Efficient Load-Balanced Butterfly Counting on GPU

Qingyu Xu
Renmin University of China
qingyuxu@ruc.edu.cn

Feng Zhang*
Renmin University of China
fengzhang@ruc.edu.cn

Zhiming Yao
Renmin University of China
jimmyyao@ruc.edu.cn

Lv Lu
Renmin University of China
lvlu@ruc.edu.cn

Xiaoyong Du
Renmin University of China
duyong@ruc.edu.cn

Dong Deng
Rutgers University - New Brunswick
dong.deng@rutgers.edu

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

ABSTRACT

Butterfly counting is an important and costly operation for large bipartite graphs. GPUs are popular parallel heterogeneous devices and can bring significant performance improvement for data science applications. Unfortunately, no work enables efficient butterfly counting on GPU currently. To fill this gap, we propose a GPU-based butterfly counting, called G-BFC. G-BFC addresses three main technical challenges. First, butterfly counting involves massive serial operations, which leads to severe synchronization overheads and performance degradation. We unlock the serial region and utilize the shared memory on GPU to efficiently handle it. Second, butterfly counting on GPU faces the workload imbalance problem. We develop a novel adaptive strategy to balance the workload among threads for efficiency. Third, butterfly counting in parallel suffers from the traversal of the huge amount of two-hop paths, also called wedges, in bipartite graphs. We develop a novel preprocessing strategy, which can effectively reduce the number of wedges to be traversed. Experiments show that G-BFC brings significant performance benefits. On eleven real datasets, G-BFC achieves 19.8× performance speedup over the state-of-the-art solution.

PVLDB Reference Format:
Qingyu Xu, Feng Zhang, Zhiming Yao, Lv Lu, Xiaoyong Du, Dong Deng, and Bingsheng He. Efficient Load-Balanced Butterfly Counting on GPU. PVLDB, 15(11): 2450 - 2462, 2022.
doi:10.14778/3551793.3551806

1 INTRODUCTION

Butterfly counting (BFC) has been proven to be an important and costly operation for large bipartite networks [43]. Butterfly counting not only proves to be of great significance in graph theory [4], but also serves as a primitive in many bipartite graph operations, such as measuring graph cohesion [4], clustering coefficient [19], and community structure [11]. In these bipartite graph operations, BFC accounts for the majority of the entire execution time, even more than 80% execution time. Even worse, in many real world applications like spam detection [12], we have to repeatedly count clustering coefficient that uses BFC [19]. Thus, there is an urgent need in accelerating BFC. On the other hand, BFC has a lot of potential in parallelism. Since GPUs have been used as powerful accelerators in many data-intensive applications [18, 25, 31], this paper studies GPU accelerations for BFC on large bipartite graphs.

Accelerating butterfly counting on GPU is of great significance. GPU can provide magnitude performance over CPU. For example, the Nvidia RTX 3090 GPU provides 35.7 TFLOPS computing capacity, which is about 40× over that of the Intel core i9 10900X CPU on our platform. Moreover, current bipartite networks are becoming extremely large. For example, if we construct a bipartite graph for Taobao of Alibaba group, the graph involves 500 million customers with 800 million commodities, and its scale is still growing [1]. It takes a long time to process large graphs, and heterogeneous acceleration is a potential solution to reduce the latency of BFC.

The algorithm of butterfly counting itself brings about three major technical challenges, given the special architecture characteristics of GPU. First, the parallel butterfly counting algorithm requires locks for consistency, but synchronization on GPUs can incur significant performance degradation. Second, the relations in bipartite graphs can be very imbalanced. For instance, a bipartite graph can be used to depict the interaction between consumers and goods in a purchasing system. In many user-commodity bipartite graphs, the number of purchased commodities, which is the number of neighbors from users, varies dramatically among different users, causing severe workload imbalance. Given that a GPU has tens of thousands of threads, distributing the workload to different threads can have an imbalance problem. Meanwhile, threads in GPU execute in a SIMT (single instruction multiple threads) mechanism. As a result, workload imbalance, or thread divergence, results in significant thread stalls and causes a sharp fall in performance. Third, the butterfly counting algorithm suffers from traversing a huge number of wedges, which are the two-hop paths in bipartite graphs. Even when using GPU, random wedge accesses are still a bottleneck for improving the algorithm.

With the bipartite graphs becoming more and more popular, there is a rapidly growing interest in accelerating the butterfly counting algorithm. For example, Shi et al. [38] proposed a parallel method for butterfly counting on the multi-core CPU. However, existing BFC works [35, 38, 43] mainly focus on accelerating BFC on CPU, and do not take GPU's characteristics into consideration. Performing BFC algorithm on GPU efficiently is non-trivial because of the aforementioned challenges. For instance, to store the priority queue, we need to optimize the utilization of GPU global memory that has high memory access latency. Moreover, there is a large

*Feng Zhang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551806

amount of literature accelerating graph algorithms with GPU, such as triangle counting [16], BFS [41], and PageRank [46]. Unfortunately, we identify two major drawbacks in applying these works on accelerating BFC on GPU. First, applying existing GPU-based graph processing studies to BFC in bipartite graphs requires complicated design. For example, we need to identify all sub-structures and add extra restrictions to count butterflies in bipartite graphs, which degrades performance. Second, we need to maintain extra lists in these methods for storing all sub-structures, which further increases the memory access speed and consumes large memory space. Based on these analyses, existing methods cannot be directly applied to BFC on GPUs.

We design G-BFC, the first solution that provides **G**PU-based **B**utter**F**ly **C**ounting, which effectively solves the challenges mentioned above. First, to efficiently handle the heavy dependencies in butterfly counting, we develop a fine-grained lock-free algorithm on GPU, which unlocks the serial region and minimizes the synchronization when multiple threads update the shared result simultaneously. Second, to solve the imbalance challenge in partitioning bipartite networks, we develop an adaptive workload partitioning strategy, which assigns vertices with a close amount of workload to be handled by the same number of threads. Third, to reduce the cost of huge time complexity in traversing wedges, we develop a novel preprocessing method that applies rule filters to reduce the number of legal wedges.

We evaluate G-BFC on eleven real-world datasets. Experiments show that G-BFC achieves an average of 19.8× speedup over the state-of-the-art butterfly counting solution. In addition, the experimental results demonstrate that G-BFC sharply reduces the number of processed wedges by 18%. Our paper has made the following contributions.

- We develop a fine-grained parallel lock-free BFC algorithm on GPU, which reduces the number of synchronizations significantly.
- We develop an adaptive workload partitioning strategy, which partitions the bipartite graphs in a balanced strategy efficiently.
- We develop a novel preprocessing method applying rule filters to reduce the cost of huge time complexity in traversing wedges efficiently.

The rest of this paper is organized as follows. Section 2 shows the problem definition, preliminaries, and existing solutions. Section 3 shows our motivation and framework for parallel butterfly counting. Section 4 shows our basic design of GPU-based butterfly counting. Section 5 shows the adaptive load balancing optimization, and Section 6 shows the memory aware edge direction optimization. Section 7 reports our experimental results, and Section 8 concludes this paper.

2 PRELIMINARIES AND RELATED WORK

2.1 Problem Definition

In this section, we provide the formal definition of mathematical symbols used in this work. All symbols appearing in this paper can be found in Table 1.

Bipartite graphs. Our solution takes undirected bipartite graphs as input, as shown in Figure 1. The bipartite graph $G(V, E)$ has two layers, as shown in Figure 1 (a), and we define $V = (L, R)$. $L(G)$ denotes the left set of vertices and $R(G)$ denotes the vertices on the right side. Meanwhile, $L(G) \cap R(G) = \emptyset, V(G) = L(G) \cup R(G)$ is the complete vertex set, and $E(G)$ denotes the edges between $L(G)$ and $R(G)$. We use n_E and n_V to denote the numbers of edges and vertices in $E(G)$ and $V(G)$, respectively. In addition, the numbers of vertices of $L(G)$ and $R(G)$ are denoted as n_L and n_R. We use (u, v) to represent an edge, where u and v must come from different layers. The set of neighbors of vertex u is defined as $N^G(u) = \{v|(u, v) \in E(G)\}$. The degree of vertex u is defined as $deg(u) = |N^G(u)|$. To demonstrate our algorithm clearly, we let $N_2^G(u)$ denote the two-hop neighbors of vertex u (any vertices that are reachable from u by a path of length of two), and we let $N_3^G(u)$ denote the three hop neighbors of vertex u (any vertices that are reachable from u by a path of length of three). When we read a graph, we re-encode the IDs of vertices so all vertices have a unique ID. We decide to let $L(G)$ start from zero in a consecutive manner.

Table 1: Symbols and meanings.

