Accelerating Biclique Counting on GPU

Linshan Qiu, Zhonggen Li, Xiangyu Ke, Lu Chen, Yunjun Gao

Zhejiang University, Hangzhou, China

{*lsqiu, zgli,xiangyu.ke,luchen, gaoyj*}@*zju.edu.cn*

Abstract—Counting (p, q)-bicliques in bipartite graphs poses a foundational challenge with broad applications, from densest subgraph discovery in algorithmic research to personalized content recommendation in practical scenarios. Despite its significance, current leading (p, q)-biclique counting algorithms fall short, particularly when faced with larger graph sizes and clique scales. Fortunately, the problem's inherent structure, allowing for the independent counting of each biclique starting from every vertex, combined with a substantial set intersections, makes it highly amenable to parallelization. Recent successes in GPU-accelerated algorithms across various domains motivate our exploration into harnessing the parallelism power of GPUs to efficiently address the (p, q)-biclique counting challenge.

We introduce GBC (GPU-based Biclique Counting), a novel approach designed to enable efficient and scalable (p, q)-biclique counting on GPUs. To address major bottleneck arising from redundant comparisons in set intersections (occupying an average of 90% of the runtime), we introduce a novel data structure that hashes adjacency lists into truncated bitmaps to enable efficient set intersection on GPUs via bit-wise AND operations. Our innovative hybrid DFS-BFS exploration strategy further enhances thread utilization and effectively manages memory constraints. A composite load balancing strategy, integrating pre-runtime and runtime workload allocation, ensures equitable distribution among threads. Additionally, we employ vertex reordering and graph partitioning strategies for improved compactness and scalability. Experimental evaluations on eight real-life and two synthetic datasets demonstrate that GBC outperforms state-of-the-art algorithms by a substantial margin. In particular, GBC achieves an average speedup of 497.8×, with the largest instance achieving a remarkable 1217.7× speedup when $p = q = 8$.

Index Terms—biclique counting, bipartite graph, GPU

I. INTRODUCTION

Bipartite graphs are pivotal for illustrating connections between two distinct sets of entities, finding practical applications across diverse domains such as social networks [49], recommender systems [13], [32], and e-commerce networks [26]. In this model, one set signifies a specific type of entity (e.g., users), while the other set represents a different type (e.g., items). Edges establish connections from entities in the first set only to those in the second, capturing relationships or interactions between them. There has been a wide range of explorations over bipartite graphs, such as community search [49], cohesive subgraph discovery [28], [48], butterfly counting [38], and graph learning [26], [55]. However, within this intricate landscape, a critical challenge emerges—the enumeration of (p, q)-bicliques, a combinatorial task with far-reaching implications. A (p, q)-biclique represents a complete subgraph with p vertices from one set and q vertices from the other. Notably, the well-known butterfly concept corresponds to the $(2, 2)$-biclique [54]. The enumeration of (p, q)-bicliques
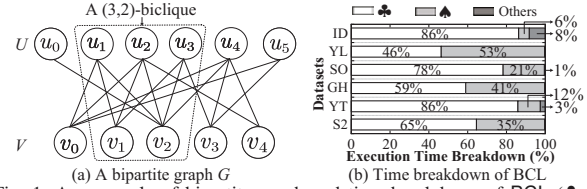

Fig. 1: An example of bipartite graph and time breakdown of BCL (♣ and ♠ denotes searching for shared 1-hop and 2-hop neighbors, respectively).

holds crucial importance in both algorithmic research and various applications, including densest subgraph detection [34], cohesive subgroup analysis [10] in bipartite graphs, and optimization of GNN information aggregation [54]. For instance, the pursuit of the (p, q)-biclique densest subgraph in a bipartite graph relies on the concept of (p, q)-biclique density [34], which is defined as the ratio between the number of (p, q)-bicliques in a subgraph S and the size of S.

Counting (p, q)-bicliques presents a formidable challenge, given its exponential increase concerning p and q [54]. Specifically, retrieving (p, q)-bicliques involves iteratively identifying the common 1-hop neighbors for a set of vertices that are mutually 2-hop neighbors[1], requiring intersections between the adjacency lists of the vertices under exploration and those in the partial results. Yang et al. undertake the pioneering investigation and introduce the leading algorithm in a backtracking manner, namely BCL [54]. However, BCL encounters scalability issues[2] concerning either dataset size or clique scale, i.e., larger p and q. We observe that the inefficiency in identifying shared 1-hop and 2-hop neighbors via intersections is the primary culprit. Figure 1(b) visually breaks down the execution time taken by BCL across six selected datasets, with labels omitted for clarity when bar heights fall below 1%. As depicted, the time devoted to searching for shared 1-hop and 2-hop neighbors peaks at more than 99%, averaging at 97%. Consequently, there is an urgent need to optimize the intersection to enhance the algorithm's efficiency.

Example 1. *Consider the identification of $(3, 2)$-bicliques in Figure 1(a), the dotted line circle presents an instance with vertices $\{u_1, u_2, u_3\}$ from layer U and $\{v_1, v_2\}$ from layer V. Notably, $\{u_2, u_3, u_4\}$ are also mutual 2-hop neighbors. Specifically, u_2 and u_3 share 1-hop neighbors $\{v_1, v_2\}$, u_2 and u_4 have 1-hop neighbors $\{v_0, v_2, v_4\}$, and u_3 and u_4 possess 1-hop neighbors $\{v_2, v_3\}$. However, despite these mutual 2-hop connections, they only share a common 1-hop neighbor, v_2, preventing them from forming a $(3, 2)$-biclique. The set of*

---

[1] A pair of vertices share at least p or q neighbors on the other side.

[2] On real-world million-scale dataset *FR*, the running time of BCL exceeds 24 hours when $p = q = 8$. Please refer to § VII-B for more details.

vertices with a mutual 2-hop neighborhood relationship serves as candidates, and the fundamental test involves checking their common neighbors, leading to exponential growth in computational costs concerning the target clique size (p, q).

