Lock-Free Triangle Counting on GPU

Zhigao Zheng, *Member, IEEE*, Guojia Wan, Jiawei Jiang, *Member, IEEE*, Chuang Hu, *Member, IEEE*, Hao Liu, Shahid Mumtaz, *Senior Member, IEEE*, and Bo Du

***Abstract*—Finding the triangles of large scale graphs is a fundamental graph mining task in many applications, such as motif detection, microscopic evolution, and link prediction. The recent works on triangle counting can be classified into merge-based or binary search-based paradigms. The merge-based triangle counting paradigm locates the triangles using the set intersection operation, which suffers from the random memory access problem. The binary search-based triangle counting paradigm sets the neighbors of the source vertex of an edge as the lookup array and searches the neighbors of the destination vertex. There are lots of expensive lock operations needed in the binary search-based paradigm, which leads to low thread efficiency. In this paper, we aim to improve the triangle counting efficiency on GPU by designing a lock-free policy named Skiff to implement a hash-based triangle counting algorithm. In Skiff, we first design a hash trie data layout to meet the coalesced memory access model and then propose a lock-free policy to reduce the conflicts of the hash trie. In addition, we use a level array to manage the index of the hash trie to make sure the nodes of the hash trie can be quickly located. Furthermore, we implement a CTA thread organization model to reduce the load imbalance of the real-world graphs. We conducted extensive experiments on NVIDIA GPUs to show the performance of Skiff. The results show that Skiff can achieve a good system performance improvement than the state-of-the-art (SOTA) works.**

Received 23 February 2024; revised 6 August 2024; accepted 16 October 2024. Date of publication 21 November 2024; date of current version 12 February 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFC2705700, in part by the National Natural Science Foundation of China under Grant 62225113 and Grant 62372333, in part by the Fundamental Research Funds for the Central Universities under Grant 2042023kf0135, in part by the Key Research and Development Program of Hubei Province under Grant 2024BAB044 and Grant 2023BAB078, in part by the Project funded by China Postdoctoral Science Foundation under Grant 2022M722459, and in part by the Knowledge Innovation Program of Wuhan - Basic Research under Grant 2023010201010063. Recommended for acceptance by J. Zhai. *(Corresponding author: Bo Du.)*

Zhigao Zheng, Guojia Wan, Jiawei Jiang, Chuang Hu, and Bo Du are with the National Engineering Research Center for Multimedia Software/Institute of Artificial Intelligence/Hubei Key Laboratory of Multimedia and Network Communication Engineering/School of Computer Science, Wuhan University, Wuhan 430072, China, and also with Hubei Luojia Laboratory, Wuhan 430072, China (e-mail: zhengzhigao@whu.edu.cn; guojiawan@whu.edu.cn; jiawei.jiang@whu.edu.cn; handc@whu.edu.cn; dubo@whu.edu.cn).

Hao Liu is with the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China, and also with Wuhan Digital Engineering Research Institute, Wuhan 430000, China (e-mail: liuhao2020@sjtu.edu.cn).

Shahid Mumtaz is with the Department of Engineering, Nottingham Trent University, NG1 4FQ Nottingham, U.K., and also with the Department of Applied Informatics, Silesian University of Technology, 44-100 Gliwice, Poland (e-mail: shahid.mumtaz@ntu.ac.uk).

Digital Object Identifier 10.1109/TC.2024.3504295

***Index Terms*—Triangle counting, lock free, thread organization, GPGPU, parallelism.**

I. Introduction

A triangle consists of three interrelated vertices of a graph, and we can express the triangle as $\Delta_{s,v,t}$ if we use $G=(V,E)$ to express a graph, where V and E are the vertex set and the edge set, $\{s,v,t\}\in V$ and $\{(s,v),(s,t),(v,t)\}\in E$. Triangle counting (TC) is one of the fundamental graph mining tasks that is widely used in social network analysis and pattern recognition applications, and it is also considered as a particular case of counting short cycles or small cliques. A triangle is the smallest subgraph, some scientists simply use the number of triangles in a graph to measure its density and closeness since there are more edges in a dense graph than in a sparse graph. Counting triangles in a graph is an intermediary operation for different applications, such as clustering coefficient computing [1] and community detection [2]. However, different applications have their own requirements for how to express the triangles. For example, the triangle list is needed in the community detection application, while only the number of triangles in a graph is enough for the cluster coefficient computing application.

Due to the increasing size of the real word graphs, researchers have investigated some significant methods to improve the efficiency of parallel algorithms for computing the exact and approximate number of triangles. Most of the current state-of-the-art works focus on processing large-scale graphs and utilizing shared-memory platforms to enhance algorithm performance [3]. Previous studies classified the triangle counting algorithms into four categories, such as set intersection, matrix multiplication, edge sampling [4], and the approximation methods [5]. These efforts have achieved high performance on shared-memory and distributed platforms of conventional CPU architectures, but the algorithm performance still can not meet the applications' requirements due to the large scale of graph size [6].

Graphic Processing Unit (GPU) was first used to accelerate graphic rendering applications, and now many researchers are trying to use it to accelerate high performance computing applications due to massive parallelism and extremely high memory bandwidth [7]. There are also significant researches focused on how to use GPUs to accelerate graph processing algorithms, such as breadth first search (BFS) and PageRank [8]. As an accelerator, GPU follows the SIMT (Single Instruction Multiple Thread) execution model, GPU can only unleash the peak performance when the algorithm meets the coalesced memory access pattern and regular thread organization scheme. Hence, in order to achieve peak performance with a GPU, the algorithm must conform to a coalesced memory access pattern and a regular thread organization scheme. However, a graph consists of vertices and edges, and the vertex degree differs greatly. In a social network graph, if a vertex stands for a single person, the degree of a vertex expresses the number of followers. Then, the degree of a vertex may range from one or tens to millions. This skewed distribution will lead to an exceeding load imbalance problem. Furthermore, the neighbors' vertex ID also ranges, leading to the random memory access problem. In addition, many expensive lock operations are needed to maintain data consistency for the parallel algorithms [9], [10]. Furthermore, the branch divergence operation will slow down GPU performance due to the SIMT model.

Both industry and academic researchers have made many contributions to solve the load imbalance [11] and random memory access problems [12], but lack focus on the expensive lock operations and the branch divergence problem, which greatly limit the GPU performance. In this work, we present a lock-free triangle counting algorithm on GPU called Skiff. Skiff is a custom-designed algorithm that is used to unleash the full potential of GPUs. In Skiff, we design a hopscotch hash [13] based method to release the lock operations on GPU. Moreover, we propose a thread assignment approach to avoid branch divergence operations. The main contributions are listed below.

- We propose a lock-free hash trie mechanism for triangle counting to release the lock operations on GPU. In the proposed lock-free hash mechanism, to minimize the collision risk, we employ a hash function on the shorter neighbor list and use the resulting hash value as the search key for the longer list. Furthermore, we introduce a memory friendly bucket placement strategy (trie) for GPUs to enhance memory utilization.
- We use a linked list manner to organize the hash trie and a level array as an index for the hash trie to speed up the memory access and improve the cache utilization.
- We implement a CTA thread organization technique to avoid the load imbalance problem of real-world graphs. In our CTA thread organization technique, we implement three policies, named Block_Degree, Block_Index, and Warp_Index, to adapt different graphs.
- We conduct a set of experiments to compare the performance of the proposed lock-free triangle counting algorithm with the state-of-the-art algorithms on GPU. The experimental results show that Skiff can achieve a good system performance improvement than the state-of-the-art works.

The remainder of this paper is organized as follows. Section II provides an overview of triangle counting applications. We then discuss the performance issues of the triangle counting algorithm on GPUs and the motivation of this work. Next, Section III presents the algorithm design. Section IV covers the hash trie mechanism for triangle counting and the thread organization technologies for the proposed algorithm. We evaluate Skiff's performance in Section V. Section VI discusses the related works, and Section VII concludes this work and identifies future research opportunities.

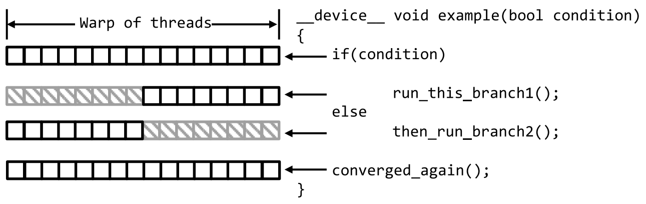


Fig. 1. CUDA warp divergence.