Symbol	Meaning
G	an undirected bipartite graph
V(G)	the vertices of G
E(G)	the edges of G
L(G),R(G)	the two differnt layers of G
u,v	vertices
(u_i, v_j, u_k)	a wedge in G
(u_i, v_j, u_k, v_h)	a butterfly in G
$N^G(u)$	the neighbor sets of vertex u
$N_2^G(u)$	the 2-hop neighbor sets of vertex u
$N_3^G(u)$	the 3-hop neighbor sets of vertex u
deg(u)	the degree of vertex u
B(u)	the number of butterflies containing a vertex u
B(G)	the number of butterflies in G
n_E, n_V	the number of edges and vertices in G

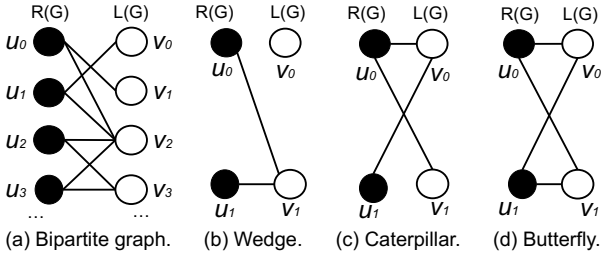


Figure 1: Wedges and butterfly patterns.

Wedge. Given an undirected bipartite graph $G(V, E)$, we define a wedge in G as (u_i, v_j, u_k), $s.t.\ i! = k$, where u_i and u_k come from the same layer, but v_j differs. Wedges can also be defined as all the 2-length paths that vertices $L(G)$ can reach to the vertices of set $R(G)$. Typically, we refer to u_i, v_j, and u_k as the start, middle, and the end vertex respectively. As shown in Figure 1 (b), we designate u_0 the start vertex, v_1 the middle vertex, and u_1 the end vertex.

Caterpillars. A caterpillar in a given undirected bipartite graph $G(V, E)$ can be defined as a path of length three. When we conduct

butterfly counting using list intersection, we need to traverse the caterpillars in bipartite graphs.

Butterfly. A butterfly can be denoted as (u_i, v_j, u_k, v_h), $s.t.\ i! = k, j! = h$, as shown in Figure 1 (d). It contains four vertices. u_i and u_k come from the same layer, while v_j and v_h come from the other layer. A butterfly has four edges, (u_i, v_j), (u_i, v_h), (u_k, v_j), and (u_k, v_h) in $E(G)$. A butterfly can also be regarded as two wedges that share the same start vertex and end vertex, but differ in the middle vertex. The number of butterflies is represented by $B(G)$, and our goal is to compute $B(G)$ in a given bipartite graph G.

Butterfly counting. Accordingly, we define butterfly counting as the procedure to retrieve $B(G)$ in a given bipartite graph $G(V, E)$.

2.2 Applications of Butterfly Counting

Bipartite graphs gain increasing attention in recent years, and there is a recent surge in studying butterfly counting. Butterfly counting has been used as an important primitive in many graph algorithms [4, 24, 26, 34, 53]. We list the following applications for illustration.

- **Graph cohesion** [24] calculates the minimum number of edges whose deletion can make the graph disconnected. It unveils hidden orderings and graph hierarchies [8, 28]. The number of butterflies per vertex can be used to calculate graph cohesion.
- **Clustering coefficient** [4] represents the probability that any two neighbors of a vertex in a bipartite graph have the same neighbors. It serves as an important indicator of community structure [4], and can be measured by the fraction of three-path connected components that form butterflies in bipartite graphs.
- **K-wing** [36] is the largest subgraph of a bipartite graph with each edge contained in at least k butterflies. It is useful in many real-world applications, such as social network analysis [11] and community detection [9]. Butterfly counting serves as the foundation of computing k-wing in bipartite graphs [36].

Moreover, with the development of big data technology [16, 41, 46, 50–52], other graph applications, such as realistic graph models [23], abnormal activities detection [6], and social intercorporate relation detection [29, 34], also include butterfly counting as a major component. We demonstrate the time breakdown for these applications on three graphs (Stack, BC-rate, and A-rate) in Figure 2. The test algorithm is the state-of-the-art approach on the CPU [4, 36]. More experimental setup details can be found in Section 7.3. Butterfly counting (BFC) occupies 38% to 86% of the total time, which becomes the bottleneck of these applications and needs GPU acceleration desperately.

2.3 Existing Solutions

In this section, we first introduce the basic algorithm of butterfly counting. Then, we briefly discuss the state-of-the-art methods.

Basic butterfly counting. We first introduce the hashmap-based BFC. This BFC randomly selects one side of the vertices as the start vertices. Then, it visits the neighbors of the start vertices, and records the number of collisions for each 2-hop neighbor. Another option of butterfly counting is to find the intersection of the 3-hop neighbors and neighbors of the vertices. The state-of-the-art BFC [43] focuses on the hashmap-based method. Our solution is also based on this method. We give the pseudocode of the hashmap-based BFC in Algorithm 1.

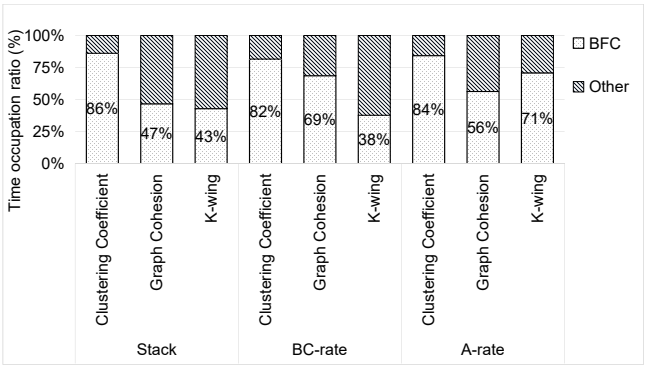


Figure 2: Total time breakdown.

Algorithm. Algorithm 1 first randomly selects a layer as the start layer (Line 2), and initializes a hashmap at each outermost for loop (Line 4). Then, it traverses all wedges for each vertex and updates the butterfly counts before updating the hashmap (Lines 5 to 7). After finishing traversing wedges of a vertex, it resets the hashmap to 0 (Line 8). In this method, since one layer is randomly selected as the starting layer, the time complexity reaches $O(\sum_{u \in L(G)} deg(u)^2)$.

Example. We illustrate the workflow of Algorithm 1 in Figure 3. We use $wedge\_num()$ to denote the list that stores the number of wedges for each vertex. Each vertex has a private copy of $wedge\_num()$. We choose $R(G)$ as our start layer. $B(G)$ and $wedge\_num()$ vectors are initialized to 0.

Algorithm 1: BFC-WEDGE [35]

```
   input  : A bipartite graph G
   output : B(G)
1  B(G) ← 0
2  S ← R(G)
3  foreach u ∈ S do
4      initialize hashmap wedge_num with 0;
5      foreach v ∈ N^G(u) do
6          foreach w ∈ N^G(v) and w.id > u.id do
7              B(G) ← B(G) + wedge_num(w)
               wedge_num(w) ← wedge_num(w) + 1
8      reset(wedge_num)
9  return B(G)
```

First, we traverse u_0's 2-hop neighbors through v_0. We visit u_1, but $wedge\_num(u_0, u_1)$ is still 0. Hence, we add 0 to $B(G)$, and then add 1 to $wedge\_num(u_0, u_1)$. Second, we visit u_1 again through v_1. The current value of $wedge\_num(u_0, u_1)$ is 1, so we add 1 to $B(G)$ accordingly, which means that we have detected one butterfly and updated the value of $wedge\_num(u_0, u_1)$ to 2. Third, we continue the process until all edges have been traversed (from dashed line to solid line in Figure 3). In the example, we obtain $B(G) = 3$ following Algorithm 1.

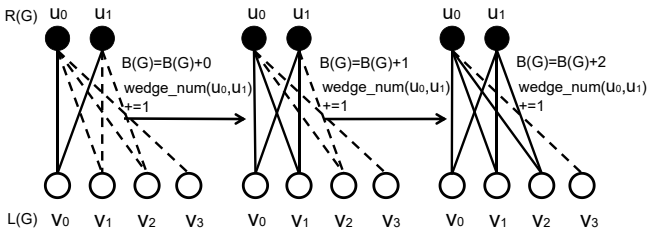


Figure 3: BFC workflow.

State-of-the-art approaches on the CPU. Based on Algorithm 1, Sanei et al. [35] utilized the degree information of layers. They choose the layer with a lower vertex degree as the starting layer, and reduce the time complexity to O(Min($\sum_{u \in L(G)} deg(u)^2$, $\sum_{v \in R(G)} deg(v)^2$). Wang et al. [43] further proposed an algorithm that does not count butterflies from only one side of the layers. Instead, they select the vertices with a large degree as the end vertices by marking the priority of the vertices, further reducing the time complexity of the algorithm to O($\sum_{(u,v) \in E(G)} Min(deg(u), deg(v))$).

2.4 GPU Parallelism

GPUs, as powerful accelerators, have been widely applied to data science applications [30, 47–49]. Different from CPU, GPU has a much powerful compute ability with thousands of computing cores and high memory bandwidth. Although new GPUs are released every year, they share almost the same CUDA parallel architecture. Accordingly, CUDA, as the most popular GPU programming model, can be used for different generations of GPUs. Threads in CUDA are organized as *blocks*, and the blocks form a *grid*. The cores on GPU are grouped into streaming multiprocessors, and a thread block is mapped to one streaming multiprocessor. During execution, threads are executed at *warp* granularity, which means that a fixed collection of threads needs to execute in a SIMT (single instruction, multiple threads) mechanism. The memory hierarchy is also sophisticated. On GPU, registers provide the fastest access speed. Along with regular caches, GPU provides a fast controllable cache, called shared memory, on each streaming multiprocessor. Threads within a block can access the shared memory accordingly, which needs to be carefully designed. The largest memory on the GPU is global memory, which is accessible to all threads, but with very high latency.