As real-world graphs undergo exponential growth, their computational demands have intensified. A recent and notable trend involves the utilization of GPUs to streamline a variety of graph algorithms, including shortest path [30], PageRank [42], breadth-first traversal [29], graph pattern mining [16], [50], leading to remarkable performance advancements. Compared to CPUs, GPUs stand out with numerous computation cores and high-bandwidth memory, making them ideal for computationally intensive tasks. Encouraged by the recent success in GPU-accelerated graph algorithms, GPU-based parallelism emerges as a promising solution to expedite (p, q)-clique counting. The independence of counting (p, q)-bicliques from each vertex presents significant potential for parallelizing (p, q)-biclique counting. Furthermore, implementing the algorithm's predominant procedures through intersections, feasibility proven on GPUs [21], [53], enhances its efficiency.

Challenges. GPUs operate in an SIMT (Single Instruction, Multiple Threads) manner, which is distinct from the typical CPU paradigm. A simple transposition of the existing algorithm to the GPU, as evidenced in §VII, does not yield satisfactory performance improvements. Specifically, three primary challenges are hindering the development of a high-performance (p, q)-biclique counting algorithm on the GPU.

Challenge I: How to implement efficient set intersection on GPU? The inefficiency of set intersection in existing GPU-based algorithms arises from redundant comparisons and substantial memory access [35]. Various efforts within the field of triangle counting [8], [21], [22] have sought to optimize set intersection on GPUs, with the binary search-based approach [21] emerging as a leading efficient technique. Performing binary searches on adjacency lists involves element-wise comparisons with elements residing in global memory, increasing computational overhead and memory access latency—particularly when dealing with lengthy adjacency lists (larger graphs) and probing deeper search trees (larger clique size). To address these challenges, we introduce a novel data structure, Hierarchical Truncated Bitmap, which hashes vertices in the adjacency list into 32-bit truncated bitmaps (integers) and aggregates the offsets of these bitmaps as a range index. This enhancement facilitates set intersection through bit-wise AND operations, mitigating redundant comparisons (§ V-A). Furthermore, we propose a vertex reordering method named Border, aiming to maximize the storage of multiple vertices within a single bitmap, thus effectively compressing the data and reducing data retrieval overhead (§ V-B).

Challenge II: How to adapt the biclique counting algorithm to align with GPU architecture? The GPU, with its abundance of available threads numbering in the thousands, demands optimal utilization to fully harness its computational prowess. However, several issues contribute to low GPU thread utilization. Firstly, as exploration depth increases, the elements pending examination in the partial result sets progressively

diminish. Using a fixed-size thread group leads to a significant portion of threads being idle. Additionally, traversing vertices through backtracking exploration one at a time results in thread and bandwidth wastage. To mitigate these concerns, we diverse a hybrid DFS-BFS search strategy to adaptively unify the tasks of multiple vertices for enhanced parallelism and thread utilization (§ IV). Secondly, the skewed distribution of vertex degrees and the unpredictable workload during depth-first traversal lead to a marked imbalance during runtime. Consequently, we employ a combination of pre-runtime and runtime task distribution strategies to achieve equilibrium in the distribution of workload among threads (§ V-C).

Challenge III: How to improve the scalability for large graphs? The limited memory capacity of GPUs presents a challenge in accommodating large-scale graphs within device memory, typically ranging from a few to tens of gigabytes. Real-world graphs, however, expand exponentially, posing difficulties for complete residence in GPU memory [19]. Various efforts have addressed limited device memory through graph partitioning [19], [35]. Yet, due to partition interdependence, frequent loading and eviction may occur, resulting in significant transmission overhead. Moreover, a substantial portion of imported data remains unutilized. This inefficiency, coupled with computation cores waiting for required data transfers, impedes throughput. Leveraging the fact that clique computations involve interactions with, at most, 2-hop neighbors, we partition the graph into disjoint and autonomous subgraphs. This approach allows computations for any given vertex to occur within the confines of the current subgraph, eliminating the need for on-demand data loading (§ VI).

Contributions. In this work, we introduce GBC (GPU-based Biclique Counting), the pioneering work facilitating (p, q)-biclique counting by harnessing the massive parallelism of GPUs. We outline the key advancements as follows:

- We devise a novel *data structure* and *vertex reordering* technique to implement highly efficient intersection computation on GPUs.
- We advocate the adoption of a *hybrid search* strategy to optimize thread utilization. Additionally, our *joint load-balancing* strategy optimizes pre-runtime and runtime workload allocation for improved efficiency.
- We propose a *communication-free partitioning* method, enabling the computation of (p, q)-bicliques on large-scale graphs using GPUs.
- We conduct *comprehensive experiments* across ten diverse datasets to demonstrate the superior performance of the proposed approach. The results showcase an average speedup of over $400\times$ compared to existing baselines.

Roadmap. We formally define the problems in § II, followed by a brief introduction to the leading CPU-based solution and our basic implementation on GPUs in § III. § IV illustrates the search paradigm, and § V presents various optimization techniques, including data structure, vertex reordering, and load balancing. Scalability consideration is addressed in § VI. Experiments are presented in § VII. We review related works in § VIII and conclude the paper in § IX.

II. Preliminaries

In this section, we provide the formal problem definition and give a brief introduction to GPUs. The frequently used notations are summarized in Table I.

A. Problem Definition

Given an unweighted and undirected bipartite graph $G = (U, V, E)$, $U(G)$ and $V(G)$ denote two disjoint sets of vertices on the upper and lower layers, respectively (i.e., $U(G) \cap V(G) = \emptyset$). $E(G) \subseteq U(G) \times V(G)$ represents the edge set of G, where a edge (u, v) can only exist between $u \in U(G)$ and $v \in V(G)$. We use $N(u, G) = \{v | (u, v) \in E(G)\}$ and $N_2(u, G) = \{u' | (u, v) \in E(G) \land (u', v) \in E(G)\}$ to denote the 1-hop neighbors (vertices directly connected to u) and 2-hop neighbors (vertices indirectly connected to u via an intermediate vertex) of vertex u in G, respectively. Additionally, we extend this notion to encompass the 2-hop neighbors sharing at least k ($k = p$ or q in our problem) common 1-hop neighbors with u, i.e., $N_2^k(u, G) = \{u' | u' \in U \cup V \text{ and } |N(u, G) \cap N(u', G)| \geq k\}$. By default, when referring to 2-hop neighbors, we specifically indicate $N_2^k(u, G)$, particularly during intersection operations. The degree of u is defined as $d(u, G) = |N(u, G)|$. For simplicity, the notation omits G when the context is self-evident.