II. Background and Motivation

In this section, we first introduce the preliminary background of this work, which includes the GPU architecture, the merge-based intersection, and binary-search-based triangle counting. In addition, we will discuss the main challenges of designing the triangle counting algorithm on GPU, which motivated us to design the lock-free triangle counting algorithm.

A. Graphics Processing Unit

Thanks to the massive degree of parallelism and high memory access bandwidth of modern GPU, it was widely used to accelerate kinds of graph processing applications, such as graph coloring [14], [15], and triangle counting [16].

GPU architecture is extended by the streaming multiprocessor (SM), we can realize the hardware parallelism of GPU by duplicating the SM structure. There are several SMs included in a GPU, and there are tens to thousands of streaming processors (also called CUDA core, we use CUDA core in this paper) that are included in an SM. In the CUDA programming model, a thread is executed on a CUDA core, a warp is executed on an SM, and 32 consecutive threads make up a *warp*. Several consecutive warps are further combined as a *block* (also called a Cooperative Thread Array, CTA, in some works), and all the blocks together are called a *Grid*. Threads within a warp are executed in a SIMD manner, whereby they execute the same instruction and access contiguous memory space during a time slice. However, when branching occurs within a warp, the threads must wait for other threads to complete the branch, which greatly reduces thread utilization. This phenomenon is the famous *branch divergence* problem. Fig. 1 illustrates the warp divergence on the GPU. From this example, we can conclude that if there are N branches, the warp efficiency is no more than $1/N$.

In the hierarchical memory model of a GPU, shared memory is exclusive to the SM, allowing threads within a block to communicate with each other through shared memory. Global memory, on the other hand, can be accessed by all threads in the grid. Given the limited memory space on a GPU compared to a CPU, it is crucial to efficiently use both local and shared memory. However, achieving full memory bandwidth on a GPU requires *coalesced memory access*, while *strided memory access* and *scattered memory access* will reduce the GPU's

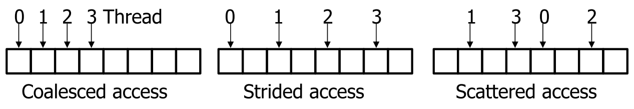


Fig. 2. Different memory accessing fashion.

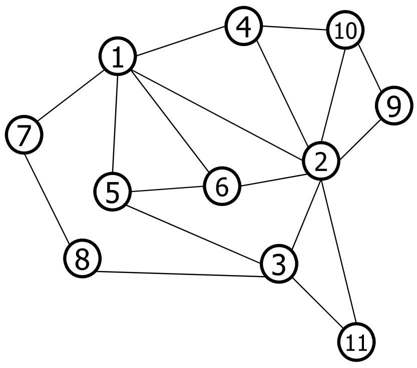


Fig. 3. A sample graph.

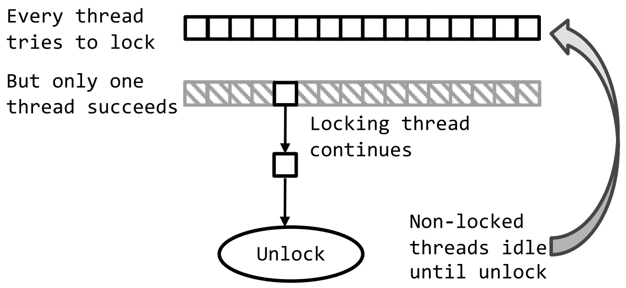

Every thread tries to lock
But only one thread succeeds
Locking thread continues
Unlock
Non-locked threads idle until unlock

Fig. 4. The lock operation on GPU.

ALGORITHM 1: Vertex-centric programming fashion

```
1 G=(V,E);
2 for $u \in V$ do in parallel
3     for $v \in N(u)$ do in parallel
4         $count$+=Intersection($u, v$);
```

ALGORITHM 2: Edge-centric programming fashion

```
1 G=(V,E);
2 for $(u, v) \in E$ do in parallel
3     $count$+=Intersection($u, v$);
```

throughput. With coalesced memory access, threads within a warp access consecutive memory addresses. On the contrary, strided memory access involves threads accessing consecutive memory spaces with a fixed stride, and scattered memory access involves randomly accessing memory spaces. We show the different memory access fashion in Fig. 2.

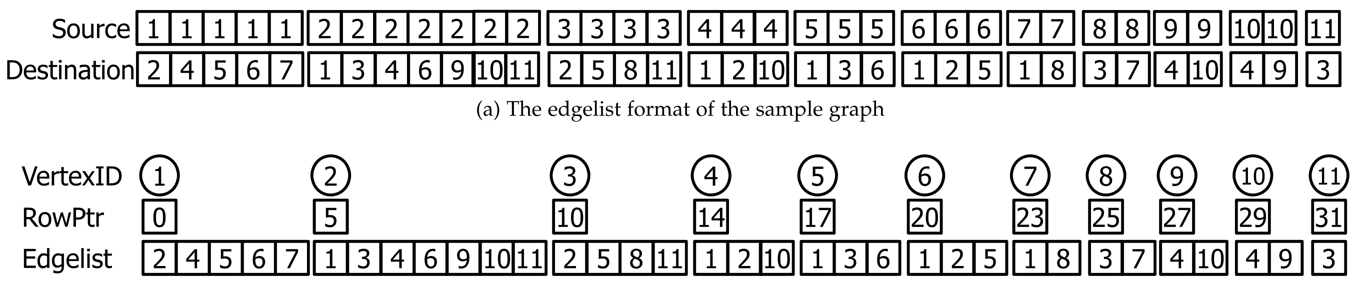
B. Triangle Counting

In a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a triangle is formed by three vertices with one edge connecting each pair of vertices. For instance, in Fig. 3, vertices 1, 2, and 4 form a triangle $\Delta_{1,2,4}$. To count the triangles, we use the *set intersection* operation, which counts the number of shared neighbors between two vertices u and v of an edge $e = (u, v)$. This number is also the number of triangles to which the edge e belongs. Specifically, we represent the set of neighbors of vertex u and v by $N(u)$ and $N(v)$, respectively. If there exists a vertex $w \in N(u) \bigcap N(v)$, then (u, v, w) forms a triangle. In Fig. 3, since vertices 1 and 2 share two neighbors, 4 and 6, there are two triangles ($\Delta_{1,2,4}$ and $\Delta_{1,2,6}$) of edge $(1, 2)$.

Since the algorithm on the GPU is implemented in parallel, multiple threads execute the same instructions during the same time slice. To ensure data consistency, many lock operations are performed. However, due to random data allocation, there are only a few threads that can be locked in each iteration, and they will be unlocked only when the operation on the fetched data is finished, and the non-locked threads are idle until unlocked. Fig. 4 illustrates an extreme example: a warp includes 16 threads, only one thread is active while the other 15 threads are idle until the active thread releases the lock.

Most graph processing algorithms perform a vertex-centric or edge-centric processing model [17], [18], algorithms 1 and 2 show the two programming models, respectively. In the vertex-centric graph processing model, a vertex is assigned to a thread/warp, and the value of a vertex can be passed to its neighbor immediately once the thread/warp to this vertex is finished. However, the workload is imbalanced due to the fact that different vertices have different numbers of degrees on real-world graphs. In social network graphs, a supernode may have more than millions of neighbors, while a normal node may just have tens or even below neighbors. Unlike the vertex-centric processing model, the edge-centric model assigns a thread/warp to an edge but not a vertex. The workload is balanced because each thread/warp processes only one edge for each instruction. The value of a vertex can be passed to its neighbor once all the threads/warps on the edge finished, but there are many more edges than vertices in real-world graphs, which makes the edge-centric processing model is slower than the vertex-centric model in each iteration [19]. In this paper, we prefer the vertex-centric programming model to meet the execution time requirements. We further implemented the CTA thread organization to overcome the load imbalance problem.