3 MOTIVATION

3.1 Revisiting Previous Butterfly Counting

Observation. Although the time complexity of the state-of-the-art method is O($\sum_{(u,v) \in E(G)} Min(deg(u), deg(v))$), several studies [38, 43] notice that the main workload of butterfly counting comes from traversing all wedges, and accordingly, they put forward corresponding solutions. For example, the state-of-the-art BFC [38] supports parallelism on CPU. It allocates partial memory for each thread to avoid writing conflicts and explores different scheduling methods. However, this solution does not consider the massive computational power and special memory hierarchy of GPU. To fully utilize GPU, we need to launch thousands of threads, and these massive threads execute simultaneously. Furthermore, we cannot afford to allocate a private local memory copy for each thread with the GPU memory limitation. In detail, assume each thread is allocated 0.1MB memory buffer, then 1024*1024 threads require 102.4GB memory, which is much larger than the memory capacity of the latest GPUs. Therefore, we need to design a more fine-grained GPU-parallel version of the BFC algorithm.

Why existing GPU-based graph algorithms do not apply? Many graph algorithms have been proposed on GPUs, and the closest work to ours is triangle counting. However, these solutions cannot be applied to BFC. Butterfly counting and triangle counting are inherently different, because intuitively, the structures they calculate are different. Moreover, existing triangle counting algorithms focus on the list intersection method, which can be parallelized using binary search algorithm [17, 18], but incurs high time complexity in butterfly counting. The reason is that we have to conduct list intersection method for not only each vertex's neighbors and 2-hop neighbors but also 3-hop neighbors. In addition, there are novel triangle counting works considering load imbalance problem on GPU [7], but ours differs from theirs in kernel function designs, detailed in Section 5.2. There are also algorithms targeting rectangles and 5-vertices structures [20, 32, 33]. However, they do not target on bipartite graphs [54] and can have quadratic works, which cause high time and space complexities [10, 39, 42]. Furthermore, we cannot project a bipartite graph to a unipartite graph, because such a projection can lead to information loss and pattern changes [36, 37]. Therefore, existing GPU-based graph algorithms do not apply and it is critical to develop a fine-grained BFC algorithm on GPU.

3.2 Design Overview

To solve the challenges against efficient butterfly counting on GPU, we develop GPU-based butterfly counting, called G-BFC, which includes a series of novel designs. Our first design is to utilize GPU parallelism and reduce serial region in butterfly counting. We propose a lock-free algorithm combined with the utilization of GPU memory hierarchy to reduce the waiting overhead and huge memory access latency. More details are discussed in Section 4. Our second design is a fine-grained adaptive strategy, which can solve the workload imbalance problem. We analyze the G-BFC process, and develop an adaptive thread allocation method in accordance with the length of adjacency lists. In addition, we adjust the parallel loop according to the length of the list. We further explore the possibility of using list intersection based method to optimize our algorithm. More details are discussed in Section 5. Our third design is to reduce the overhead of traversal of redundant wedges. Because the main workload of G-BFC comes from the huge number of vertices to be traversed, it is crucial for us to narrow the scope of wedges. We develop a novel preprocessing method to reorder the input bipartite graph by utilizing the vertices' degree information. More details are discussed in Section 6.

Workflow. We demonstrate our workflow in Figure 4, which can be divided into three major steps. First, we pre-process the graph by directing the edges. In detail, this process reduces the memory consumption for the input graph and accelerates the later steps in the workflow, which is the design to reduce the overhead of traversal of redundant wedges. Second, we apply diverse parallelization for different vertices dynamically to balance workload, which is

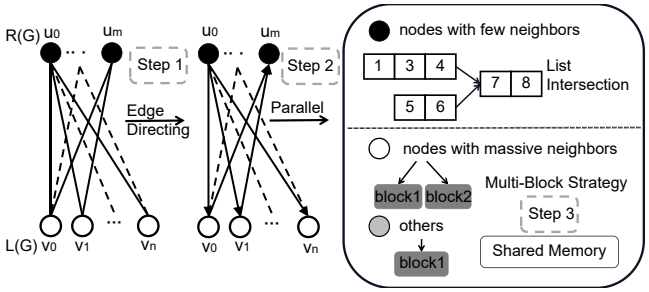


Figure 4: G-BFC Workflow.

the design to solve the workload imbalance problem. Third, we utilize shared memory in GPU to reduce serial regions and minimize waiting latency for the threads in the same block.

4 GPU-BASED BUTTERFLY COUNTING

In this section, we introduce our basic design of G-BFC. First, we propose a lock-free design in accordance with GPU characteristics to remove the busy-waiting overhead. Second, we analyze the memory usage of G-BFC to ensure its scalability. Third, we demonstrate our G-BFC design.

4.1 Lock-Free Design for Butterfly Counting

Section 2 shows that the serial BFC algorithm updates a variable and a hashmap to count the butterflies. However, in the parallel design, if the variable and hashmap are visible to massive threads, write conflicts occur. Intuitively, to avoid conflicts, we can restrict that only one thread can access the shared variable and hashmap when there are multiple operations that can potentially cause conflicts. Accordingly, we apply an extra lock to guarantee consistency. When the locked region contains multiple steps, the time spent waiting for the thread to release the lock can be significant. As a result, the serial region must be minimized or even removed. There are three options for reducing waiting overhead when we develop BFC on GPUs.

- **Shared lock vector.** We can use a thread block to process a vertex, and threads inside a block share the same copy of hashmap. To guarantee the consistency of the hashmap, we utilize a vector of locks when reading and writing the hashmap.
- **Thread-level hashmap.** To completely remove the atomic operations, we develop a thread-level kernel, which assigns one thread to handle a vertex. Accordingly, each thread has a private hashmap.
- **Operation rearrangement.** We still apply the block-level kernel, but to minimize the overhead, we remove the lock vectors and move the read operation of the hashmap to the end of the process.

Analysis. The first option utilizes a shared lock vector. We allocate a shared lock vector that has the same size as the shared hashmap. Once a thread is updating the hashmap, it needs to update the corresponding position of the lock vector first so that the other threads can access other positions of the shared hashmap to avoid conflicts. Since the serial region still exists, only a minor reduction in overhead is possible. The second option is to allocate a thread to count a vertex and let each thread have its own hashmap. Although we cannot afford to allocate a private local copy for each thread, there are vertices with few neighbors. In detail, for the groups of vertices whose degrees are smaller than 16, we allocate a private copy for them to accelerate the performance of G-BFC. In this circumstance, we do not need to have lock operations. However, we need to consider the usage of the various memory hierarchy. Assume that we have a global memory of M GB, and we choose to start from L_G. Then, each hashmap takes $|L_G|$*4 (the size of an integer) byte memory space. In real-world datasets, it is common that even a small dataset can have thousands of vertices. Hence, we have to repeat the execution of the kernel for $|L_G|^2 * 4/M$ times to get all vertices processed. We cannot launch numerous threads because of the limited resources of each streaming multiprocessor. In addition, this option causes high warp divergence, which can hurt the performance. The third option is to rearrange operations and unlock the serial region, which can tackle the shortcomings of the former choices. Hence, we choose the third option.

Design. The fine-grained G-BFC is developed in accordance with the GPU characteristics. To limit warp divergence and speed up the operation, we assign a thread block to each vertex at first. Second, we carefully unlock the serial region by rearranging the reading operations to avoid conflicts. We defer the add-up operation after we complete updating the hashmap. Third, we sum up the final butterfly counts using the warp-level communication functions.

4.2 Memory Hierarchy Utilization

As mentioned above, we use hashmap to store the number of visited times of each vertex, which causes a great consumption of GPU memory resources and cannot be loaded into the shared memory and registers. Therefore, we must store the hashmap in the global memory of the GPU. However, this leads to a problem: the data needs to be read and written through global memory multiple times, and the memory access overhead is large.

Considering the random memory access of 2-hop vertices, we reduce the memory access overhead by taking advantage of the GPU memory hierarchy. In detail, we consider using the controllable shared memory, which provides fast access speed. The benefits of using shared memory are enormous. In particular, accessing global memory can be 150 times slower than accessing shared memory, which implies that we access 150 vertices in the shared hashmap in the same amount of time as accessing one vertex in global memory. Accordingly, based on our original design in Section 4.1, we load the vertices stored in the original hashmap into the shared memory part by part, and only consider the number of butterflies between the current start vertex and the vertices stored in the shared memory each time. During each time, we update the visited times of vertices in shared memory and sum up the butterflies in the shared hashmap. We then reload the next partition of vertices in shared memory and continue the counting procedure until we finish processing the adjacency list of the vertex.

4.3 G-BFC

Design. Classic BFC algorithm uses hashmap for efficiency. The hashmap is reset to 0 after each vertex is calculated. Therefore, we allocate threads and hashmap space for each vertex in parallel.