Definition 1 ((p,q)-Biclique) [54]. *Given a bipartite graph $G = (U, V, E)$, a **biclique** $B(L, R)$ is a complete bipartite subgraph of G, where $L \subseteq U(G)$, $R \subseteq V(G)$, and $\forall (u, v) \in L \times R$, $(u, v) \in E(G)$. A biclique $B(L, R)$ is a **(p,q)-biclique** if it satisfies $|L| = p$ and $|R| = q$, where p, q are two integers.*

Problem Statement. Given a bipartite graph G and two integers p, q, the problem of **biclique counting** aims to determine the cardinality of (p, q)-bicliques of G.

B. GPU Architecture

GPU is a high-performance hardware with tens of streaming multiprocessors (SMs), each functioning as an autonomous processing unit equipped with hundreds of cores. The CUDA (Compute Unified Device Architecture) programming model organizes 32 threads into a warp, where threads execute in a Single Instruction Multiple Thread (SIMT) manner. Hence, program branches would lead to thread divergence and degenerate performance. In CUDA, a block consists of multiple warps and is assigned to an SM. All blocks collectively form a grid with all GPU threads.

GPU typically has multiple levels of memory hierarchy. Global memory, shared among all threads, is the largest GPU memory reservoir. Reading (writing) data from (to) it incurs significant latency. In CUDA, all threads within a warp access the consecutive addresses in the global memory, which is known as coalesced memory access. Therefore, accessing non-consecutive memory for threads in the same warp necessitates multiple memory transactions, leading to low bandwidth. Each SM has a fast, small shared memory (typically 16KB to 64KB per SM) exclusively accessible by threads within a block.

TABLE I: Symbols and descriptions.

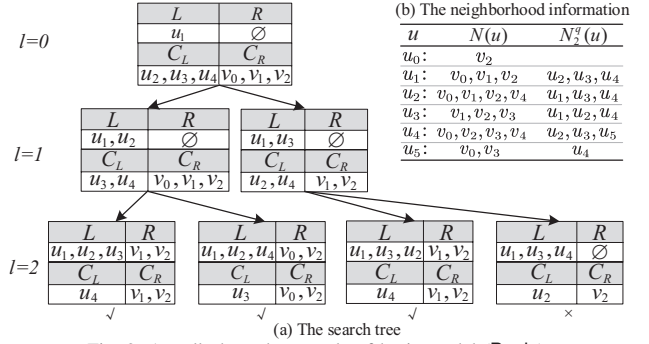| Symbols	Description
G	the bipartite graph
$U(G)(V(G))$, $E(G)$	the upper (lower) layer vertex set and the edge set of G
$N(u, G)$, $N_2(u, G)$, $N_2^k(u, G)$	1-hop neighbors, 2-hop neighbors of u, and 2-hop neighbors sharing at least k 1-hop neighbors with u
$B(L, R)$	the biclique B with vertex sets L and R
p, q	the sizes of L and R, respectively
C_L, C_R	the candidate sets for L and R, respectively

Fig. 2: A walk-through example of basic model (Basic).

III. Base Model & GPU Baseline

In this section, we first discuss the basic model and leading solutions on CPU, followed by our baseline design on GPU. Then we give an overview of our proposed methods.

A. Basic Model for Biclique Counting

The basic model (referred to Basic) for biclique counting first selects one side of the vertices as the start vertices (say U)[3]. Subsequently, it assembles the 2-hop neighbors of these vertices that share at least q 1-hop neighbors, constituting the candidate set for this layer (denoted as C_L). Through iterative procedures, candidates in C_L are incrementally added to L to expand the partial result on layer U, and C_L is updated by intersecting with the 2-hop neighbors of vertices in L. Simultaneously, the candidate set C_R undergoes updates by intersecting with 1-hop neighbors of vertices in L. After iterating the above steps p times, we form a (p, q)-biclique $B(L, R)$ by selecting q vertices (if available) from C_R to R and combining them with L. Otherwise, Basic backtracks and explores adding other candidates to form a (p, q)-biclique. The following example illustrates the workflow of Basic.

Example 2. *Given G in Figure 1(a), and two parameters $p=3, q=2$, the workflow of the Basic to retrieve $(3, 2)$-biclique starting from u_1 is presented in Figure 2(a). The neighborhood information of G is summarized in Figure 2(a). At level 0, Basic initializes L by adding u_1 and establishes C_L, C_R as $N_2^q(u_1)$ and $N(u_1)$, correspondingly. With the inclusion of u_2 in L, Basic traverses the left branch from level 0 to level 1. Subsequently, Basic updates C_L and C_R. Specifically, we have $C_L = C_L \cap N_2^q(u_2) = \{u_3, u_4\}$ and $C_R = C_R \cap N(u_2) = \{v_0, v_1, v_2\}$. Basic advances*

---

[3]Without loss of generality, assuming Basic consistently opts for layer U as the starting layer.

to the leftmost leaf at level 2 by appending u_3 to L and updating C_L and C_R. Consequently, a $(3,2)$-biclique, i.e., $(\{u_1, u_2, u_3\}, \{v_1, v_2\})$ is found by selecting 2 vertices from C_R to R in conjunction with L. Similarly, another two $(3,2)$-bicliques can be identified, namely $(\{u_1, u_2, u_4\}$ and $\{v_0, v_2\})$ and $(\{u_1, u_3, u_2\}, \{v_1, v_2\})$ (a duplicate).

State of the art solution on CPU. Building upon Basic, BCL utilizes the degree information of layers to select the layer that minimizes the time complexity as the starting layer. Yang et al. [54] parallelize BCL, resulting in BCLP, where vertices from the selected layer are distributed to different CPU threads, and each thread executes the BCL algorithm. Nevertheless, it encounters performance issues outlined in § I when confronted with dataset size or biclique scale.

Running Basic in parallel on a GPU is straightforward by distributing tasks of different vertices to distinct threads (groups). However, simply porting it to the GPU does not fully harness the high-performance capabilities of the processor. In § III-B, we present the trivial GPU baseline derived from Basic and elaborate on why its efficiency is not fully realized, despite utilizing such a high-performance processor.