We represent the degrees of vertices u and v as $d(u)$ and $d(v)$, respectively. The workload for each edge $e = (u, v)$ is defined as $d(u) + d(v)$, while in the vertex-centric programming model, the workload for vertex v can be expressed as $d^2(v) + \sum_{u_i \in N(v)} d(u_i)$. Intuitively, these equations imply that the degree distribution skewness leads to linear and quadratic workload imbalances [16]. However, the edge-centric mechanism requires the edges list graph presentation fashion to meet the higher parallelism [16], which limits the size of the accommodated graph. Fig. 5(a) and 5(b) show the edgelist and compressed sparse row (CSR) representation of the sample graph


Source 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 5 5 5 6 6 6 7 7 8 8 9 9 10 10 11
Destination 2 4 5 6 7 1 3 4 6 9 10 11 2 5 8 11 1 2 10 1 3 6 1 2 5 1 8 3 7 4 10 4 9 3
VertexID 1 2 3 4 5 6 7 8 9 10 11
RowPtr 0 5 10 14 17 20 23 25 27 29 31
Edgelist 2 4 5 6 7 1 3 4 6 9 10 11 2 5 8 11 1 2 10 1 3 6 1 2 5 1 8 3 7 4 10 4 9 3

(a) The edgelist format of the sample graph

(b) The compressed sparse row (CSR) format of the sample graph

Fig. 5. The edgelist and CSR representation format of the sample graph.

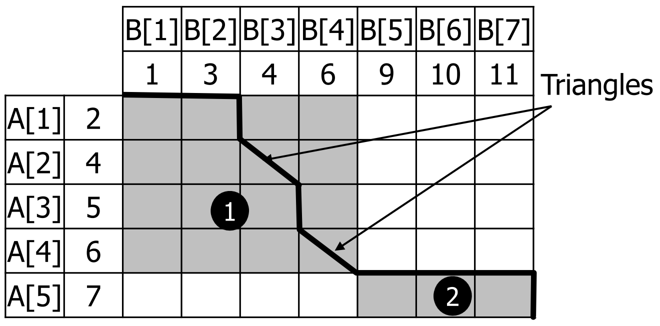


Fig. 6. Merge-based triangle counting of edge $(1, 2)$ of the sample graph, the adjacency list of vertex 1 and 2 are stored in the array A and B, respectively.

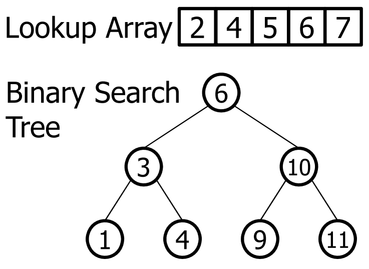


Fig. 7. Binary-search-based triangle counting of edge $(1, 2)$ of the sample graph. In this method, the shorter array is used as the lookup array while the longer array is used to build the binary search tree.

in Fig. 3. The CSR format requires fewer memory spaces and stores graphs compactly with minimal overhead, enabling fast traversals, lookups, and basic graph computations. Previous works have stored multiple formats of the graphs in GPU on-board memory to maintain efficiency, but we prefer a single CSR graph in GPU memory in this paper.

C. Motivations

Most state-of-the-art triangle counting algorithms on GPU follow a merge-based or binary search-based paradigm [10]. The merge-based approach evenly assigns workload to threads, which leads to low thread efficiency. Furthermore, the merge-based method executes the algorithm with an irregular memory access pattern. However, unlike the merge-based method, the binary-search-based approach will suffer from high time complexity when the vertices are not sorted. On the other hand, lots of divergent branches exist in the binary-search-based method which slows down the performance of the algorithm.

Figs. 6 and 7 illustrate the merge-based and binary-search-based intersection approach for edge $(1, 2)$ of the sample graph in Fig. 3, respectively. In the merge-based intersection approach, two arrays A and B are used to store the adjacency list of the two vertices of edge $(1, 2)$. In the intersection operation, two pointers are used to scan through the neighbor list of vertex 1 and 2. A triangle is located once we find a neighbor in both lists, and then the pointers are incremented until all the vertices in the lists are traversed. Overall, we can conclude that the time complexity of the merge-based triangle counting algorithm is $\mathcal{O}(\sum_{v\in V} d^2(v))$ from the above traversal pattern. In Fig. 6, $A[]$ is the neighbor list of vertex 1, and $B[]$ is the neighbor list of vertex 2. Suppose that one thread/warp processes four vertices. Fig. 6 shows that thread/warp ❶ is processing 16 operations and located vertex 4 and 6 in both the two neighbor lists to form two triangles $\Delta_{1,2,4}$ and $\Delta_{1,2,6}$. There are only 3 operations for thread/warp ❷, which will lead to a serious load imbalance problem. Furthermore, thread/warp ❷ is visiting $A[5]$ and $B[5]$ when thread/warp ❶ is visiting $A[1]$ and $B[1]$, this strided memory access fashion will lead to low memory throughput.

In the binary-search-based intersection approach, the longer neighbor list array is used to build the binary search tree, while the shorter one is used as the lookup array. In this example, there are 5 neighbors of vertex 1 and 7 neighbors of vertex 2. So, the vertex 1's neighbor is taken as the lookup array, and the vertex 2's neighbor is organized as the binary search tree. The vertices in the lookup array were compared with the binary search tree in parallel in each iteration. The time complexity of the binary-search-based approach is $\mathcal{O}(|E|^{2/3} log\sqrt{|E|})$, and it is better than the merge-based intersection approach when $m << n$ (for sparse graphs), i.e., the binary-search-based approach is un-benefiting for the non-sparse graphs. Furthermore, in the binary-search-based approach, the neighbor list must be sorted, and the time complexity of sorting can be as high as $\mathcal{O}(N)$ (where $N = |V|$ is the vertex number of graph G), which is costly. Moreover, the binary search will also lead to the strided memory access fashion when the nodes in the tree are sorted in a descending manner. For instance, there are two active threads in runtime in Fig. 7, if thread/warp 2 is visiting

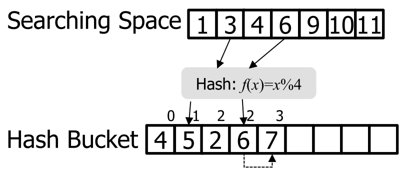


Fig. 8. Hash based approach.

search key 11 when thread/warp 1 is visiting search key 4, this situation will lead to a strided memory access fashion. In addition, there are lots of branch divergence operations, which will reduce thread utilization.

In the hash-based approach, the neighbor list of the active edge's source vertex and destination vertex was listed in the specified array. One of the neighbor lists (usually the shorter one) was used to build the hash bucket, while the other one was chosen as the searching space. In the execution stage, the algorithm searches to see if the entity of the searching space exists in the bucket. So, a linear search is achieved to check whether a similar neighbor from the longer list exists in the bucket. Fig. 8 illustrates the basic hash-based triangle counting scheme. In this example, we use the hash function $f(x) = x\%4$ to build the hash bucket of vertex 2's neighbor list A, while vertex 1's neighbor list B is the searching space. We can find all triangles in parallel with $\mathcal{O}(1)$ time complexity.

Conflict is the performance killer of building the hash bucket. The "read-write" conflict occurs when more than one entity is located in the same hash bucket position, such as entity 2 and 6 of neighbor list A in Fig. 8. In this example, we set the neighborhood H length as 4, so that we can move the conflicted vertex 6 to the first free bucket of 2, and then insert the new hashing vertex 7 to the next free bucket. There is a bit-mask link in vertex 6 to show the real location of vertex 7, which is shown as the dotted arrow in Fig. 8. This method is easy to implement, but the lock is needed for the probing scheme to find a free bucket when the conflict occurs, which will slow down the thread efficiency.

III. Algorithm Design

In this section, we introduce the proposed lock free triangle counting algorithm, *Skiff*, to solve the above problems of existing works.

A. Lock Free Hash-Based Intersection

Previous research [20] shows the pointers to nodes that are close to the leaves of the hash trie would likely boost the key search. In Skiff, we propose a nonblocking hash-trie to meet the cache locality of the GPU.