Initially, to launch more threads to maximize the computing power of GPU, we choose the layer with a large number of vertices as the starting layer for parallelism. However, since the global memory of the GPU is limited, we cannot load all the hashmaps for every block into the global memory at one time. Accordingly, we change our starting layer to the other side. As discussed in Section 4.1, we use a block to handle the workload of a vertex. Because of the limited memory, we set the grid dimension to its maximum to maximize the GPU parallelism. However, due to the limited shared memory resources, we store the adjacency list and offset of the vertex in the constant memory of the GPU, which can be shared globally. As stated in Section 4.2, we load the hashmap to shared memory for fast memory access. Note that G-BFC is not a simple transplant of BFC-WEDGE [35]. First, we reduce the overhead of the serial region by carefully rearranging the operation of the BFC-WEDGE. Second, we have tried different methods to alleviate the time cost of GPU AtomicAdd operations. Third, we design other optimizations targeting GPU characteristics, such as load balancing and edge direction, detailed in Section 5 and Section 6.

Example. We show an example of G-BFC in Figure 5, which calculates the number of butterflies of a vertex. For instance, we launch a block for u_0. When traversing the neighbor vertices of u_0, we assign the neighbors to different threads in the block for parallel processing. For example, $thread_0$ in $block_0$ visits the neighbor vertices of v_0 in $L(G)$. However, due to the limited shared memory resources, we need to load the 2-hop neighbor vertices of u_0, $N_2^G(u_0)$, into memory part by part in different loops. For example, if we load 128 vertices at a time, we only need to update the map for the first 128 vertices in this loop, which are u_0 to u_{127}. Then, we sum up the butterflies in the map by shuffle and sum operations. $Thread_0$ continues this process until all vertices in $N_2^G(u_0)$ have been processed. So do the other threads in $block_0$. These threads perform the same operations for different vertices in $N^G(u_0)$.

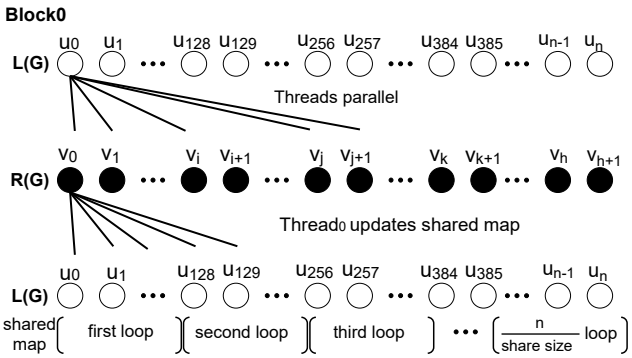


Figure 5: G-BFC example.

Algorithm. We show the pseudocode of G-BFC in Algorithm 2. Each block handles a vertex, and different threads in the block handle different neighbor vertices. Therefore, each block has a private hashmap, which can be stored in shared memory and added up to obtain the butterfly counts.

Next, the processing flow of each vertex is as follows. First, we choose the layer with fewer vertices as the start layer, which is denoted as S (Lines 2 to 4). We initialize the shared hashmap to 0 (Line 6) by assigning this task to the first thread of each block. Second, we obtain the neighbors of the vertex by analyzing its adjacency list (Line 9). Each thread traverses the corresponding neighbor list and accesses the 2-hop neighbors (Line 10). Third, we judge whether the 2-hop neighbors are stored in shared memory. If so, we update the shared map, and broadcast this update to the other threads (Line 12). Here, we use $atomicadd(add, Val)$ to update the shared map because multiple threads can visit the map. Then, we need to use the $threadfence()$ function to ensure the consistency of the shared map (Line 13). We assign the block to another vertex after it finishes processing the current workload (Line 15). Every time the 2-hop neighbors are processed, the results in the shared memory must be updated. In the *block_reduce* operation, because the threads in a block are responsible for different parts, a private copy of butterfly counts is saved to their own local variables. After processing the second for-loop in Line 8, that is, after processing this vertex, we add up the number of butterflies in each thread, using the shuffle and sum operations to obtain the number of butterflies for this vertex (Line 14). It is worth noting that we not only need block synchronization, but also need synchronization every time we set and reset the shared map.

Algorithm 2: G-BFC

```
   input  : A bipartite graph G
   output : B(G)
1  B(G) ← 0
2  S ← R(G)
3  if |R(G)| > |L(G)| then
4      S ← L(G)
5  foreach u = blockIdx.x and u.id < |S| do
6      if threadIdx.x==0 then
7          initialize shared hash map sh_wedge_num with 0;
8      foreach k ∈ [0, |S|/sharedMemorySizeInBlock] do
9          foreach v ∈ N^G(u) do
10             foreach w ∈ N^G(v) do
11                 if w.id ∈ sh_wedge_num and w.id > u.id
                   then
12                     atomicAdd(sh_wedge_num(w))
13     threadfence();
14     B(G) ← blockreduce(sh_wedge_num)
15     u.id += gridDim
16 return B(G)
```

Complexity analysis. In Algorithm 2, the complexity of the preparation from Lines 1 to 4 is $O(1)$. The for-loop in Line 5 takes $O(S)$ times, where S is the number of vertices in the layer with a smaller number of vertices. It can be reduced to $O(S/P)$ with parallelism, where P is the number of blocks. The initialization in Line 7 can be done in $O(1)$. The for-loop in Line 8 takes $O(S/M)$, where M is the shared memory space of the block. The two for-loops in Lines 9 and 10 take $O(\sum_u \sum_v deg_{(u,v)\in E}(v))$, where $deg()$ represents the vertex's degree. As a result, the complexity of Algorithm 2 is $O(\frac{S}{M \cdot P} \sum deg^2(v))$.

5 ADAPTIVE BFC LOAD BALANCING OPTIMIZATION

In this section, we explain how we develop the fine-grained workload scheduling strategy.

5.1 BFC Load Balance Design

In this part, we show our general design to address the load imbalance problem. From Section 3, we observe that synchronization inside each block is an important operation that assures the consistency of reading and writing operations of the global memory and shared memory by different threads.

Analysis. Different blocks in our algorithm are responsible for processing different vertex, whose adjacency lists vary in length, and the threads inside each block handle different partitions of the adjacency list. The workloads for threads within each block vary significantly, and the load imbalance problem occurs.

Existing works mainly assign different numbers of blocks or warps to handle vertex with different workloads [21, 44]. However, they are suboptimal in our scenario. These methods do not adopt the optimal butterfly counting algorithm for different kernels. In contrast, we change both the number of threads and kernel according to the application scenario.

For example, Figure 6 shows that the workloads of blocks 0, 1, and 2 are different, and this difference is related to the length of each vertex's adjacency list. Furthermore, the workload fluctuates within each block. For block 0, the lengths of the 2-hop neighbor list processed by distinct threads also vary a lot, resulting in load imbalance at thread granularity within a block. Load imbalance causes delays and impairs the efficiency of G-BFC, as a result of the synchronization procedure. Motivated by these drawbacks, we need to adjust G-BFC according to the analysis. Hence, we can make full advantage of the massive computation power compared with just assigning different numbers of threads.

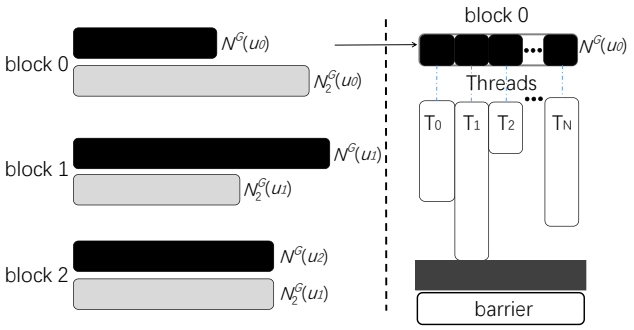


Figure 6: Workload balancing model.

General design. We allocate vertices with different lengths of adjacency lists and 2-hop neighbor lists to corresponding buckets. For vertices with fewer than 16 neighbors, we use a single thread to calculate the butterfly counts. In detail, we utilize the list intersection based method, so there are no conflicts. For vertices with neighbors more than 16 but less than 1024, we use a thread block to handle the vertex, which is the same as we mentioned in Section 4.3. For vertices with neighbors more than 1024, we use multiple blocks to process the vertex. Such processing aims to make better use of GPU resources. We store the graph in the CSR format [15], which is a compact and memory-hierarchy-friendly graph representation. We read the list directly when judging the neighbors of the vertex, with no additional operations required. To calculate the number of 2-hop neighbors of a vertex, we use an estimation method and add the lengths of adjacency lists of all neighbors. We set the thresholds based on the following summaries.

- If the number of neighbors is smaller than 16, we consider using list intersection to perform butterfly counting, and store the adjacency list in shared memory to decrease the memory access overhead.
- If the number of neighbors is between 16 and 1024, then we use a block to process this vertex at the same time. The processing of this vertex is the same as we mentioned in Section 4.3. We adjust the usage of shared memory according to the list length of 2-hop neighbors.
- If the number of neighbors is greater than 1024, we use two blocks to process the workload of this vertex at the same time.

This adaptive method is useful for improving experimental results and for determining the optimal number of threads for processing different vertices. To summarize our selection, we list the characteristics of the vertices as well as our preferences in Table 2. Different kernels vary in butterfly counting algorithms and the utilization of GPU features.

