memory accesses. However, on large graphs, it suffers with low warp execution efficiency due to the unbalanced workload among threads in a warp. Such divergence is insignificant on small-scale low-degree graphs. As the graph scale and average degree increase, Polak's thread-level work imbalance becomes more and more serious and it quickly loses advantages.

More advanced implementations such as Fox, Hu's, and TRUST implemented different strategies to balance the workload of threads. A key improvement that they all introduced over Polak is using multiple threads to process one edge. Among them, TRUST is the most successful, as seen by its leading performance in all medium to large datasets we tested. TRUST achieved this by using a fine-grained approach that leads to a balanced workload and a careful design of the hash table. However, TRUST does not perform well in small datasets because it uses an entire block to process a vertex. For small graphs, the number of 2-hop neighbors of a vertex can be very small, sometimes even less than the warp size of 32. Furthermore, the overhead in building the hash table becomes more significant in smaller datasets.

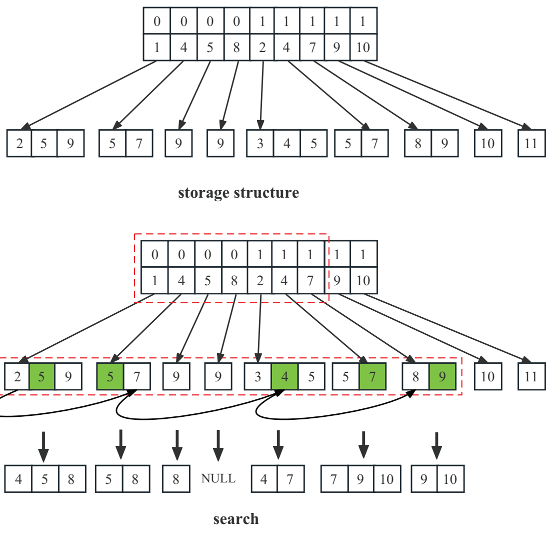


Fig. 14. Overview of the GroupTC algorithm we propose. The red boxes represent the data processed by a block of threads.

We propose an algorithm named GroupTC to address the weaknesses of TRUST and Polak. GroupTC is edge-centric and uses binary search to compute the intersection between neighbor lists. Unlike all existing methods that treat processing one vertex/edge as the basic computational unit, GroupTC considers the processing of multiple edges (known as edge chunks) as the basic unit. The idea is that each edge chunk demands computational work big enough to use up all the thread resources, especially under small low-degree graphs. This guarantees that each thread has a comparable workload.

Specifically, GroupTC (Figure 14) utilizes a block of $n$ threads to compute $n$ consecutive edges. For each edge $(u, v)$, we treat the neighbor list of $u$ as the search table, and the neighbors of $v$ as search keys. We cache the starting point and length of the neighbor lists of $u$ and $v$ in shared memory. Then, each thread accesses the neighbors of $v$ by iterating over the neighbor lists with a fixed stride and searching for them in the neighbor list of $u$. Note that this means neighboring threads always access neighboring members of the 1-hop list and likely so on the 2-hop list, ensuring coalesced memory access. GroupTC utilizes the binary search approach instead

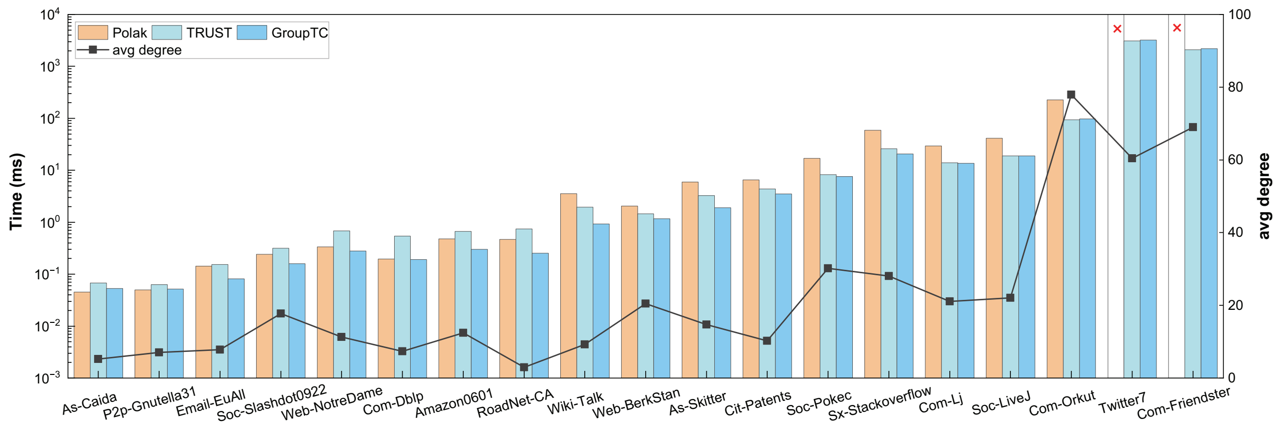


Fig. 15. Running time of GroupTC in comparison to Polak and TRUST under different datasets on the V100 GPU.