In the proposed hash trie structure, we use a linked list manner to organize the hash trie, and we use a bucket that can store H vertex instead of the traditional hash trie (which can just store a single vertex). We first use a level array to act as the index of the hash trie, to ensure the algorithm can locate the active nodes quickly. Each item of the array holds a pointer to nodes in one level of the trie. Using this scheme, we can locate

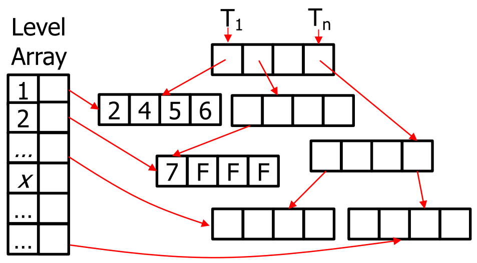


Fig. 9. Hash trie based triangle counting scheme.

the neighbor vertex list of v in $\mathcal{O}(1)$ memory access. Suppose there are N vertices in graph G, then the level array should be $\lceil log_H N \rceil$. We can see that the level array is a compact structure with limited space. By using the level array to index the hash trie, we can only load the trie but not all the hash tree into the memory quickly, and there is no empty bucket in the hash tree (use a virtual bucket to express the buckets without elements), which can release the pressure of the GPU memory allocator compared to the traditional hashing method. As the same with the traditional hash based triangle counting approach, we first load the neighbor list of the source and destination vertex of the active edge into arrays, and then use the shorter neighbor list to build the hash trie, while the longer one was treated as the searching space.

Fig. 9 illustrates the proposed hash trie based triangle counting scheme. In this example, we use the hash trie structure to store the searching space, which is the longer neighbor list of the source vertex. This example sets the bucket as 4, vertex 1's neighbor list {2, 4, 5, 6, 7} has 5 elements. Therefore, the neighbor of vertex 1 needs to be stored in at least $\lceil 5/4 \rceil = 2$ buckets. We can locate the triangles of edge $(1, 2)$ only by comparing vertex 2's neighbor with the two buckets of vertex 1's neighbor in parallel, i.e., searching the 2 buckets of vertex 1 from the first 2 elements of the level array.

Coalesced memory access can reduce memory access time and memory access divergence. Furthermore, organizing the graph vertex compactly can significantly reduce the memory space of the GPU. In the proposed hash trie scheme, we set the bucket size as the same as the GPU warp to avoid memory access divergence. On the other hand, this configuration can also map the memory access to the GPU cacheline, which can improve the GPU memory access efficiency by improving the cache efficiency [21]. Furthermore, we store each bucket element consecutively to coalesce the memory access. Since the GPU threads (warps or CTA) search for the common neighbors begins with the first elements in various buckets. Therefore, storing all the elements of the same bucket contiguously will introduce strided memory access.

B. Hash-Based Triangle Counting

Algorithm 3 shows the procedure of the proposed hash trie based triangle counting.

In the proposed algorithm, we use the shorter neighbor list to build the hash bucket. We use B to express the bucket and set the bucket length as H, *temp* is an integer to mark the position of

ALGORITHM 3: Lock free hash based triangle counting

```
Input: Neighbor list $A$ and $B$
Output: Triangles of edge $(u, v)$
1  ▷ Phase 1: Build the bucket
2  Initialization
3      $bucket \leftarrow \varnothing$;
4  if $|A| > |bucket|$ then
5      malloc(bucket);
6      $bucket \leftarrow \varnothing$;
7  while $a \in A$ do in parallel
8      while $bucket[pos] \neq null$ do
9          pos++;
10     if $pos <= temp + H - 1$ then
11         $bucket[temp].dist \leftarrow bucket[temp].dist + (1 << (H - 1 - pos + temp))$;
12         return;
13     else
14         for $i \leftarrow H - 1 ... 1$ do
15             for $j \leftarrow H - 1 ... H - 1 - i$ do
16                 if $(bucket[pos - i].dist >> j)\%2 == 1$ then
17                     $hash\_item = bucket[pos - i + H - 1 - j]$;
18                     $bucket[pos].value \leftarrow hash\_item.value$;
19                     $hash\_item.dist \leftarrow hash\_item.dist - (1 << j) + 1$;
20                     $pos = pos - i + H - 1 - j$;
21                     if $pos <= temp + H - 1$ then
22                         $bucket[temp].dist \leftarrow bucket[temp].dist + (1 << (H - 1 - pos + temp))$;
23                         return;
24                     else
25                         break;
26             if $bucket.isFull() == true$ then
27                 break;
28         if $bucket.isEmpty() == true$ then
29             break;
30 ▷ Phase 2: Search triangles from bucket
31 for $b \in B$ do in parallel
32     $hash \leftarrow udf\_hash(b)$;
33     for $i \leftarrow 0 ... H - 1$ do
34         $dist \leftarrow buck[hash].dist$;
35         if $(dist >> i)\%2 == 1$ then
36             if $bucket[hash + H - 1 - i].value == b$ then
37                 return $triangle(u, v, b)$;
38     return -1;
```

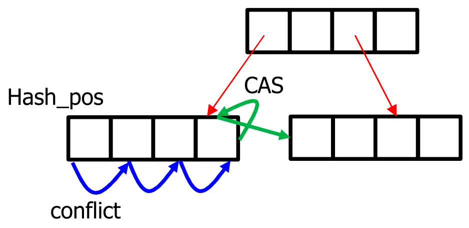


Fig. 10. The optimistic policy of the hash trie.

the vertex in the temporary bucket, and $bucket[temp].dist$ is the length of the temporary bucket. If, unfortunately, the conflict occurs, we will find a blank position in the following H space and insert the conflicted value into the blank position, shown as lines 10–13 of algorithm 3. If the whole B is filled by conflicted/hash values, then we need to move an element between the hash position and the first blank position backward so that the empty space moves closer to the current hash position. Repeat this process until the empty space moves into B, then we can insert and use that space for the conflicted vertex. Details are shown on lines 15–26 of the algorithm 3. By using this way, we can keep key-value pairs close to the original bucket, which will keep the chains short and achieve good memory locality.

We can build the hash bucket in constant time if there are no conflicts since limited vertices are included in a hash bucket, while the worst-case complexity is $\mathcal{O}(n)$, where n is the vertex number of the graph. We can find the triangles in $\mathcal{O}(1)$ time complexity as line 33–38 show in algorithm 3. Hence, the expected time complexity is constant, while the worst-case is $\mathcal{O}(n)$ of the proposed method.

IV. Implementation

In this section, we will introduce our optimization techniques for Skiff, including achieving lock-free operations and improving cache utilization.

A. Lock Free Operations in Skiff

Modern GPUs organize the threads in the SIMT (Single instruction multiple threads) model so that the instructions for all threads in the same warp are executed in a lock-step. In this thread organization model, the lock operation significantly degrades GPU efficiency since the instruction is finished only when the most time-consuming task in a warp is finished. Furthermore, this scheme will also lead to stragglers between warps. In this paper, we implemented the optimistic policy to process the conflict when building the hash trie. The proposed optimistic policy puts the conflict items at the following position of the original item before the hash trie node is full, and both the conflict item and the original item are located in the same node without lock operations. One compare-and-swap (CAS) operation is needed for a hash trie at the end element of the hash trie (the hash trie node is full when the last element is added). Fig. 10 illustrates the optimistic policy of the hash trie, and algorithm 4 shows the CAS operation (the ptr in algorithm 4 is the pointer of the current value).

ALGORITHM 4: CAS operation on GPU

```
1 ▷ Function CAS($ptr, oldvalue, newvalue$)
2 if $*ptr = oldvalue$ then
3     $*ptr \leftarrow newvalue$;
4     return true;
5 else
6     return false;
```

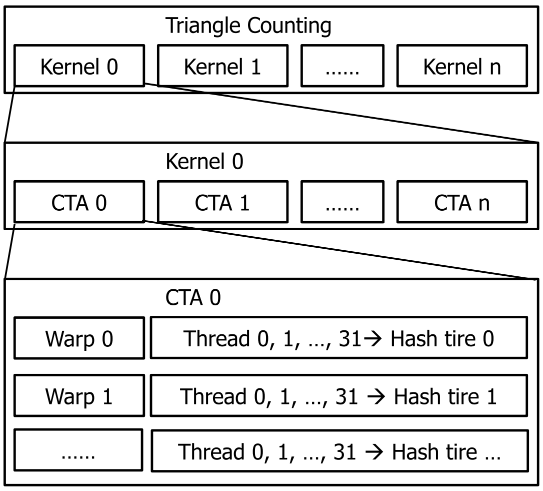
In Fig. 10, the conflicted item was put at the next position of the original item without lock operation (which is shown as the blue line in Fig. 10). A CAS operation is needed if there are too many conflicts until the trie node is full, and we further put the conflicted item into the next node under the protection of the CAS operation (which is shown as the green line in Fig. 10). A new bucket will be allocated once the current bucket is full and more conflicts occur, and a pointer from the current bucket's last item to the new bucket's first item will be used to link the two buckets.