5.2 BFC Kernel Details

Multi-block kernel. As stated in Section 5.1, if the number of neighbors of a vertex exceeds 1024, two blocks are used to process this vertex at the same time. In this case, these two blocks are respectively responsible for the butterfly counting for half of the neighbors. We still combine the shared memory and global memory to perform butterfly counting of this part of the vertex. It is worth noting that the estimated number of 2-hop neighbors of this part of the vertex can be very large. Because in our prior method, a portion of the 2-hop neighbors must be transferred into shared memory for calculation every time, we have the number of loops as $|N_v^2|/SharedMemoryBlockSize$. When $|N_v^2|$ is large, the time overhead caused by the loop can cover the saved memory access time optimized by our design in shared memory. Therefore, for this portion of the vertices, we do not use shared memory part by part for computation. We directly put the map in the global memory for butterfly counting.

Warp kernel. We develop a novel adaptive butterfly counting function, which adjusts the parallel loop in accordance with the vertices' 2-hop neighbor length. This warp kernel is optimized from the kernel function we mentioned in Section 4.3 by adopting load balance strategy. In the kernel function stated in Section 4.3, we assume that the adjacency lists of the neighbors of each vertex have similar lengths, but in real datasets, the distribution of the degree of vertices varies a lot. For the estimation of the number of 2-hop neighbors of a vertex, we use the sampled average length to represent the expectation of the vertices' degree. Accordingly, we

Table 2: Summary of the kernels used for butterfly counting.

Kernel type	Threshold	Features	Counting method	Shared memory
Thread kernel	$\|N^G_{v\_thread}\|$<16	The number of neighbors of a vertex is very small (<than 16)	list intersection& hashmap	yes
Warp kernel	$\|N^G_{v\_warp}\|$<1024	Adjust parallel section according to N^G_2: N^G_2 > threshold: assign threads to parallel 2-hop neighbors N^G_2 <= threshold: assign threads to parallel neighbors	hashmap	yes
Multi-block kernel	$\|N^G_{v\_multi}\|$>1024	The number of elements in the 2-hop neighbor list is huge	hashmap	no

multiply the number of its neighbors and the average number of vertices of its neighbors' adjacency lists.

The discrepancy between approximate and real numbers of vertices can be mitigated by experimentally choosing appropriate thresholds, as detailed in Section 7.5. Thus, if the estimated number of 2-hop neighbors is large, it is likely that the thread workloads in the block are very imbalanced. We need to minimize this imbalanced circumstance as well. Therefore, when the estimated number of 2-hop neighbors of a vertex is more than a threshold (4096 by default), we let the thread process the 2-hop neighbors of the vertex in parallel when traversing the neighbors. If the number of 2-hop neighbors is smaller than the threshold, we assign different neighbors to the corresponding threads for traversal, as mentioned in Section 4.3.

Thread kernel. The last kernel function we develop is the thread based kernel. This method is generally applied to vertices with a small number of neighbors and 2-hop neighbors, because no matter whether it is a warp-based kernel or a block-based kernel, it can cause a waste of computing resources in this circumstance. Considering the memory consumption discussed in Section 4.3, if we keep a separate hashmap for each thread, we can incur huge memory consumption. Additionally, because the number of 2-hop neighbors of the vertex is small, this causes a huge waste of resources. Because it is rather challenging to accurately estimate the length of the 2-hop neighbors, we load all possible 2-hop neighbors from the global memory. Then, we load the 2-hop neighbors part by part into shared memory. Because of the sparsity of 2-hop vertices, this increases the probability of cache misses and causes waste of computing resources and storage resources. Therefore, we consider another solution to calculate the butterfly counts in parallel. The butterfly counting method we use in this kernel function is to conduct list intersections using the adjacency list and 3-hop neighbors.

5.3 List Intersection-Based BFC

In this part, we further optimize the thread-level kernel by the list intersection.

Analysis. In butterfly counting, we also need to process graphs with very unbalanced vertex degree distributions. For vertices with few neighbors, list intersection is used to save memory usage as well as to accelerate the counting process. BFC traverses the caterpillars of each vertex, as mentioned in Section 2.1, and then conducts list intersection with its neighbors.

The method used here is the binary search. Then, our parallel method is to select the side with more vertices as the start layer, because we allocate threads for each vertex to execute tasks without worrying about memory consumption. This method maximizes the parallel efficiency of GPU, alleviates the estimated number of caterpillars that a vertex lies in and the neighbor lists, and reduces the time for list intersection.

Comparison. Although the hashmap based method has smaller time complexity than the list intersection based method, its memory consumption is unbearable if we keep a separate hashmap for each thread. To better utilize the GPU resources, we use the list intersection based butterfly counting algorithm when the length of the adjacency list of vertices is relatively small. We assign 32 threads to a block. When the vertices' adjacency list is small, we can store all the adjacency lists in shared memory. This largely reduces the memory access cost. In addition, the list intersection time is also bearable because of the small length of the adjacency list. Note that existing GPU-based list-intersection algorithms [13] cannot be used in butterfly counting, even though they have been applied to triangle counting [18]. Because butterfly counting requires traversing 3-hop neighbors, it is not feasible to use these methods for high-degree vertices. We are the first work that innovatively adapts it to the complex situation of butterfly counting.

Algorithm. We provide the algorithm of G-BFC with load balancing optimization in Algorithm 3. At first, we decide which kernel to launch (Line 2). We call the thread kernels separate functions. If the estimated length of the vertices' 2-hop neighbor lists is smaller than the threshold, we use a thread block to parallelize this vertex (Lines 4-5). Otherwise, we assign two blocks to process this vertex. We assign half of the estimated 2-hop neighbors to each block (Line 8), and replace the hashmap in shared memory with a hashmap stored in the global memory (Line 10). If the thread based kernel is launched, we conduct the list intersection based butterfly counting (Line 12). We store the adjacency list in the shared memory (Line 12), which significantly decreases the time required for vertex processing. When making the list intersection, we use the vertex list in shared memory as the search list (Line 13). Hence, we also reduce the time cost for binary search.

Complexity analysis. In Algorithm 3, the outermost for-loop in Line 1 takes $O(S/P)$ times, where P denotes the number of launched cores. Lines 5 to 6 have a complexity of $O(S \cdot deg(v) \cdot deg(w))$, as analyzed in Algorithm 2. From Lines 7 to 10, Algorithm 3 takes $O(deg(u) \cdot deg(v))$ complexity. The for-loop in Line 14 takes $O(deg(u))$, and the binary search function in Line 15 is expected to take $O(\sum_{v,(u,v)\in E} \sum_{w,(w,v)\in E} deg(w) \cdot \log(S))$. As a result, the time complexity for Algorithm 3 is $O(\sum_{u\in S} \min(\frac{S}{M\cdot P} \sum_{(u,v)\in E} deg(v), \sum_{v,(u,v)\in E} \sum_{w,(w,v)\in E} deg(w) \cdot \log(S)))$.

Algorithm 3: G-BFC with load balancing optimization

```
input  : A bipartite graph G
output : B(G)
1  foreach u ∈ S do
2      kernel_type = judge(|N^G(u)|)
3      if kernel_type ≠ thread_based then
4          initialize hash map wedge_num with 0;
5          if |N_2^G(u)| < threshold then
6              run Lines 8 to 13 of Algorithm 2
7          else
8              foreach
                k ∈ [0, |S|/sharedMemorySizeInBlock] do
9                  foreach v ∈ N^G(u) do
10                     foreach w ∈ N^G(v)/2 do
11                         if w.id ∈ global_wedge_num and
                            w.id > u.id then
12                             atomaticadd(global_wedge_num(w))
13     else
14         Memcopy(N^G(u), shared_memory);
15         foreach v ∈ N^G(u) do
16             Binary_search(v, N_3^G(u));
17 return B(G)
```

6 MEMORY AWARE EDGE DIRECTION OPTIMIZATION

We present our memory-aware edge direction optimization in this section.

General design. We develop a memory-aware edge direction optimization, which is tightly coupled to our G-BFC and GPU architecture. Previous edge direction methods, such as [43], only focus on accelerating butterfly counting on CPU, which are not specially designed for GPU memory hierarchy. Different from these existing works, our design has the following distinctions. First, G-BFC adapts to the complicated GPU memory hierarchy. For example, the work [43] also contains an edge direction step but does not need to consider GPU memory consumption. In our work, we accelerate the algorithm on GPU with limited memory. We reconstruct the graph and do not have to maintain the extra priority queue. Second, we design an edge direction strategy to reduce the graph's storage space so that it can fit into the separate GPU global memory. Previous works [38, 43] have overlooked this problem. Third, motivated by the drawbacks, we develop a novel edge direction optimization to limit the number of wedges we need to traverse to further improve the performance of G-BFC. We use a k-core degeneration [22] based method to mark the direction of the edges. Then, we change the adjacency list and narrow the scope of legal wedges.

Detailed design. We first select the vertex with the highest degree, and mark all its edges as out edges. Then, we mark the remaining edges connected to its neighbors as out edges. In other words, our vertex adjacency list stores only the neighbors to which its outgoing edge points. After removing these edges, we continue to look for the vertex with the highest degree, and then mark the direction of the edges just like the previous vertex, until all vertices have been processed. Now, we store the directed graph. The adjacency list of each vertex records only such vertex to which the outgoing edge of the vertex points, and the wedges we traverse must start from a vertex with a higher degree. In other words, for wedge (u_i, v_j, u_k), the highest degree must be u_i, and we do not need to know the degree relationship of v_j and u_k.