B. Baseline for GPU Implementation

The exploration initiated from a vertex u establishes a search tree with u as its root. The search trees of distinct vertices are mutually independent. To exploit parallelism, we allocate the vertices in the selected layer to various thread blocks. These blocks then autonomously undertake exploration to retrieve bicliques for their allocated vertices in a backtracking manner. Notably, vertices with 2-hop neighbors less than $p - 1$ are not allocated. This exclusion is due to the impossibility of finding a (p, q)-biclique on the search trees rooted at these vertices. To avoid duplicate results (e.g., the 1st and 3rd leaf nodes of the search tree in Figure 2), we assign a unique vertex priority (Definition 2) for the vertices on the selected layer and traverse vertices from high priority to low priority. The vertex priority defined in Definition 2 prevents concentration of computation on high-degree vertices caused by the power-law distribution of vertex degrees, thereby achieving a more balanced workload. Moreover, neighbors with lower priority are not stored to reduce memory overhead.

Definition 2 (Vertex Priority). *For any vertex u in the selected layer (suppose U) of the bipartite graph G, the priority $\mathcal{P}(u)$ is an integer in $[1, |U|]$. For any two vertices u and w in U, $\mathcal{P}(u) > \mathcal{P}(w)$ if:*
1) $|N_2^q(u)| < |N_2^q(w)|$, or
2) $|N_2^q(u)| = |N_2^q(w)|$ and $id(u) < id(w)$
where $id(u)$ is the unique vertex ID of u

Directly using recursion to materialize Basic on GPUs is infeasible. On one hand, managing memory distribution becomes challenging, and memory consumption is high when recursion is employed on GPUs [4]. On the other hand, recursion makes it difficult to share intermediate results among threads. Therefore, we opt for iteration and use arrays C_R and C_L to respectively store intersection results in each

level for backtracking. Nodes in the search trees entail two computationally intensive operations:
1) Intersect $C_L[l-1]$ and $N_2^q(u)$ for $C_L[l]$;
2) Intersect $C_R[l-1]$ and $N(u)$ for $C_R[l]$.

These two operations unify the processes of computing the candidate sets for both L and R into intersection calculations, holding significant potential for parallel execution, effectively harnessing the substantial parallelism capabilities of GPU.

We employ parallel binary search for conducting intersections on GPUs, a technique well-regarded for its efficiency [21]. With parallel binary search, threads organized within a warp retrieve an element from the sorted set $C_L[l-1]$ and perform searches against the elements within the sorted set $N_2^q(u)$. The rationale behind fetching elements from $C_L[l-1]$ rather than $N_2^q(u)$ is rooted in the fact that the size of $C_L[l-1]$ often remains smaller than that of $N_2^q(u)$. Likewise, we adopt a similar approach for computing the intersection results between $C_R[l-1]$ and $N(u)$.

Nonetheless, transferring the algorithm directly to the GPU leads to inefficiencies. In empirical experiments, there are instances where the GPU implementation, despite employing thousands of threads, exhibits slower performance compared to its CPU counterpart (§ VII-B). The limitations of this design have been elucidated as challenges in § I.

C. Solution Overview

To address these limitations, we introduce GPU-based biclique counting, named GBC, which includes a series of innovative designs. We initially formulate a search paradigm, specifically a hybrid DFS-BFS search, to enhance thread utilization (§ IV). Building upon this paradigm, we put forth a series of optimization techniques to further elevate the performance of GBC. Firstly, we introduce a novel data structure, i.e., Hierarchical Truncated Bitmap (HTB), featuring truncated bitmaps to facilitate fast intersection (§ V-A). We further devise a vertex reordering strategy tailored to compact bitmaps to mitigate memory overhead and access costs (§V-B). Secondly, we develop a joint load balancing strategy, which combines pre-runtime task allocation and runtime task stealing, to achieve an equitable distribution of workloads across GPU blocks (§ V-C). Lastly, we propose a communication-free graph partitioning method, known as BCPar, to handle graphs that surpass the capacity of device memory (§ VI).

IV. HYBRID DFS-BFS EXPLORATION

In this section, we introduce the search paradigm in GBC, which serves as the backbone of our methodology.

The GPU baseline follows DFS exploration, which gives rise to two performance issues. Firstly, due to GPU memory transactions being executed in a coalesced manner, DFS exploration leads to bandwidth wastage as only one vertex is processed at a time. Secondly, as the search layers deepen, the sizes of intermediate results ($C_L[l]$ and $C_R[l]$) progressively decrease, often falling below the number of threads in a warp. Consequently, at deeper search levels, there are significantly fewer active threads compared to the large pool of available threads, resulting in substantially reduced thread utilization.
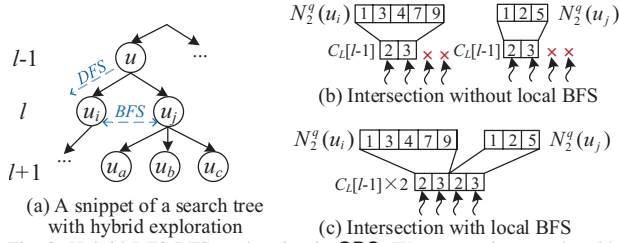
Fig. 3: Hybrid DFS-BFS exploration in GBC (We use vertices newly added on the selected layer to denote nodes in the search tree for brevity).

Note that, the search nodes at level l sharing the same parent possess identical $C_L[l-1]$ and $C_R[l-1]$ (e.g., C_L and C_R before update in different nodes at level $l = 1$ in Figure 2(b)). When applying intersection operations, the only distinction lies in their adjacency lists (e.g., $N_2^q(u_2), N(u_2)$ of u_2 vs. $N_2^q(u_3), N(u_3)$ of u_3 at level 1 in Figure 2). Furthermore, the intersection operations among these children are mutually independent. This observation presents an opportunity to simultaneously apply intersections for the children, akin to a BFS approach concerning the parent, which contributes to bandwidth utilization and enhanced parallelism. However, embracing a global BFS exploration poses considerable memory challenges [27]. Consequently, considering the strengths and weaknesses of both DFS and BFS, we harmonize their advantages while mitigating their drawbacks. This assembly results in a hybrid DFS-BFS strategy, where GBC is executed in a DFS fashion globally while incorporating a local BFS strategy in specific regions. This approach enhances parallelism while effectively managing memory constraints. To clarify, let's consider a node in the search tree with n children at level l. In order to concurrently compute $C_L(l)$ for all children, we duplicate $C_L[l-1]$ n times and concatenate these duplicates to form an extended array, which is then stored in shared memory. Subsequently, each thread within a warp retrieves an element from this extended array and performs a binary search within $N_2^q(u)$. A similar approach is applied to calculate $C_R[l]$.