We can avoid most lock operations using the optimistic conflict control policy. In traditional hash trie building operation, there are up to N CAS operations for a graph with N vertices. But in the proposed method, we can avoid most of the CAS operations. There are no more than w items in a trie node if we set the width of the trie as w. We can further prescribe a vertex can be hashed into a new trie node only when the previous trie is full. We can further conclude that at least $\lceil log_w N \rceil$ tries for a given graph G with N vertices are needed. Hence, there are no more than $\lceil log_w N \rceil$ CAS operations when building the hash trie of graph G. Since there is no write operation when counting the triangles, all the conflicts occur only in the hash trie building phase. No more than $\lceil log_w N \rceil$ CAS operations are needed in the proposed algorithm.

B. Thread Organization

In CUDA, all the threads in the same warp run simultaneously in lockstep. Hence, the execution time of a warp depends on the thread assigned to the heaviest task. In the typical warp-based thread assignment with the vertex-centric implementation of triangle counting, the load imbalance problem will occur due to workload divergence, which is led by the irregular degree of the vertices and the skewed data of the hash join operation. In order to solve this problem, this paper implements the CTA thread organization approach to assign vertices to a warp or a block or even a set of blocks according to the vertex degree flexibly rather than assign a vertex to a fixed warp, i.e. if there are 32 threads in a warp, than we assign $\lceil d(v)/32 \rceil$ warp to the vertex with degree $d(v)$, Fig. 11 is an illustration of the CTA thread organization hierarchy.

The GPU cache is much smaller than the CPU system (the cache line of NVIDIA A100 GPU is 128KB, while this data of NVIDIA GeForce RTX 2080 Ti GPU is 64KB), just a few blocks can be loaded into a cache line during the memory access. In our thread organization model, a CTA includes blocks and warps that can flexibly match the GPU cache line at the same time, which can help us improve the utilization of GPU memory.



Triangle Counting
Kernel 0
Kernel 1
......
Kernel n
Kernel 0
CTA 0
CTA 1
......
CTA n
CTA 0
Warp 0
Thread 0, 1, ..., 31→ Hash tire 0
Warp 1
Thread 0, 1, ..., 31 → Hash tire 1
......
Thread 0, 1, ..., 31 → Hash tire ...

Fig. 11. The CTA thread organization hierarchy.

On the other hand, the GPU onboard memory is limited, which is stretched for large-scale real-world graphs. A straightforward method to process large-scale graphs on GPU is to partition them into several small sub-tasks. In this paper, we implement the degree-based workload partition and index-based workload partition, respectively, to partition the large-scale graph, which does not fit the GPU memory. By doing so, we have Block_Degree, Block_Index, and Warp_Index, three kinds of thread assignment fashion in our CTA model. In the Block_Degree (block-based thread organization with degree-based workload partition) model, we assign $\lceil d(v)/32 \rceil$ blocks to the vertex v with degree $d(v)$. In the Block_Index (block-based thread organization with index-based workload partition) model, we assign $\lceil \sum d(v)/32 \rceil$ blocks to the vertices v associated with the target index of the level array. While similar to the Block_Index model, we assign $\lceil \sum d(v)/32 \rceil$ warps to the vertices v associated with the target index of the level array for the Warp_Index (in warp-based thread organization with index-based workload partition) model. In this thread assignment model, we can ensure there is at least one thread to process one edge connected to the active vertex.

C. Improve Cache Utilization

In the SIMT execution model, all the threads executed in parallel with the single instruction multiple data (SIMD) are combined with multithreading. In this data accessing fashion, all the threads in the same warp (or CTA) access the contiguous memory space with the same instructions. In this manner, we can maximally reduce the memory accessing time and improve efficiency only when all the data for a warp are located in the contiguous memory space. To meet this GPU memory accessing fashion requirement, we set the hash trie as a 32-way trie (we can also set the hash trie width according to the threads

TABLE I
GRAPHS USED IN THE EXPERIMENTS

Datasets	**Vertices**	**Edges**	**Avg. Degree**	**Max Degree**	**Diameter**
Amazon	735,322	5,158,012	14.03	3,567	18
Stanford	281,903	2,312,497	16.41	38,626	164
dblp	986,207	6,707,236	13.60	979	23
orkut	3,072,626	117,185,083	76.28	9,683	9
youtube	2,999,999	2,987,627	1.99	28,754	24
roadnet	1,971,282	5,533,214	5.61	12	8,440
wiki	2,394,386	5,021,410	4.19	100,032	11
liveJournal	4,847,571	68,993,773	28.46	22,887	20
RMAT	9,999,993	160,000,000	32.0	3,015	14
twitter	41,652,225	637,759,209	30.62	3,081,112	23
webbase	113,180,896	906,517,783	16.01	816,127	379

number of the CTA in the CTA thread organization manner). So that a node can contain up to 32 items with the same width as the warp, by using this method, we can access a node with a single warp, and we can also load a few nodes into the GPU cache to improve cache utilization.

Data layout is a crucial factor that impacts memory access efficiency, and the most commonly used layouts are the structure of arrays (SoA) and the array of structures (AoS). SoA organizes elements into a single array, storing different elements in contiguous memory spaces. In contrast, AoS stores different elements interleaved, i.e., all elements in a structure are located in contiguous memory spaces. SoA can load homogeneity in continuous memory space and is flexible in locating data for coalesced access, which benefits SIMD instructions. To adhere to the coalesced memory access model, this paper employs the SoA data layout.

V. EVALUATION

We designed a set of experiments to evaluate the performance of Skiff in this section and compared Skiff with the state-of-the-art works on both power-law and random graphs.

A. Dataset and Experimental Setup

In this paper, both directed and undirected graphs are used in our experiments. As in most state-of-the-art graph algorithms, we use (u, v) to represent the undirected edge between vertex u and v, while $< u, v >$ is used to represent the directed edge.

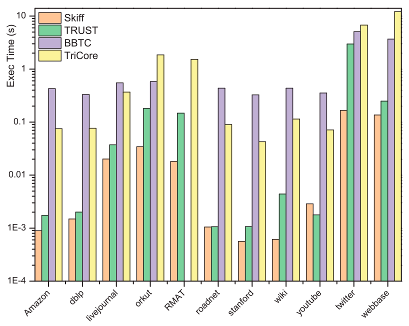
In our experiments, we performed 11 datasets, both from real-world and synthetic graphs. The details of the graph and the attributes are shown in Table I, 10 real-world graphs obtained from Stanford Network Analysis Project (SNAP) [22] and the Laboratory for Web Algorithmics (LAW) [23], [24], [25], while the synthetic datasets, `RMAT` is generated by PaRMAT [26]. Except for `roadnet` and `RMAT`, all the graphs are power-law graphs.

The experiments presented in this paper are conducted on a host with 8 `NVIDIA GeForce RTX 2080 Ti GPU`, which is a Turing architecture-based GPU with 4352 CUDA cores and 11GB GDDR6 on-board memory. The host machine is equipped with 4 Intel(R) Xeon(R) 4210 CPUs, each at 2.2 GHz, and equipped with 376 GB memory. The host machine is running Ubuntu OS version 20.04.1. The algorithm is compiled with g++ 7.5.0 and CUDA Toolkit 11.4, and the optimization flag is set to `-O3` in all experiments.

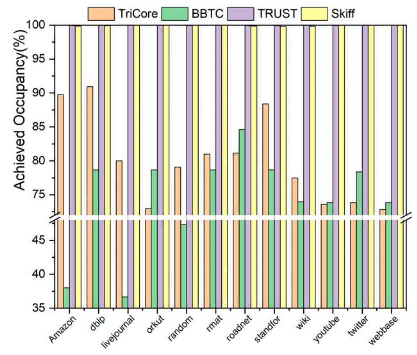
B. Compared State-of-the-Art Works

Most of the recent works are focused on the implementation of the triangle counting algorithm on the CPU, and few of them focus on the optimization techniques on the GPU. In this paper, we compared Skiff with state-of-the-art works, such as TriCore [10], TRUST [27], and BBTC [28]. All these algorithms are implemented on the NVIDIA GPU. We describe some operation details of these methods as follows.