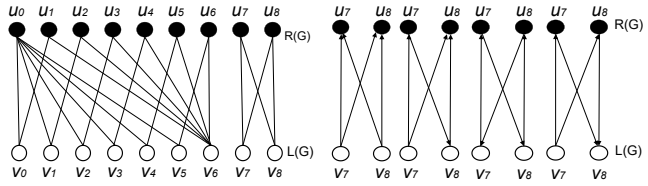
As mentioned in Section 2.1, the butterfly is composed of two wedges. Here, to prevent double calculation of butterfly, we stipulate a path to reach the end vertex. We stipulate that the start vertex is the vertex with the highest degree, and the middle vertex is the vertex that has an undirected edge with the start vertex. Therefore, we specify the edge direction from the start vertex to the middle vertex (i.e., $u_i \Rightarrow v_j$). Similarly, the end vertex is the vertex that has an undirected edge with the middle vertex, and then we set the edge direction as that from the middle vertex to the end vertex. We reduce the number of wedges we need to traverse by specifying the direction of the wedges.

We only reconstruct the adjacency list of the graph. Our original algorithm only needs to traverse the vertices on one side, but now we are based on the edge direction for butterfly counting. We must traverse all vertices. In the following part of this section, we show that the number of wedges that need to be visited is significantly decreased and the time complexity has been reduced accordingly. We also prove that our method counts all the butterflies correctly.

Example. We show an example of how to transform an undirected bipartite graph into a directed bipartite graph by our edge direction optimization. Figure 7 (a) shows an example of a bipartite graph. The degrees of the vertices u_0 and v_6 are both 6, and the degrees of the remaining vertices are all 2. In the original algorithm, regardless of which side we choose to start the butterfly counting from, we still need to traverse 23 wedges in total. Meanwhile, we reduce the wedges to be traversed after directing the edges. As we stated above, we specify that the index of the $L(G)$ vertex is smaller than that of $R(G)$. Although the degrees of u_0 and v_6 are both 6, we start with v_6 to indicate the direction. Then, the edges of v_6, which connect u_1 to u_6, are all outgoing edges of v_6. Then, from u_1 to u_6, the remaining edges of each vertex are marked as their outgoing edges. Accordingly, all the edges of u_0 are marked as outgoing edges. Fortunately, this time we no longer need to traverse the wedges in u_0, because the edges of all vertices from v_0 to v_5 are in-edges, with the adjacency lists of the vertices empty. Next, we mark the edges of v_7. After that, the total number of wedges we need to traverse is 8. This method greatly reduces the number of wedges we need to traverse.

To demonstrate the directed bipartite graph, we select a subgraph of Figure 7 (a) and show its directed results in Figure 7 (b). In Figure 7 (b), the edge direction phase generates only the last output. The reason is that we start from the vertex u_7, and firstly mark all connected edges as out-edges. Accordingly, we have edges of u_7 to v_7, and u_7 to v_8. Next, we check the neighbors of v_7 and v_8 except for u_7. The vertex u_8's degree is equal to that of the start vertex u_7, so we mark the connected edges as v_7 to u_8, and v_8 to u_8. Finally, we obtain the last subgraph in Figure 7 (b).

Algorithm. We provide the complete version of the optimized G-BFC in Algorithm 4. In detail, we first initialize a priority vector (Line 2). After that, we locate the vertex with the maximum degree



(a) Bipartite graph of vertices. (b) Directed bipartite graphs.

Figure 7: Illustration for bipartite graphs.

in S (Line 6), and set it as the highest priority (Line 7). We then remove the vertex from the vertex set (Line 8). Second, we set the priority of the neighbors of the vertex to be lower than that of the vertex, followed by peeling the neighbors off (Lines 9 to 11). Then, we conduct the reconstruction of the graph. Third, we traverse through the edges and mark the edges in accordance with the priority of the source vertex and the destination vertex (Lines 12 to 14). We let the vertex with higher priority be the source vertex and store the destination vertex in its adjacency list. Hence, we reduce the memory consumption of Algorithm 3. After finishing redirecting the edges and reconstructing the graph, we start butterfly counting as Algorithm 3 does.

In our method, we do not need to use extra arrays for butterfly counting, which means that we do not incur additional GPU memory consumption. We even reduce the number of vertices of the adjacency list, because the original undirected graph needs to store the edge information twice.

Complexity analysis. We next analyze the complexity for Algorithm 4. In Line 5, the number of for-loops is smaller than S, where S denotes the number of vertices of the start layer. Because the vertices are sorted in order, the complexity of the *max* function in Line 6 is $O(1)$. The complexities of *set_priority* in Lines 7 and 10, and *delete* in Lines 8 and 11 are both $O(1)$. The complexity of the for-loop in Line 9 is $O(deg(v))$, where $deg()$ denotes the degree of a certain vertex. Based on these analysis, the complexity from Lines 1 to 11 is $O(\sum_{v \in S} deg(v))$. The complexity from Lines 12 to 14 is $O(E)$ where E denotes the number of edges. The complexity of *rebuild* in Line 15 is $O(V)$, because here we only count the length of vertices' adjacency lists. We use O(ALG) to denote the time complexity for Algorithm 3. Therefore, the complexity of Algorithm 4 is O(S·$deg(v)$+E+V+ALG).

Algorithm 4: G-BFC with edge direction optimization

```
   input  : A bipartite graph G
   output : B(G)
1  B(G) ← 0
2  S ← V(G)
3  E ← E(G)
4  p = priority_list;
5  foreach i ∈ range(S) do
6      v = max(S, adjacency_list_length);
7      set_priority(v);
8      S.delete(v);
9      foreach u ∈ N^G(v) do
10         set_priority(u);
11         s.delete(v);
12 foreach e ∈ E do
13     if p(e.src) < p(e.dst) then
14         swap(e.src, e.dst);
15 rebuild graph;
16 run Algorithm 3;
17 return B(G)
```

7 EVALUATION

In this section, we evaluate the performance of G-BFC, and compare it to the cutting-edge butterfly counting method.

7.1 Experimental Setup

Methodology. We compare our GPU-based butterfly counting, denoted as "G-BFC", with three existing methods. The first method is the vertex priority based butterfly counting [43], which is the state-of-the-art butterfly counting, denoted as "BFC-VP++". The second method is the parallel list intersection based butterfly counting on CPU [35], denoted as "BFC-WE-CPU". The third method is the GPU version of list intersection based butterfly counting, denoted as "BFC-WE-GPU", which has the same kernel function as we have in the thread kernel in the load balancing optimization of G-BFC. BFS-WE-GPU has the same start layer as that of the basic G-BFC (Algorithm 2), and stores the start layer in shared memory. Then, BFS-WE-GPU traverses the 3-hop neighbors of vertices in the start layer and uses binary search to conduct the list intersection. Additionally, we evaluate G-BFC without using shared memory, denoted as "G-BFC (w/o sharedMem)", G-BFC without load balancing optimization, denoted as "G-BFC (w/o loadBalance)", and G-BFC without edge direction optimization, denoted as "G-BFC (w/o edgeDirect)".

Platform. We conduct the evaluation on a platform equipped with an Intel Core i9-10900X CPU, and an Nvidia Geforce RTX 3090 GPU. The CPU has 10 cores, which can provide 1,504 GFLOPS. The GPU has 10,496 light-weight cores, which can provide 35.6 TFLOPS. The platform has 128GB memory and the operating system is Ubuntu 20.04.01.

BFC supported graph algorithms. As discussed in Section 3, BFC has been used as an important primitive in many graph algorithms. We apply G-BFC to three graph algorithms, k-wing [36], clustering coefficient [53], and graph cohesion [24], to validate its end-to-end acceleration effectiveness.

Datasets. We employ eleven real datasets that have been widely used in prior studies [5, 18, 38, 43, 45] in our evaluation. The detailed descriptions of the datasets are shown in Table 3, including Wikilens, Amazon (Wang), Sexual escorts, Trip Advisor, BookCrossing (Ratings), Stack Overflow, Yahoo songs, Amazon Ratings, Epinions, Tracking Network, and LiveJournal [2], which can be obtained from KONECT [3]. In Table 3, R and L are different layers in the bipartite graph. $|R|$ represents the total number of vertices in the R layer. $|L|$ represents the total number of vertices in the L layer. $|E|$ represents the total number of edges in the graph. To illustrate the difference in the distribution of vertex degrees, we also list the maximum vertex degree in R and L.