of hashing because constructing a hash table for multiple edges means that there may be many more distinct values in the neighbor lists. This situation will require a larger hash table and a careful design of the hash function. Therefore, we choose the binary search method so that our implementation has consistent performance on any type of graph.

Furthermore, we incorporated three optimization techniques to improve performance.

- We assume the data follows a popular format in that, for any edge $(u, v)$, we have $u < v$. Based on this, we only need to search against a partial 2-hop neighbor list. Take the processing of edge $(0, 4)$ in Figure 14 as an example: when searching for the neighbors of vertex 4, we only need to consider the neighbors of 0 with IDs beyond 4 ($[5, 8]$). Similarly, for the edge $(0, 8)$, no search is required.
- The threads access the neighbor list of vertex $v$ in edge $(u, v)$ with a stride. For vertices $v$ with a large number of neighbors, a thread needs to work on multiple such neighbors. In this case, we record the offset position of the last accessed element in the neighbor list of $u$ and start the next search from this offset value to save computations.
- For an edge $(u, v)$, we face two choices for picking either $u$ or $v$ to build the search table. On one hand, choosing the vertex with a smaller neighbor list is a common optimization choice. On the other hand, we can choose a vertex that also exists in other edges to increase the cache hits on the search table. In cases where $u$ both has a smaller neighbor list and exists in other edges, $u$ is the obvious better choice for building the search table. In cases where these two techniques disagree, such as when $u$ exists in other edges but $v$ has a smaller neighbor list, we pick $u$ if the length of the neighbor list of $u$ is less than twice the length of the neighbor list of $v$. This resolution is based on empirical evidence.

GroupTC inherits the advantages of TRUST, where each thread has a balanced workload and data is accessed in a manner that optimizes memory access. However, not having to optimize a hash table means that GroupTC can perform consistently well on different types of datasets.

**Evaluation:** We conducted experiments to compare the performance of GroupTC with Polak and TRUST. According to Figure 15, GroupTC exhibits excellent performance across the board. GroupTC outperforms Polak in 17 out of the 19 datasets, achieving a speedup of 1.03-3.83X, and performs lower than Polak in the two smallest datasets, with speedup of 0.85X and 0.96X. Compared with TRUST, GroupTC outperforms it in medium-sized datasets, with a speedup of 1.09-2.92X. For large datasets, GroupTC exhibits comparable performance to TRUST, achieving a speedup of 0.94-1.01X.

In the profiling metrics, GroupTC's warp execution efficiency is very high, and global load requests are very low. Despite the gld_transactions_per_request being high due to warp processing more edges and side effects of edge flipping optimization. But overall, GroupTC achieves very high warp execution efficiency and requires the lowest number of memory accesses among small-scale low-degree graphs. In large-scale high-average degree graphs, since the time complexity of the binary search is higher than that of the hash table search, GroupTC requires more memory accesses than TRUST.

## VI. Conclusions and Future Work

To solve the triangle counting problem, numerous ideas and algorithms have been proposed. They largely follow three approaches, namely intersection-based, matrix multiplication-based, and subgraph matching-based. A previous comparative study [22] has shown that the intersection-based implementations are more effective than those that follow the other two approaches because they excel at reducing unnecessary computation. In this study, we take a deep investigation into the GPU implementations that follow the intersection-based approach based on a comprehensive framework with 19 real graph datasets. The empirical results show that the TRUST algorithm demonstrates superior performance when processing medium to large-scale datasets whereas the Polak algorithm is the best performer when dealing with smaller datasets. Our algorithm analysis and profiling metrics reveal that TRUST excels in workload balancing and efficient memory usage, which are the factors contributing to its outstanding performance on large-scale datasets. In contrast, Polak's simple design is the key to its efficiency when handling smaller datasets.

Building upon the insights we gained from our analysis, we introduce a novel intersection-based algorithm named GroupTC with high performance across a diverse range of datasets. Remarkably, GroupTC consistently outperforms Polak in all datasets, achieving a speedup of up to 3.83X. Compared to TRUST, GroupTC exhibits significant improvements when processing small to medium-sized datasets, achieving a speedup of up to 2.92X. For large datasets, GroupTC still performs at a comparable level, with a speedup of 0.94-1.01X. These achievements underline the versatility and effectiveness of the GroupTC algorithm in addressing the challenges posed by different data characteristics.