Suppose there are k warps handling tasks for the children of a specific vertex, and let $|C_L[l-1]|(|C_R[l-1]|) = m$. Without optimization, we would be required to perform $\lceil \frac{m}{32 \times k} \rceil \times n$ intersection operations, while with optimization, only $\lceil \frac{m \times n}{32 \times k} \rceil$ intersection operations are necessitated. In cases where $m < 32 \times k$, which is common in practice, the number of intersection operations is n without optimization, compared to $\lceil \frac{m}{32 \times k} \times n \rceil$ with optimization. It's worth noting that as the search level increases, the value of m typically decreases, leading to a more efficient optimization performance.

Example 3. *Consider a snippet of a search tree in Figure 3(a), where we compute $C_L[l]$ for all children $\{u_i, u_j\}$ of vertex u at level l. Suppose there are 4 threads in a warp. As depicted in Figure 3(b), using only DFS with all threads assigned to a vertex (u_i or u_j) to obtain $C_L[l]$ for u_i, u_j requires two separate intersection operations, with two out of four threads remaining idle, resulting in a significant waste of computing resources. In contrast, with hybrid DFS-BFS exploration, where we handle u_i and u_j concurrently by*

Algorithm 1: GBC

Input: a bipartite graph G, two integer p and q
Output: all (p,q)-bicliques $\mathcal{B}$

1 Select one layer in G as an anchor;
2 Collect 2-hop neighbors $N_2^q(\cdot)$ for vertices in the anchored layer;
3 Filter unpromising vertices and obtain a set $Roots$;
4 GPUBasedListing $(G, \mathcal{B}, p, q, Roots)$;
5 **return** $\mathcal{B}$;
6 **procedure** GPUBasedListing $(G, \mathcal{B}, p, q, Roots)$
7 $i \leftarrow blockIdx$;
8 **while** $i < |Roots|$ **do**
9   $C_R[blockIdx], C_L[blockIdx] \leftarrow p$ empty arrays;
10   $u \leftarrow Roots[i], L \leftarrow \emptyset, l \leftarrow 0$;
11   $C_R[blockIdx][0] \leftarrow N(u)$;
12   $C_L[blockIdx][0] \leftarrow N_2^q(u)$;
13   **while** $l \geq 0$ **do**
14    $B \leftarrow$ next batch in $C_L[blockIdx][l]$;
15   **if** $threadIdx < set\_num$ **then**
16   IntersectionBatch (B, C_R, l, V);
17   **else**
18   IntersectionBatch (B, C_L, l, U);
19   **if** $l = p$ **then**
20   **foreach** $R \subseteq C_R[l-1] \land |R| = q$ **do**
21      $\mathcal{B} \leftarrow \mathcal{B} \cup (L, R)$;
22     $l \leftarrow l-1, L \leftarrow L - \{u\}$;
23   **return**;
24   **foreach** $u' \in C_L[blockIdx][l]$ **do**
25   **if** *Pruning conditions are not satisfied* **then**
26      $u \leftarrow u', l \leftarrow l+1, L \leftarrow L \cup \{u'\}$;
27      $C_L[blockIdx][l] - \{u'\}$;
28   **if** $C_L[blockIdx][l] = \emptyset$ **then**
29     $l \leftarrow l-1, L \leftarrow L - \{u\}$;
30   $i \leftarrow i + gridDim$;
31 **procedure** IntersectionBatch (B, Arr, l, G)
32 **while** $B \neq \emptyset$ **do**
33   Copy $Arr[l-1]$ $|B|$ times into buffer;
34   Intersect with $N(u')$ or $N_2^q(u')$ where $u' \in B$;
35   Write results into $Arr[l]$;

---

duplicating $C_L[l-1]$, only one intersection is necessary to acquire the results for u_i, u_j, as depicted in Figure 3(c). This approach leads to a more efficient utilization of computing resources, ultimately reducing time consumption.

Batching. The shared memory's capacity is quite limited and adhering to the BFS phase of the previous hybrid exploration strategy may lead to an explosion of shared memory usage with some large-sized adjacency lists. To overcome this limitation and ensure the strategy aligns with the constraints of limited memory, we resort to batching. This adaptation allows us to effectively manage the memory while maintaining the benefits of parallelism. Concretely, GBC divides the children into multiple independent batches with size $\lfloor \frac{|B|}{|C_L[l-1]|} \rfloor$, where B is the buffer allocated in shared memory for storing the duplicates. Once a batch is processed using BFS, it switches to DFS instead of continuing with the remaining batches.

3195

(a) Intersection with CSR   (b) Hierarchical Truncated Bitmap (HTB)   (c) Intersection with HTB
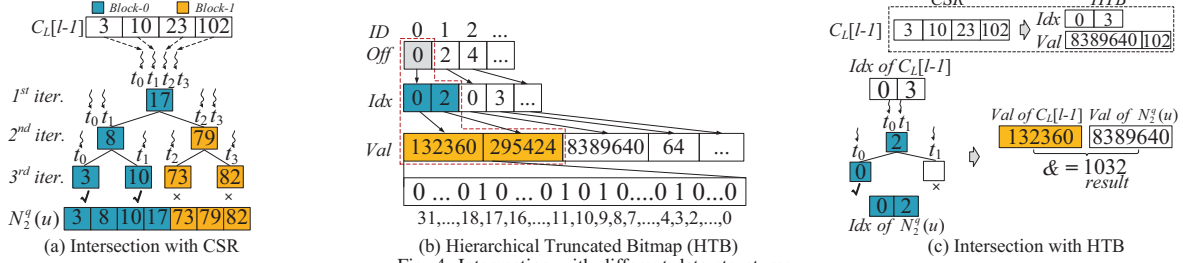
Fig. 4: Intersection with different data structures.