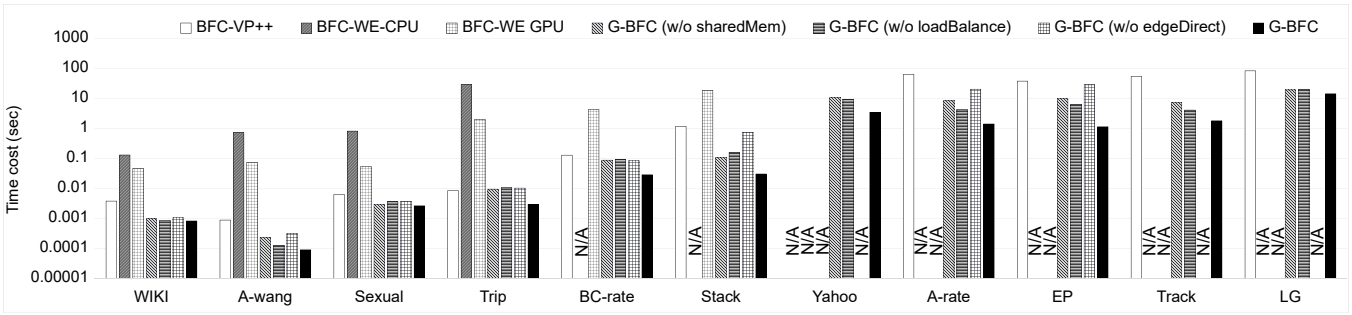


Figure 8: Performance on different datasets.

Table 3: Summary of the datasets. BC# represents the number of butterflies in the graph. MD(L/R) represents MaxDegree(L)/MaxDegree(R).

Dataset	|L|	|R|	|E|	BC #	$MD(L/R)$
Wikilens (WIKI)	0.3K	5K	27K	6M	1.7K/80
Amazon (A-wang)	0.8K	26K	29K	3.6K	0.8K/44
Sexual escorts (Sexual)	6.6K	10K	51K	0.2M	0.1K/0.6K
Trip Advisor (Trip)	1.8K	0.1M	0.18M	11K	2K/22
BookCrossing(Ratings) (BC-rate)	78K	0.2M	0.4M	1.5M	8K/0.7K
Stack Overflow (Stack)	97K	0.5M	1.3M	18.3M	6K/5K
Yahoo songs (Yahoo)	0.6M	1M	259M	582M	468M/307M
Amazon Ratings (A-rate)	1.2M	2.1M	5.8M	35.8M	3K/12K
Epinions (EP)	0.1M	0.8M	13.7M	170B	162M/1K
Tracking Network (Track)	0.4K	18M	37M	600B	11B/346
LiveJournal (LG)	3.2M	7.5M	112M	$7 \cdot 10^{10}B$	0.3K/1M

7.2 BFC Performance

We show the execution time of different methods in Figure 8. If the execution time of a program exceeds 10 hours, we shall terminate the procedure (denoted as N/A in the measurements). We can see that our solution, G-BFC, has the lowest time cost. We have the following observations.

First, G-BFC achieves the highest performance in all cases. On average, G-BFC achieves 19.8× speedup over BFC-VP++, 4507.6× speedup over BFC-WE-CPU, and 308.8× speedup over BFC-WE-GPU. Although BFC-VP++ and BFC-WE-CPU are parallel in CPU, their performance is limited by the CPU capacity. The reason for the high performance of G-BFC lies in the full utilization of the GPU computing capability. G-BFC provides carefully designed memory access patterns, such as the lock-free counting strategy, as well as novel optimizations including load balancing and edge direction.

Second, all the three optimizations of shared memory utilization, load balancing, and edge direction considerably enhance the performance of G-BFC. On average, the shared memory optimization, mentioned in Section 4, brings about 38% performance improvement. The load balancing optimization, mentioned in Section 5, brings 26% performance improvement. The edge direction optimization, mentioned in Section 6, brings nearly 95% performance improvement. Edge direction optimization brings the greatest performance improvement, which implies that reducing the number of wedges to be traversed introduces great benefits to butterfly counting.

Third, G-BFC exhibits different performance behaviors on various datasets. Compared to BFC-VP++, G-BFC achieves 44× performance speedup on dataset *A-rate*. However, on dataset WIKI, the performance speedup of G-BFC is only 4.5×. The reason is that for small datasets, the compute capacity of CPU is sufficient for handling the workload, and processing directly on CPU does not require the data transmission between CPU and GPU. However, for large datasets, the benefits of massive parallelism on GPU can amortize the overhead of the memory transition. Besides, n_v and n_e in *A-rate* are much larger than those of *WIKI*, and the CPU cannot provide sufficient computing power to process all these vertices and edges efficiently.

7.3 BFC Supported Graph Algorithms

As discussed in Section 2.2, k-wing [36], clustering coefficient [53], and graph cohesion [24] have been used as key graph algorithms in many applications. Table 4 shows the performance improvement of G-BFC for k-wing, clustering coefficient, and graph cohesion. We can see from Table 4 that BFC occupies an average of 41% of the total time. For clustering coefficient, it can occupy more than 80% of the time. For k-wing and graph cohesion, BFC also accounts for the majority of the time, which is consistent with the study [36]. With our solution of G-BFC, we save 38% execution time on average, and thus BFC is no longer a performance bottleneck in these algorithms.

Table 4: Speedup of G-BFC over CPU-based BFC and time contribution of BFC in different algorithms.

Dataset	Clustering coefficient			Graph cohesion			K-wing decomposition		
	Speedup	BFC	G-BFC	Speedup	BFC	G-BFC	Speedup	BFC	G-BFC
WIKI	2.23	58.37%	2.99%	1.88	49.01%	2.10%	1.43	52.19%	22.30%
A-wang	3.36	75.41%	3.89%	1.68	42.02%	0.91%	1.68	42.30%	1.11%
Sexual	3.73	74.18%	0.87%	1.78	44.14%	0.24%	4.36	78.08%	1.30%
Trip	2.75	65.10%	1.06%	2.31	58.23%	1.11%	1.33	25.09%	0.22%
BC-rate	4.64	82.12%	2.88%	3.03	69.18%	2.01%	1.59	38.23%	0.43%
Stack	2.05	86.01%	35.19%	1.66	47.20%	6.89%	1.58	43.91%	6.21%
A-rate	5.14	84.23%	3.70%	1.97	56.11%	7.33%	2.39	71.29%	13.09%
EP	2.45	61.12%	1.70%	3.39	72.02%	2.06%	3.55	73.98%	2.18%
Track	7.11	85.68%	0.32%	3.04	70.03%	3.32%	2.07	52.85%	1.20%
Yahoo	1.82	51.11%	6.09%	1.43	30.03%	0.12%	1.40	29.21%	0.02%
LG	2.53	61.59%	2.99%	1.64	40.41%	1.05%	1.60	39.09%	0.83%

7.4 Performance Profiling

Time breakdown. We compare BFC-VP++ and G-BFC, and show their detailed time breakdown in Table 5. The total time for G-BFC we demonstrate in Table 5 includes the BFC time and the edge direction time. The BFC time represents the accelerated butterfly counting running time. For small datasets including WIKI, A-wang,

Sexual, Trip, BC-rate, and Stack, we do not apply the edge direction stage for both BFC-VP++ and G-BFC, because their edge direction time significantly exceeds their BFC time, and the speedup is relatively small.

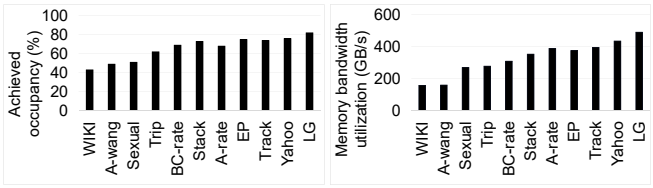
Table 5: Time breakdown of different methods. "N/A" means that the method is not applicable. "OoT" means timeout.

Dataset	BFC-VP++ (ms)		G-BFC (ms)			Speedup	
	kernel	total	edge direction	kernel	total	kernel	total
WIKI	4.37	4.37	N/A	1.08	1.08	4.05	4.05
A-wang	1.95	1.95	N/A	0.32	0.32	6.09	6.09
Sexual	17.78	17.78	N/A	3.74	3.74	4.75	4.75
Trip	107.13	107.13	N/A	10.29	10.29	10.41	10.41
BC-rate	328.83	328.83	N/A	85.23	85.23	3.86	3.86
Stack	2040	2040	N/A	738.26	738.26	2.76	2.76
Yahoo	OoT	OoT	1210	3400	4610	N/A	N/A
A-rate	62846	66580	13600	14000	27060	4.49	2.41
EP	37498	40760	4115	1136	5250	33.01	7.76
Track	53611	87740	6392	1788	8180	29.98	10.73
LG	OoT	OoT	67809	30425	98234	N/A	N/A

We have the following observations. First, in most cases, G-BFC achieves benefits in BFC time. On average, G-BFC achieves 11.8× speedup in BFC time and 63% time saving in total time. Second, G-BFC brings significant performance benefits in the preprocessing stage for large datasets. The reason is that as the size of bipartite graphs increases, more wedges are required to be traversed. Accordingly, preprocessing reduces a large proportion of the unnecessary wedges, thus improving the performance. Third, it can be seen that our algorithm indeed has large performance benefits compared with the state-of-the-art BFC-VP++, and our optimization reduces the time of the butterfly counting, bringing large savings in BFC time, especially for large datasets. In addition, edge direction is extremely beneficial for graphs that are used multiple times. For example, the large-scale graph benchmark, Graph500 [27], serves as a guide for many HPC designs and allows preprocessing. Edge direction can be applied to many downstream applications that are based on butterfly counting, such as spam detection [14] and recommendation algorithms [40]. In our application scenarios, the bipartite graphs only need to be preprocessed once and then used multiple times. Hence, the edge direction overhead can be amortized.