To further enhance GroupTC's efficiency when dealing with large datasets, we will explore alternative methods for optimizing the computation of neighbor list intersections. The primary factor contributing to GroupTC's slightly slower performance on large datasets compared to TRUST is the slower search time of the binary search when compared to a hash table lookup. In our upcoming research, we will focus on developing an algorithm specifically designed to address this bottleneck.

## VII. Acknowledgment

We would like to acknowledge the support from the National Key R&D Program of China (2022YFB4501703), the National Natural Science Foundation of China (KY0402022036), the Provincial Key R&D Program of Jiangxi (012031379055). Y. Tu and M. Pham are supported by an award (1R01GM140316) from the National Institutes of Health (NIH), USA. We also thank the help from the hieco organization and hard work from our anonymous reviewers.

## References

[1] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. "Fast parallel algorithms for counting and listing triangles in big graphs". In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14.1 (2019), pp. 1–34.

[2] Ariful Azad, Aydin Buluç, and John Gilbert. "Parallel triangle counting and enumeration using matrix algebra". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 804–811.

[3] Mauro Bisson and Massimiliano Fatica. "High performance exact triangle counting on gpus". In: *IEEE Transactions on Parallel and Distributed Systems* 28.12 (2017), pp. 3501–3510.

[4] Mauro Bisson and Massimiliano Fatica. "Update on static graph challenge on gpu". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–8.

[5] James Fox et al. "Fast and adaptive list intersections on the gpu". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–7.

[6] O. Green, P. Yalamanchili, and L. M. Munguia. "Fast Triangle Counting on the GPU". In: *Workshop on Irregular Applications: Architectures & Algorithms*. 2014.

[7] Oded Green, Robert McColl, and David A Bader. "GPU merge path: a GPU merging algorithm". In: *Proceedings of the 26th ACM international conference on Supercomputing*. 2012, pp. 331–340.

[8] Chuangyi Gui et al. "Fast triangle counting on GPU". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–7.

[9] Lin Hu, Naiqing Guan, and Lei Zou. "Triangle counting on GPU using fine-grained task distribution". In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2019, pp. 225–232.

[10] Lin Hu, Lei Zou, and Yu Liu. "Accelerating triangle counting on GPU". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 736–748.

[11] Yang Hu, Hang Liu, and H Howie Huang. "Tricore: Parallel triangle counting on gpus". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 171–182.

[12] Farzad Khorasani et al. "CuSha: vertex-centric graph processing on GPUs". In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 2014, pp. 239–252.

[13] Matthieu Latapy. "Main-memory triangle computations for very large (sparse (power-law)) graphs". In: *Theoretical computer science* 407.1-3 (2008), pp. 458–473.

[14] *nvprof*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html\#remote-profiling-with-nvprof.

[15] Santosh Pandey et al. "H-index: Hash-indexing for parallel triangle counting on GPUs". In: *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE. 2019, pp. 1–7.

[16] Santosh Pandey et al. "Trust: Triangle counting reloaded on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 32.11 (2021), pp. 2646–2660.

[17] Adam Polak. "Counting triangles in large graphs on GPU". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pp. 740–746.

[18] Thomas Schank and Dorothea Wagner. "Finding, counting and listing all triangles in large graphs, an experimental study". In: *Experimental and Efficient Algorithms: 4th International Workshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005. Proceedings 4*. Springer. 2005, pp. 606–609.

[19] Xuanhua Shi et al. "Graph processing on GPUs: A survey". In: *ACM Computing Surveys (CSUR)* 50.6 (2018), pp. 1–35.

[20] *SNAP*. https://snap.stanford.edu/data/.

[21] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. “Fast subgraph matching on large graphs using graphics processors”. In: *Database Systems for Advanced Applications: 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part I 20*. Springer. 2015, pp. 299–315.

[22] Leyuan Wang et al. “A comparative study on exact triangle counting algorithms on the gpu”. In: *Proceedings of the ACM Workshop on High Performance Graph Processing*. 2016, pp. 1–8.

[23] Carl Yang, Aydın Buluç, and John D Owens. “GraphBLAST: A high-performance linear algebra-based graph framework on the GPU”. In: *ACM Transactions on Mathematical Software (TOMS)* 48.1 (2022), pp. 1–51.

[24] Abdurrahman Yaşar et al. “Fast triangle counting using cilk”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–7.

[25] Abdurrahman Yaşar et al. “Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments:(update on static graph challenge)”. In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–4.