Algorithm 1 shows the pseudo-code of GBC. At first, we select one layer and collect the 2-hop neighbors for each vertex in the selected layer[4]. The main part of the algorithm is GPUBasedListing. In this procedure, we first initialize the arrays C_R and C_L to store the intermediate results and distribute root vertices to each thread block (lines 9–12). Then we retrieve bicliques level by level (lines 13–29). For each batch B in $C_L[blockIdx][l]$, we employ the first set_num threads to calculate $C_R[l]$ by intersecting $C_R[l-1]$ and $N(u')$ for $u' \in B$ via IntersectionBatch. While the other threads calculate the results of $C_L[l]$ simultaneously (lines 14–18). When the search reaches the last level, we store the qualified bicliques into $\mathcal{B}$ and backtrack (lines 19–23). Otherwise, for each vertex $u' \in C_L[blockIdx][l]$, we examine if the intersection results satisfy pruning conditions, i.e. $|C_R \cap N(u')| \geq q$ and $|C_L \cap N_2^q(u')| \geq p - l - 1$. The qualified vertices move to the next iteration (lines 24–27).

Discussion. GBC adopt the layer-based approach as BCLP [54] for efficiency consideration. However, they differ significantly. First, GBC adopts iteration instead of recursion to alleviate memory issues and opens the potential for reusing intermediate results. Moreover, to fully harness the parallel power of the GPU, we implement a hybrid DFS-BFS exploration rather than pure DFS as in BCLP. Second, BCLP preallocates arrays for each vertex (i.e., task) on the selected layer, leading to high memory demand on the GPU due to the large number of parallel tasks. Additionally, the labeling technique of BCLP introduces substantial data movement, which proves time-consuming on the GPU. Therefore, GBC employs parallel intersection computation, which is efficient on the GPU and helps conserve memory. Third, we introduce novel optimizations for enhanced efficiency, including optimizing data structures, vertex reordering, and load balancing strategies. In summary, we did not directly utilize the parallel framework of BCLP. Instead, we implemented numerous non-trivial optimizations specifically tailored for GPU architecture to achieve superior performance with GBC.

V. OPTIMIZATIONS

In this section, we first discuss the limitations of parallel binary search for intersection computation. Motivated by this, we present an advanced data structure and vertex reordering technique to enhance intersection implementation. Finally, we address load imbalance through pre-runtime and runtime workload distribution.

[4]We adopt the layer selecting strategy proposed in [54] for its effectiveness.

A. Hierarchical Truncated Bitmap

As mentioned in § I, intersection computation constitutes the majority of the workload, accounting for over 90%. Although proven to be efficient [6], two drawbacks persist in parallel binary search on GPU. Firstly, binary search necessitates entry-by-entry comparisons, resulting in high computational overhead. Secondly, comparisons in each iteration require accessing entries stored in global memory, potentially leading to excessive memory transactions. The following example illustrates how intersection performs with binary search under CSR format on GPU and its corresponding limitations.

Example 4. *In Figure 4(a), suppose we compute $C_L[l] = C_L[l-1] \cap N_2^q(u)$ with $C_L[l-1] = \{3, 10, 23, 102\}$ and $N_2^q(u) = \{3, 8, 10, 17, 73, 79, 82\}$, and the entries in $C_L[l-1]$ work as the search keys (same workflow for $C_R[l]$). For simplicity, we assume there are 4 threads in a warp, and each memory transaction accesses 4 integers. In the first iteration, 4 threads access entry 17 residing in the same block with one memory transaction. In the second iteration, thread-0,1 and thread-2,3 access entries 8 and 79 distributed across two data blocks, hence, two memory transactions are required. Similarly, another two memory transactions are required.*

Limitation. Totally, binary search, in Example 4, performs five memory transactions, while a majority experiencing *repeated scheduling*. Specifically, block-0 and block-1 have been transmitted thrice and twice, respectively. These substantial memory transactions stem from the *entry-by-entry comparisons* with entries residing in different blocks of global memory.

Inspired by the observations above, the key insight to enhance the efficiency of parallel binary search is to narrow down the search space to avoid entry-entry comparisons and compress the data to reduce transaction overhead. To achieve this, we devise a new GPU-friendly data structure, known as Hierarchical Truncated Bitmap (HTB), to store the adjacency lists. The main idea of HTB is to use a single 32-bit integer to represent multiple neighbors of a vertex, where each integer accommodating neighbor IDs within the range $[i, i+32]$, where i signifies the ordinal position in the consecutive integer sequence. For each neighbor u' of vertex u, we hash it to the j-th bit in the i-th integer and set the bit value to 1 to denote its presence, where $i = ID(u')/32, j = ID(u')\%32$. We then aggregate is and perceive it as an index for these collections.

Figure 4(b) presents a sketch of HTB, which consists of three tiers of arrays: (i) **Off** denotes the starting position of Idx corresponding to each vertex, wherein the index of vertex

u occupies $Idx[Off[ID(u)] : Off[ID(u) + 1] - 1]$, (ii) **Idx** signifies the series of the integers that conserve the neighbors, i.e., is, and (iii) **Val** indicates the specific bit position within a 32-bit integer where a neighbor is stored, i.e., js.

Example 5. *Suppose $ID(u) = 0$, for each $u' \in N_2^q(u)$ in Figure 4(a), HTB hashes them into (i, j) pairs, which are $\{(0,3), (0,8), (0,10), (0,17), (2,9), (2,15), (2,18)\}$. For instance, u' with $ID(u') = 3$, whose (i, j) pair is $(0, 3)$, is stored in the 0-th integer with the j-th bit being 1. Consequently, $N_2^q(u)$ is distributed into two integers, i.e., the 0-th and 2-nd integer stored in $Idx[Off[0] : Off[1] - 1] = \{0, 2\}$, with values in Val being $\{132360, 295424\}$, respectively.*

Intersection is executed in a two-phase manner with HTB. In the first phase, we leverage Idx to determine the search range, followed by the extraction of Val containing the collection of neighbors. In the second phase, a bitwise AND operation is conducted to ascertain whether the search key exists in the search list or not. Figure 4(c) depicts the workflow of binary search with HTB.