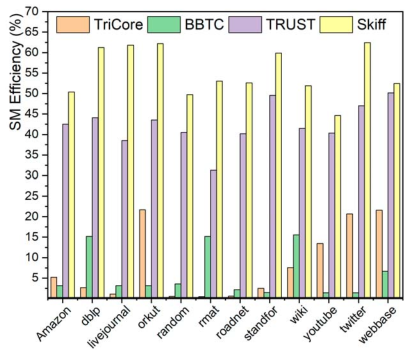
- *TriCore*. TriCore [10] is a TC algorithm on GPU that uses the binary search mechanism. It organizes one neighbor list as a lookup list and the other as a binary search tree. Then, it searches the binary tree to determine if the lookup items exist in the tree in parallel. This method is easy to implement in parallel but suffers from high time complexity when the vertex is unsorted. Furthermore, divergent branches exist when searching the binary tree, which will slow down the algorithm's performance.
- *TRUST*. TRUST [27] is a vertex-centric hashing-based triangle counting algorithm on GPU that avoids collisions by reordering the vertex IDs of a graph. TRUST further proposed an interleaved hash table to layout and a linear search policy to meet the coalesced memory access requirement of the GPU. Though reordering the vertex ID can reduce most collisions, the reordering overhead is also in-negligible.
- *BBTC*. BBTC [28] is a lightweight triangle counting algorithm that aims to process large-scale graphs that do not fit a single GPU device. BBTC implemented the divide-and-conquer approach on GPU to divide the computation into medium-grained tasks. This method can process large-scale graphs on GPU, but the load imbalance problem led by the irregular graphs becomes the noticeable drawback of BBTC.



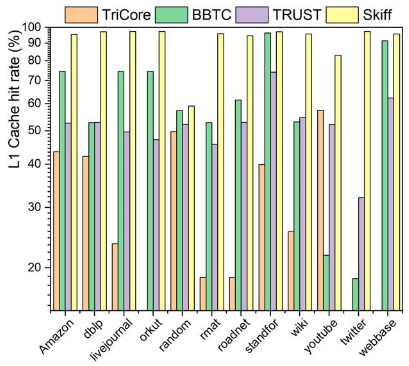
(a) The execution time of the state-of-the-art works



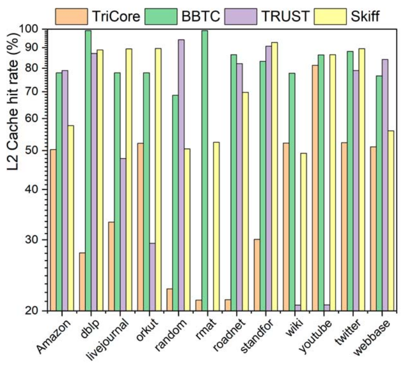
(b) The achieved occupancy of Skiff VS. the state-of-the-art works



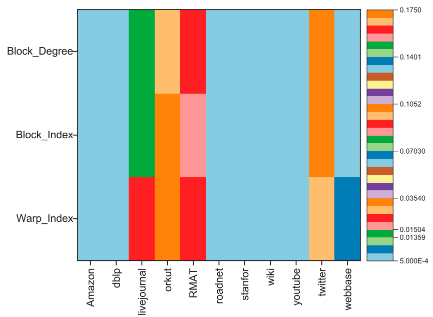
(c) The SM efficiency of Skiff VS. the state-of-the-art works



(d) The L1 cache hit rate of the SOTA works



(e) The L2 cache hit rate of the SOTA works



(f) Different thread organization of Skiff

Fig. 12. The evaluation results of Skiff vs. the state-of-the-art works.

C. Overall Performance of the Compared Works

We compared the algorithm performance of the proposed method with state-of-the-art works, such as TriCore, TRUST, and BBTC, on the 11 graphs listed in the Table I.

Fig. 12(a) shows the execution time of Skiff and compares it with the state-of-the-art works (note: we just compared the run time in this experiment, while the pre-processing time, such as the graph partition and memory allocation, is omitted). Each bar represents the execution time of the state-of-the-art works. To meet the best performance of TRUST, we turn on all the effective optimization methods according to article [27]'s instructions, such as using the vertex-centric hashing programming model, co-optimizing workload imbalance to handle the hash collision, virtual combination policy to avoid redundant copies in a warp, and recording the vertices according to the out degree. Furthermore, BBTC can achieve the best performance with the hashmap-based intersection policy and partition the graphs into 32 pics [28]. In this experiment, we follow the parameter setting policy of the authors in article [28] to achieve the best performance of BBTC. The experimental result shows that Skiff can achieve the best performance compared to state-of-the-art works, which indicates the proposed optimization method works well on both the power law graph and also the RMAT graph. Furthermore, we can also conclude that only Skiff and TRUST scales with the edges of the evaluated graphs, while only the BBTC achieves a stable runtime on all 11 graphs.

Achieved occupancy is a hardware performance metric that was used to measure the active warps of a particular scheduler relative to the warps on each SM during every clock cycle. A high achieved occupancy indicates there is no idle thread in each warp, and the theoretical occupancy is 100%. Fig. 12(b) shows the achieved occupancy while Fig. 12(c) is the SM efficiency of Skiff VS. the state-of-the-art works. Among all the compared works, only the BBTC has low achieved occupancy, but the others are pretty good, which means all these works except the BBTC can unleash the computing capabilities of GPU. Significantly, the achieved occupancy of TRUST and Skiff is very close to 100%, which indicates both these works can make full use of the parallel computing power of GPU. Thanks to the lock-free scheme, the threads in the Skiff kernel can keep active and can pass the computing results to the next step without waiting, which makes Skiff achieve the best performance on both the achieved occupancy and SM efficiency in comparison to the state-of-the-art works.

SM efficiency is another metric that measures how each kernel used the SM, and it is a high-level metric. SM efficiency is positively correlated with achieved occupancy. Therefore, Skiff can achieve the best performance on SM efficiency compared to the state-of-the-art works, which is shown in Fig. 12(c).

The cache hit rate is one of the most important metrics to measure the effectiveness of the cache. Fig. 12(d) and Fig. 12(e) show the L1 and L2 cache hit rate of Skiff VS the state-of-the-art works. Skiff achieves the best L2 cache hit performance but can only get the best performance on the part of the evaluated graphs. Skiff implemented the SoA data layout to load the homogeneity data in continuous memory space to ensure the coalesced memory access fashion. Skiff further implemented the CTA thread organization model to load as many vertices as possible into the same warp/block. Both of these techniques improved the L2 cache hit. Unfortunately, because most of the evaluated graphs are power law graphs, thanks to the CTA thread organization fashion, there are some huge blocks created for the huge vertices. These huge blocks can reduce the L1 cache hit rate since the L1 cache has limited space.

D. The CTA Thread Organization

In this experiment, we run Skiff using warp-based, block-based thread organization, degree-based workload partition, and index-based workload partition, respectively, to test the effectiveness of the thread organization. According to Section IV-B, we assign $\lceil d(v)/32\rceil$ blocks to the vertex v with degree $d(v)$ in Block_Degree (block-based thread organization with degree-based workload partition) model, assign $\lceil\sum d(v)/32\rceil$ blocks to the vertices v in Block_Index (block-based thread organization with index-based workload partition) model, and assign $\lceil\sum d(v)/32\rceil$ warps to the vertices v in Warp_Index (in warp-based thread organization with index-based workload partition) model.

The effect of the Block_Degree, Block_Index, and Warp_Index thread organization model for Skiff is shown in Fig. 12(f). The experimental result indicates that the Block_Degree and Block_Index thread organization model can achieve better performance than the Warp_Index thread organization model on `livejournal`, while the result reversed on twitter. In general, the Warp_Index thread organization model can achieve the best performance on most graphs.

VI. Related Work

There are many recent works focused on how to accelerate the triangle counting algorithm. To solve the data transfer challenge on the accelerators with limited bandwidth, [29] proposed a processing-in-memory (PIM) architecture. Svelto [30] solved the unbalanced task level parallelism problem by designing a dynamic task scheduling mechanism. CECI [31] proposed the Compact Embedding Cluster Index (CECI) method for subgraph listing, which can be used to locate the triangles. SWTC [32] reduced the duplicate edge counting problem of the streaming graph by designing a fixed-length slicing strategy. Most of these state-of-the-art works are focused on universal architecture, but our works are focused on the GPU accelerator. In this section, we classified the state-of-the-art triangle counting works on GPU into three domains: the matrix multiplication approach, the subgraph matching approach, and the set intersection approach.