Occupancy. The achieved occupancy is the average active warp ratio. To assess the achieved occupancy of G-BFC, we use the Nvidia performance measurement tool *nvprof*. We show the occupancy of our platform in Figure 9 (a). G-BFC achieves high occupancy in most datasets. On average, G-BFC achieves 65.6% occupancy. For large datasets such as Yahoo and LG, the achieved occupancy can reach 81%.

Memory bandwidth. DRAM throughput represents the sum of read and write bandwidth utilization of GPU memory. We use *nvprof* to analyze the achieved memory bandwidth, as shown in Figure 9 (b). G-BFC achieves high memory bandwidth utilization on datasets with millions of vertices, but on small datasets like WIKI and A-wang, the achieved bandwidth utilization is moderate. The reason is that there still exists dependencies in butterfly counting, and the amount of data is too small for GPU capacity to be fully utilized.



(a) Occupancy. (b) Memory bandwidth.

Figure 9: Performance profiling.

7.5 Threshold Selection

To demonstrate the threshold selection, we select three representative datasets for description ("Stack", "Trip", and "EP"). We refer the parameters of $N^G_{v\_thread}$, $N^G_{v\_warp}$, and N^G_2, mentioned in Table 2, as *threshold*1, *threshold*2, and *threshold*3, respectively, and we obtain the optimal result when *threshold*1, *threshold*2, and *threshold*3 are set to be 16, 1024, and 4096. The search space for *threshold*1 is {16,32,64,128}, for *threshold*2 is {512, 1024,2048,4096}, and for *threshold*3 is {1024,2048,4096,6144}. Due to space limitation, we describe the results in brief. We have explored different combinations of thresholds, and the combination of thresholds that we choose achieves the optimal performance. The selected combination achieves 738 ms in *Stack*, 10.29 ms in *Trip*, and 1136 ms in *EP*, which can bring 7% performance improvement on average compared with other combinations. The reason is that with the increasing degree of vertices, due to limited memory resources, the thread-level kernel does not have sufficient degree of parallelism as the block-level kernel, and the extra time cost exceeds the benefits gained. For similar reasons, when *threshold*2 is too large, the block-level kernel can also be exhausted in processing the vertices. For *threshold*3, G-BFC achieves the optimal performance when *threshold*3 is set to 4096.

8 CONCLUSION

In this paper, we develop a novel GPU-based butterfly counting, called G-BFC, which enables efficient butterfly counting on GPU. G-BFC can make full use of GPU resources and reduce the overhead of memory accesses. In addition, we provide optimizations regarding the problem of butterfly counting and GPU characteristics. We tackle the load imbalance issue and reduce the number of wedges G-BFC needs to traverse by preprocessing, which successfully improves the efficiency of G-BFC. Experimental results demonstrate the effectiveness of performing butterfly counting on GPU and show that our solution significantly outperforms the state-of-the-art butterfly counting algorithm.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004401), and National Natural Science Foundation of China (61732014, 62172419, and 62072458). This work is also sponsored by CCF-Tencent Open Research Fund. Bingsheng's work is in part supported by a Singapore MoE Tier 2 grant (T2EP20121-0030). The research of Dong Deng was supported by NSF grants #2152908 and #2212629. Q. Xu, F. Zhang, Z. Yao, L. Lu, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and the School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] 2021. https://www.taobao.com/about/intro.php
[2] 2021. https://ssc.io/trackingthetrackers/
[3] 2021. http://konect.cc
[4] Sinan G Aksoy, Tamara G Kolda, and Ali Pinar. 2017. Measuring and modeling bipartite graphs with community structure. *Journal of Complex Networks* 5, 4 (2017), 581–603.
[5] A. Azad, A. Buluc, and J. Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 804–811.
[6] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*. 119–130.
[7] Mauro Bisson and Massimiliano Fatica. 2017. High performance exact triangle counting on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510.
[8] Guido Caldarelli, Romualdo Pastor-Satorras, and Alessandro Vespignani. 2004. Structure of cycles and local ordering in complex networks. *The European Physical Journal B* 38, 2 (2004), 183–186.
[9] Jie Chen and Yousef Saad. 2010. Dense subgraph extraction with application to community detection. *IEEE Transactions on knowledge and data engineering* 24, 7 (2010), 1216–1230.
[10] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on cpu and GPU. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1190–1205.
[11] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008).
[12] Hossam Faris, Al-Zoubi Ala M, Ali Asghar Heidari, Ibrahim Aljarah, Majdi Mafarja, Mohammad A Hassonah, and Hamido Fujita. 2019. An intelligent system for spam detection and identification of the most relevant features based on evolutionary random weight networks. *Information Fusion* 48 (2019), 67–83.
[13] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A Bader. 2018. Fast and adaptive list intersections on the GPU. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.
[14] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*. Citeseer, 721–732.
[15] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.
[16] O. Green, P. Yalamanchili, and L. M. Munguia. 2014. Fast triangle counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.
[17] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *Proceedings of the 2021 International Conference on Management of Data*. 736–748.
[18] Yang Hu, Hang Liu, and H Howie Huang. 2018. Tricore: Parallel triangle counting on GPUs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 171–182.
[19] Zan Huang. 2010. Link prediction based on graph topology: The predictive value of generalized clustering coefficient. *Available at SSRN 1634014* (2010).
[20] Shweta Jain and C Seshadhri. 2017. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th international conference on world wide web*. 441–449.
[21] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2020. NextDoor: GPU-Based Graph Sampling for Graph Machine Learning. *arXiv preprint arXiv:2009.06693* (2020).
[22] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-core decomposition of large networks on a single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.
[23] Myunghwan Kim and Jure Leskovec. 2012. Multiplicative attribute graph model of real-world networks. *Internet mathematics* 8, 1-2 (2012), 113–160.
[24] Pedro G Lind, Marta C Gonzalez, and Hans J Herrmann. 2005. Cycles and clustering in bipartite networks. *Physical review E* 72, 5 (2005), 056127.
[25] Shih-Hsiang Lo, Che-Rung Lee, Yeh-Ching Chung, and I-Hsin Chung. 2011. A parallel rectangle intersection algorithm on GPU+ CPU. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 43–52.
[26] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *PVLDB* (2020).
[27] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.
[28] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.
[29] Donald Palmer. 1983. Broken ties: Interlocking directorates and intercorporate coordination. *Administrative Science Quarterly* (1983), 40–55.
[30] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring Data Analytics without Decompression on Embedded GPU Systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).
[31] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[32] Ali Pinar, C Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*. 1431–1440.
[33] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. 2012. Graft: An approximate graphlet counting algorithm for large graph analysis. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 1467–1471.
[34] Garry Robins and Malcolm Alexander. 2004. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Computational & Mathematical Organization Theory* 10, 1 (2004), 69–94.
[35] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *ACM SIGKDD*. 2150–2159.
[36] A. E. Sariyuce and A. Pinar. 2018. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 504–512.
[37] SheshboloukiAida and Z. Tamer. 2022. sGrapp: Butterfly Approximation in Streaming Graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 16, 4 (2022), 1–43.
[38] Jessica Shi and Julian Shun. 2020. Parallel algorithms for butterfly computations. In *Symposium on Algorithmic Principles of Computer Systems*. SIAM, 16–30.
[39] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: high performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
[40] Xiaoyuan Su and Taghi M Khoshgoftaar. 2009. A survey of collaborative filtering techniques. *Advances in artificial intelligence* 2009 (2009).
[41] Koji Ueno and Toyotaro Suzumura. 2013. Parallel distributed breadth first search on GPU. In *20th Annual International Conference on High Performance Computing*. IEEE, 314–323.
[42] Jia Wang, Ada Wai-Chee Fu, and James Cheng. 2014. Rectangle counting in large bipartite graphs. In *2014 IEEE International Congress on Big Data*. IEEE, 17–24.
[43] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex Priority Based Butterfly Counting for Large-scale Bipartite Networks. *PVLDB* (2019).
[44] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.
[45] Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. 2017. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.
[46] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient pagerank and spmv computation on amd GPUs. In *ICPP*. IEEE, 81–89.
[47] Feng Zhang, Zheng Chen, Chenyang Zhang, Amelie Chi Zhou, Jidong Zhai, and Xiaoyong Du. 2021. An efficient parallel secure machine learning framework on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2021).
[48] Feng Zhang, Zaifeng Pan, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. G-TADOC: Enabling Efficient GPU-Based Text Analytics without Decompression. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*.
[49] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2016. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2016), 905–918.
[50] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535.
[51] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: A High-Performance Framework for Enabling Near Orthogonal Processing on Compression. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2022), 459–475.
[52] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.
[53] Peng Zhang, Jinliang Wang, Xiaojia Li, Menghui Li, Zengru Di, and Ying Fan. 2008. Clustering coefficient and community structure of bipartite networks. *Physica A: Statistical Mechanics and its Applications* 387, 27 (2008), 6869–6875.
[54] Rong Zhu, Zhaonian Zou, and Jianzhong Li. 2018. Fast rectangle counting on massive networks. In *ICDM*. IEEE, 847–856.