Example 6. *Utilizing HTB, we first perform a binary search with the Idx of $C_L[l-1]$ ($\{0, 3\}$) over that of u ($\{0, 2\}$), yielding the search result $\{0\}$ (left part in Figure 4(c)). Thereafter, we circumvent the need for searching entries in $N_2^q(u)$ stored in the 3-rd integer. Next, we perform the bitwise AND operation between the values in Val of $C_L[l-1]$ and $N_2^q(u)$ corresponding to the 0-th integer (i.e., 8389640 and 132360), resulting in the outcome 1032, including neighbors $\{3, 10\}$ (right part in Figure 4(c)).*

We analyze the number of memory transactions in Example 6. Executing binary search requires one memory transaction to load the Idx of $N_2^q(u)$ and another memory transaction to access the corresponding entries in Val. Consequently, two memory transactions are needed, leading to a reduction of three memory transactions compared to directly executing a binary search with CSR. Furthermore, the utilization of bitwise AND operations contributes to computational efficiency.

B. Vertex Reordering

HTB's efficiency depends on the degree of discreteness in the adjacency lists. The narrower the gap between adjacent entries, the fewer integers are required to accommodate the adjacency lists. This results in higher efficiency for HTB, as fewer memory reads are required and more entries can be compared in a single bitwise AND operation. In this section, we introduce Border, a vertex reordering technique to optimize the data layout to reduce the size of HTB.

Various methods have been devised for reordering vertices in unipartite graphs to improve the hit rate of cache [7], [9], [40], [51]. However, reordering vertices for (p, q)-biclique counting on GPU poses new challenges. First of all, we have to reorder vertices in different layers of G separately since reordering them as a whole would disrupt the vertices' orders on the left and right sides. More importantly, HTB introduces a new requirement for the reordering algorithm to maximize

the number of vertices hashed into one integer. Motivated by the above considerations, we devise Border tailored to reorder vertices of the bipartite graph to prevent their disruption while optimizing the layout for efficiency consideration of HTB.

Algorithm 2 shows the pseudo-code of Border. Border first maps G to an adjacency matrix M, where $M(i, j) \in \{0, 1\}$ denotes the entry located at the i-th row and the j-th column. Within this matrix, we refer to 32 entries whose indices are within $[i, 32 \times k : 32 \times (k + 1)](k \in \mathbb{Z}^+)$ in each row as a *block*. An *m-block* refers to *block* containing m 1s. Our observation on real datasets reveals that the abundance of 1-*block*s significantly dominates the realm of non-zero *block*s, which deteriorates the efficiency of HTB owing to their sparsity. Consequently, the pursuit of minimizing the quantity of 1-*block*s results in a more condensed bit representation within an integer of HTB, thus reducing the sizes of *Val* required to encode each adjacency list and improve HTB's efficiency. Border employs a greedy strategy in each iteration to minimize the number of 1-*block*s, which includes four steps:

- **Step 1**: Identify vertex v_m with most 1-*block*s (line 9);
- **Step 2**: Construct the candidate set $CandV$ by including vertices sharing the fewest common neighbors with v_m (lines 10–12);
- **Step 3**: Calculate profits for all vertices in $CandV$ and select vertex v_n with the highest profit (lines 13–16);
- **Step 4**: Exchange the positions of v_m and v_n and update matrix M (lines 17–19).

Intuitively, each iteration endeavors to reduce the quantity of 1-*block*s in M as much as possible, and the profit derived from exchanging v_m and v_n is numerically equivalent to the count of reduced 1-*block*s. Concerning the *block*s that v_m locates at, the change in the number of 1-*block*s is divided into two parts: the number of 1-*block*s converted into 0-*block*s, denoted as x_m, and the number of 0-*block*s converted into 1-*block*s, denoted as y_m. Similarly, for v_n, these two parts are denoted as x_n and y_n. The profit is calculated as $x_m + x_n - y_m - y_n$.

Notably, we reorder the vertices based on their degrees before executing Border. This preprocessing enhances the compactness of the layout for the adjacency lists of each vertex, consequently diminishing the count of 1-*block*s and thereby reducing the number of iterations. Additionally, step 2 in Border is efficiently performed using matrix multiplication.

Example 7. *Using the adjacency matrix in Figure 5(a) as an example to illustrate an iteration in Algorithm 2, assuming 4 bits constitute a block of HTB for brevity.*

In Figure 5(b), the integers positioned above the vertices indicate the quantity of 1-blocks (depicted as gray boxes) for vertices to be reordered. Border first identifies the vertex with the highest count of 1-blocks, specifically, v_5. Subsequently, Border finds vertices sharing the fewest common neighbors with v_5 as candidates to exchange order. This is achieved by multiplying the vector $(1, 1, 1, 1)$ (neighbors of v_5) with matrix M, resulting in $CandV = \{v_1, v_3, v_4, v_{11}\}$. Border proceeds to compute the potential profits, denoted by the integers above the vertices in Figure 5(c), associated with the exchange of

Algorithm 2: Border

Input: a bipartite graph $G(U, V, E)$, number of
        iterations itr, reorder layer rl
Output: G with reordered vertices

1 **if** $flag = 0$ **then** exchange U, V;
2 Map G to an adjacency matrix M;
3 $OBV \leftarrow \emptyset$, $CandV \leftarrow \emptyset$, $max\_profit \leftarrow 0$;
4 **foreach** $blk \in 1\text{-}blocks$ **do**
5   **if** $blk[\cdot, v] = 1$ **then**
6   OBV.append(v);

7 Count the number of 1-$block$s for each vertex in OBV;
8 **while** $itr - -$ **do**
9   Find the vertices v_m with the most 1-$block$s;
10   **foreach** $v \in U$ **do**
11   **if** $|N(v) \cap N(v_m)|$ *is the smallest* **then**
12       $CandV \leftarrow CandV \cup \{v\}$;

13   **foreach** $v \in CandV$ **do**
14   **if** $Profit(v) \geq max\_profit$ **then**
15       $v_n \leftarrow v$;
16       $max\_profit \leftarrow Profit(v)$;

17   Exchange the order of v_m and v_n;
18   Update the number of 1-$block$ of each vertex;
19   Update the matrix M;

20 **return** G;

---

v_5 *with candidates in* $CandV$. **Border** *finally selects* v_3 *for order exchange with* v_5, *which yields the maximum profit. The matrix after the exchange is shown in Figure 5(d).*

C. Load Balancing

Load imbalance is a common dilemma in multi-threaded environments. In the studied problem, two factors contribute to this issue: (1) the disparate sizes of adjacency lists lead to uneven workloads, and (2) the dynamic nature of DFS-oriented exploration poses challenges in estimating workloads during runtime, exacerbating the load imbalance issue and hindering the application of only static solutions. To tackle this issue, we formulate a joint load balancing strategy that integrates pre-runtime task allocation with runtime task stealing.