Matrix multiplication approach. In the matrix multiplication approach, the triangle is founded by computing the intersections between neighbor lists by using the traditional graph theoretic formulations. DALTON et al. [33] proposed a global sort-based SpGEMM approach. In this work, the authors proposed a reordering scheme to identify the entries of the matrix. Liu et al. [34] developed the GPU SpGEMM algorithm on the basis of SpGEMM, by allocating the memory space for the upper bound size for the short rows first and then for the longer ones to improve memory efficiency. Ata et al. [35] proposed a wedge sampling policy for triangle counting, and Tom et al. [36] proposed a 2D cyclic decomposition method to balance the computation and communication overhead on a distributed-memory system. Gamma [37] designed a new dataflow model to reduce the traffic of data reuse. LOTUS [38] designed a structure-aware memory accessing policy for triangle counting on power-law graphs.

Subgraph matching approach. In the subgraph matching approach, the triangle is set as the basic subgraphs. Ullmann [39] first leverages the subgraph matching approach for triangle counting. In this implementation, the solutions were composed of partially incremental and then completed queries. Based on this work, researchers proposed kinds of evolutionary versions, such as GADDI [40] and SPath [41]. Gunrock [11] is a data-centric graph processing system on GPU, which also implemented a subgraph matching approach for triangle counting. DOULION [3] implemented an MPI-based parallel triangle counting algorithm on distributed memory by dividing the graphs into non-overlapping partitions. Hu et al. [42] proposed a novel method to solve the load imbalance problem and improve GPU parallelism.

Set intersection approach. The basic idea of the set intersection approach is to intersect the two neighbor lists of the two vertices of the same edge and try to find if there are any comment items in these two neighbor lists. This method is straightforward, but the algorithm efficiency is low since there are many redundant traversals. LRC [43] developed a binary search method on GPU and a hash-based segmentation approach to pruning the search space. Similar to LRC, TriCore [10] also developed the binary search-based method on GPU, which organizes one neighbor list as the lookup list and the other one as the binary search tree. This method is easy to implement in parallel but suffers from high time complexity when the vertex does not sort. TRUST [27] implemented a vertex-centric hashing-based triangle counting algorithm on GPU that avoids collisions by reordering the vertex IDs of a graph. Though reordering the vertex ID can reduce most collisions, the reordering overhead is also innegligible. BBTC [28] is a lightweight triangle counting algorithm, that aims to process large-scale graphs that do not fit on a GPU device by partitioning graphs into small pics. Gui et al. [9] designed a prefetching scheme to load the active vertices to reduce the data transfer latency and implemented a heuristic reordering policy to keep the workload balance.

In this paper, we proposed a hash-based intersection triangle counting algorithm on GPU, which is a set intersection approach. Unlike the previous approach, we hash the shorter neighbor list to the hash trie and use the longer one as the search key to reduce the collisions and meet the lock-free mechanism. Furthermore, we introduced a memory friendly bucket placement strategy (trie) for GPU to improve the memory utilization of GPU.

VII. Conclusion and Future Opportunities

This paper introduces Skiff which uses the lock-free policy to implement the hash-based intersection triangle counting on GPU. Particularly, it introduces the lock-free policy on GPU to reduce the conflicts of the hash trie. Skiff further implemented a CTA based thread organization model to keep load balance on real-world graphs, the source code of the proposed work is released on GitHub https://github.com/whu-zhigao/skiff.

The experimental results show Skiff outperforms state-of-the-art works. Even the experimental results of Fig. 12(a) show Skiff scales with the graph size, but it just works on the static graphs. However, it will be another problem once the graph is large enough that can not loaded into a single GPU. In the future, we will focus on other aspects of the triangle counting algorithm, such as triangle counting on distributed systems, triangle counting for dynamic graphs, triangle counting on CPU/GPU hybrid systems, or some other new devices, such as architecture and FPGA.

References

[1] S. M. Arifuzzaman, M. Khan, and M. Marathe, “Parallel algorithms for counting triangles and computing clustering coefficients,” in *Proc. SC Companion: High Perform. Comput., Netw. Storage Anal.*, Piscataway, NJ, USA: IEEE Press, Nov. 2012, pp. 1448–1449.

[2] K. Shin, T. Eliassi-Rad, and C. Faloutsos, “CoreScope: Graph mining using k-core analysis—Patterns, anomalies and algorithms,” in *Proc. IEEE 16th Int. Conf. Data Mining (ICDM)*. Piscataway, NJ, USA: IEEE Press, Dec. 2016, pp. 469–478.

[3] S. Arifuzzaman, M. Khan, and M. Marathe, “Fast parallel algorithms for counting and listing triangles in big graphs,” *ACM Trans. Knowl. Discovery Data*, vol. 14, no. 1, Dec. 2019.

[4] A. Turk and D. Turkoglu, “Revisiting wedge sampling for triangle counting,” in *Proc. World Wide Web Conf. (WWW)*, New York, NY, USA: ACM, 2019, pp. 1875–1885.

[5] S. Bhatia, “Approximate triangle count and clustering coefficient,” in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA: ACM, 2018, pp. 1809–1811.

[6] T. A. Kanewala, M. Zalewski, and A. Lumsdaine, “Distributed, shared-memory parallel triangle counting,” in *Proc. Platform Adv. Sci. Comput. Conf. (PASC)*, New York, NY, USA: ACM, 2018, pp. 1–12.

[7] J. E. Simons and J. Buell, “Virtualizing high performance computing,” *ACM SIGOPS Operating Syst. Rev. (SIGOPS)*, vol. 44, no. 4, pp. 136–145, Dec. 2010.

[8] C. Yang, A. Buluç, and J. D. Owens, “GraphBlast: A high-performance linear algebra-based graph framework on the GPU,” *ACM Trans. Math. Softw.*, vol. 48, no. 1, pp. 1–51, Feb. 2022.

[9] C. Gui, L. Zheng, P. Yao, X. Liao, and H. Jin, “Fast triangle counting on gpu,” in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Piscataway, NJ, USA: IEEE Press, Sep. 2019, pp. 1–7.

[10] Y. Hu, H. Liu, and H. H. Huang, “Tricore: Parallel triangle counting on GPUs,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Piscataway, NJ, USA: IEEE Press, Nov. 2018, pp. 171–182.

[11] Y. Wang et al., “Gunrock: GPU graph analytics,” *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 1–49, Aug. 2017.

[12] H. Liu, H. H. Huang, and Y. Hu, “IBFS: Concurrent breadth-first search on GPUs,” in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA: ACM, 2016, pp. 403–416.

[13] R. Kelly, B. A. Pearlmutter, and P. Maguire, “Lock-free hopscotch hashing,” in *Proc. Symp. Algorithmic Princ. Comput. Syst. (APOCS)*, Philadelphia, PA, USA: SIAM, 2020, pp. 45–59.

[14] Z. Zheng et al., “Feluca: A two-stage graph coloring algorithm with color-centric paradigm on GPU,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 160–173, Jan. 2021.

[15] X. Shi et al., “Frog: Asynchronous graph processing on gpu with hybrid coloring model,” *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 1, pp. 29–42, Jan. 2018.

[16] O. Green, P. Yalamanchili, and L.-M. Munguía, “Fast triangle counting on the GPU,” in *Proc. 4th Workshop Irregular Appl.: Archit. Algorithms (IA3)*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 1–8.

[17] R. R. McCune, T. Weninger, and G. Madey, “Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing,” *ACM Comput. Surveys*, vol. 48, no. 2, pp. 1–39, Oct. 2015.

[18] A. Roy, I. Mihailovic, and W. Zwaenepoel, “X-stream: Edge-centric graph processing using streaming partitions,” in *Proc. 24th ACM Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA: ACM, 2013, pp. 472–488.

[19] J. Zhou, C. Xu, X. Chen, C. Wang, and X. Zhou, “Mermaid: Integrating vertex-centric with edge-centric for real-world graph processing,” in *Proc. 17th IEEE/ACM Int. Symp. Cluster, Cloud. Grid Comput. (CCGRID)*, 2017, pp. 780–783.

[20] M. J. Steindorfer and J. J. Vinju, “Optimizing hash-array mapped tries for fast and lean immutable JVM collections,” in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl. (OOPSLA)*, New York, NY, USA: ACM, 2015, pp. 783–800.

[21] Z. Zheng, X. Shi, and H. Jin, “Parallel overlapping community detection algorithm on gpu,” *IEEE Trans. Big Data*, vol. 9, no. 2, pp. 677–687, Feb. 2023.

[22] J. Leskovec, “Stanford network analysis project,” Nov. 2009. Accessed Jul. 11, 2018. [Online]. Available: http://snap.stanford.edu/index.html

[23] C. S. D. of the University of Milan, “Laboratory for web algorithmics,” Nov. 2002. Accessed Jul. 17, 2018. [Online]. Available: http://law.di.unimi.it/index.php

[24] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Int. Conf. World Wide Web (WWW)*, New York, NY, USA: ACM, 2004, pp. 595–602.
[25] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 20th Int. Conf. World Wide Web (WWW)*, New York, NY, USA: ACM, 2011, pp. 587–596.
[26] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-efficient graph processing on GPUs," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, Piscataway, NJ, USA: IEEE Press, 2015, pp. 39–50.
[27] S. Pandey et al., "Trust: Triangle counting reloaded on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2646–2660, Nov. 2021.
[28] A. Yaşar, S. Rajamanickam, J. W. Berry, and U. V. Çatalyürek, "A block-based triangle counting algorithm on heterogeneous environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 444–458, Feb. 2022.
[29] X. Wang et al., "Triangle counting accelerations: From algorithm to in-memory computing architecture," *IEEE Trans. Comput.*, vol. 71, no. 10, pp. 2462–2472, Oct. 2022.
[30] M. Minutoli et al., "Svelto: High-level synthesis of multi-threaded accelerators for graph analytics," *IEEE Trans. Comput.*, vol. 71, no. 3, pp. 520–533, Mar. 2022.
[31] B. Bhattarai, H. Liu, and H. H. Huang, "CECI: Compact embedding cluster index for scalable subgraph matching," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA: ACM, 2019, pp. 1447–1462.
[32] X. Gou and L. Zou, "Sliding window-based approximate triangle counting with bounded memory usage," *VLDB J.*, vol. 32, no. 5, pp. 1087–1110, Mar. 2023.
[33] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—Matrix multiplication for the GPU," *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 1–20, Oct. 2015.
[34] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 370–381.
[35] A. Turk and D. Turkoglu, "Revisiting wedge sampling for triangle counting," in *Proc. World Wide Web Conf. (WWW)*, New York, NY, USA: ACM, 2019, pp. 1875–1885.
[36] A. S. Tom and G. Karypis, "A 2d parallel triangle counting algorithm for distributed-memory architectures," in *Proc. 48th Int. Conf. Parallel Process. (ICPP)*, New York, NY, USA: ACM, 2019, pp. 1–10.
[37] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2021, pp. 687–701.
[38] M. Koohi Esfahani, P. Kilpatrick, and H. Vandierendonck, "Lotus: Locality optimizing triangle counting," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, New York, NY, USA: ACM, 2022, pp. 219–233.
[39] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, Jan. 1976.
[40] S. Zhang, S. Li, and J. Yang, "Gaddi: Distance index based subgraph matching in biological networks," in *Proc. 12th Int. Conf. Extending Database Technol.: Adv. Database Technol. (EDBT)*, New York, NY, USA: ACM, 2009, pp. 192–203.
[41] P. Zhao and J. Han, "On graph query optimization in large networks," *Proc. VLDB Endowment*, vol. 3, no. 1–2, pp. 340–351, Sep. 2010.
[42] L. Hu, L. Zou, and Y. Liu, "Accelerating triangle counting on GPU," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA: ACM, 2021, pp. 736–748.
[43] N. Ao et al., "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 470–481, May 2011.

Zhigao Zheng (Member, IEEE) is with Wuhan University. He is the executive committee member of the Technical Committee on Distributed Computing & Systems and Embedded Systems of CCF. His research interests include cloud computing, big data processing, and AI systems. He published more than 20 peer-reviewed publications (such as the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and IEEE TRANSACTIONS ON COMPUTERS). He is the PC member of several TOP conferences, such as TheWebConf (formally WWW) and NeurIPS, and the vice PC chair of CPSCom 2023. He is also the editorial board member of some high quality journals, such as the IEEE TRANSACTIONS ON CONSUMER ELECTRONICS, MOBILE NETWORKS AND APPLICATIONS. He joins research projects from various governmental and industrial organizations, such as the National Science Foundation of China (NSFC), the Ministry of Science and Technology, and the Ministry of Education. He is a member of ACM and CCF.

Guojia Wan is a Postdoctoral Fellow with the School of Computer Science, Wuhan University. His research interests include knowledge graph, graph neural networks, and graph for science.

Jiawei Jiang (Member, IEEE) received the Ph.D. degree in computer science from Peking University, China, in 2018. Currently, he is a Professor with the School of Computer Science, Wuhan University, China. His research interests include databases, big data management and analytics, and machine learning systems.

Chuang Hu (Member, IEEE) received the B.S. and M.S. degrees from Wuhan University, in 2013 and 2016, respectively, and the Ph.D. degree from The Hong Kong Polytechnic University, in 2019. Currently, he is an Associate Researcher with the School of Computer Science, Wuhan University. His research interests include edge learning, federated learning/analytics, and distributed computing.

Hao Liu works with Wuhan Digital Engineer Institute. He is a member of the Youth Committee of CICC and a member of the Information Fusion Branch of CSAA. His research interests include information fusion and situation cognition.

Shahid Mumtaz (Senior Member, IEEE) is an IET Fellow, an IEEE ComSoc and ACM Distinguished Speaker, a recipient of IEEE ComSoc Young Researcher Award (2020), the Founder and an Editor-in-Chief of the IET *Journal of Quantum Communication*, the Vice-Chair of the Europe/Africa Region IEEE ComSoc Green Communications Computing Society, and the Vice-Chair of IEEE Standard P1932.1: Standard for Licensed/Unlicensed Spectrum Interoperability in Wireless Mobile Networks. He is an Author of four technical books, 12 book chapters, 300+ technical papers (200+ IEEE journals/transactions, 100+ conference proceedings), and received two IEEE Best Paper Awards in the area of mobile communications. Most of his publications are in the field of wireless communication. He is serving as a Scientific Expert and Evaluator for various research funding agencies. He was awarded an Alain Bensoussan Fellowship in 2012. He was the recipient of the NSFC Researcher Fund for Young Scientists in 2017 from China.

Bo Du received the Ph.D. degree in photogrammetry and remote sensing from the State Key Lab of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University, Wuhan, China, in 2010. Currently, he is a Professor with the School of Computer Science and Institute of Artificial Intelligence, Wuhan University. He is also the Director of National Engineering Research Center for Multimedia Software, Wuhan University, Wuhan, China. He has more than 80 research papers published in the IEEE TRANSACTIONS ON IMAGE PROCESSING, IEEE TRANSACTIONS ON CYBERNETICS, IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING. Fourteen of them are ESI hot papers or highly cited papers. His research interests include machine learning, computer vision, and image processing. He serves as an Associate Editor for *Neural Networks*, *Pattern Recognition*, and *Neurocomputing*. He also serves as a Reviewer of 20 Science Citation Index (SCI) magazines including IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, IEEE TRANSACTIONS ON CYBERNETICS, IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, IEEE TRANSACTIONS ON IMAGE PROCESSING, IEEE JOURNAL OF SELECTED TOPICS IN APPLIED EARTH OBSERVATIONS AND REMOTE SENSING, and IEEE GEOSCIENCE AND REMOTE SENSING LETTERS. He won the Highly Cited Researcher (2019 and 2020) by the Web of Science Group. He won the IEEE Geoscience and Remote Sensing Society 2020 Transactions Prize Paper Award. He won the International Joint Conferences on Artificial Intelligence (IJCAI) Distinguished Paper Prize, the IEEE Data Fusion Contest Champion, and the IEEE Workshop on Hyperspectral Image and Signal Processing Best Paper Award, in 2018. He regularly serves as a senior PC member of IJCAI and AAAI. He served as Area Chair for ICPR.