Pre-runtime load balancing. Evenly distributing vertices in the selected layer among different blocks results in pronounced load imbalance due to variations in the sizes of search trees that arise from the uneven lengths of adjacency lists. It is noteworthy that the height of each search tree remains consistent, with its value being p or q, depending on the selected layer. Consequently, allocating vertices located at higher levels in the search trees to each thread block yields a more equitable distribution of workloads among the blocks. We embrace an edge-oriented approach that evenly distributes the second-level vertices of the search trees across different thread blocks.

Runtime load balancing. The dynamic nature of DFS, characterized by unquantifiable workloads, gives rise to disparities of workloads among the blocks at runtime. To guarantee a more equitable distribution of workloads, we implement a work stealing mechanism, whereby the unoccupied blocks are dynamically assigned (sub)tasks from those currently engaged.
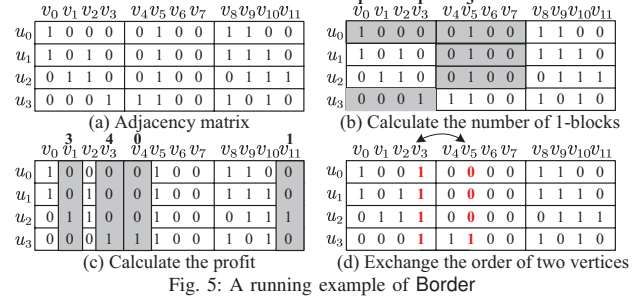

Fig. 5: A running example of Border

In Figure 6, we maintain an array *GCL* with the same size as the number of blocks on the GPU to denote the processing status of blocks, where *GCL*[i] records the number of processed vertices by block-i and serve as the starting point if the work is stolen. *GCL*[i] will be set to *0xFFFFFFFF* when all vertices assigned to block-i have been processed, indicating that work stealing should skip this block. We have the block that has completed its assigned tasks persist in searching for unfinished blocks through *GCL*, as shown in Figure 6(a). Once a qualified block is identified, as depicted in Figure 6(b), it locks the corresponding entry (block) in *GCL* and reads the value. Subsequently, it calculates the index of the next vertices to be processed and updates the *GCL* array as in Figure 6(c). Finally, it frees the locked entry as illustrated in Figure 6(d).

Note that, [33], [45], [50] statically allocate thread groups based on adjacency list lengths, which cannot ensure load balance due to the dynamic nature of computations [24]. Additionally, [50] incurs overhead from adjacency list partitioning, while [33] requires synchronization before thread allocation. In contrast, we integrate dynamic work stealing to ensure load balance during execution by efficiently utilizing idle threads.
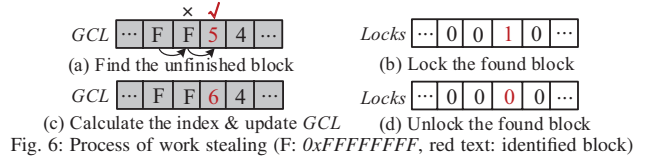

Fig. 6: Process of work stealing (F: *0xFFFFFFFF*, red text: identified block)

VI. OUT-OF-MEMORY SETTING

Given that global memory on GPU is considerably smaller than its CPU counterpart, typically ranging from a few gigabytes to several tens of gigabytes, the feasibility of holding the entire graph in global memory has been rendered impractical due to the swift proliferation of real-world graphs. With their sizes surpassing several tens of gigabytes and, in some cases, even reaching several hundred gigabytes, the necessity of segmenting large graphs into more manageable subgraphs that can fit within the confines of global memory has become a pressing imperative. Despite certain literature having delved into graph partitioning [5], [15], [23], their oversight in considering the properties of the bicliques has led to superfluous data transfer, which even dominates the overall performance. This occurs when required data is transferred on demand, and a certain portion of data is transferred multiple times [19]. To enhance graph partitioning for biclique counting on GPU, we introduce a biclique-aware partitioning solution named BCPar.

Algorithm 3: BCPar

Input: vertex set U of selected layer, neighbor structures $N(\cdot)$ and $N_2^q(\cdot)$, memory budget M

Output: partition $P = \{g_1, g_2, ..., g_k\}$

1 Compute weight $w(u) = |N(u)| + |N_2^q(u)|$, $u \in U$;
2 Compute average weight
  $avg_w(u) = \frac{1}{|N_2^q(u)|} \sum_{u' \in N_2^q(u)} w(u')$, $u \in U$;
3 Initialize an array L with U in descending order;
4 $P \leftarrow \emptyset$;
5 **while** $U \neq \emptyset$ **do**
6     $p \leftarrow \emptyset$, $p' \leftarrow \emptyset$, $Q \leftarrow \emptyset$, $cost \leftarrow 0$;
7     $u \leftarrow L.\text{pop}()$, $p.\text{insert}(u)$;
8   **foreach** $u' \in N_2^q(u) \cup u$ **do**
9        $p'.\text{insert}(u')$;
10        $cost{+}= (|N(u')| + |N_2^q(u')|)$;
11   **while** *True* **do**
12   **foreach** $u' \in p'$ **do**
13   **foreach** $v \in N_{in}(u')$ **do**
14   **if** $v \notin Q$ **then**
15                 $Q.\text{insert}(v)$;
16   Increase the weight of v in Q by $w(v)$;
17        $u \leftarrow Q.\text{pop}()$;
18   **foreach** $u' \in N_2^q(u) \cup u$ **do**
19   **if** $u' \notin p'$ **then**
20              $p'.\text{insert}(u')$;
21              $cost{+}= (|N(u')| + |N_2^q(u')|)$;
22   **if** $cost > M$ **then**
23           $P.\text{append}(p)$,**break**;
24   **else** $p.\text{insert}(u)$, $U.\text{delete}(u)$;

25 **return** P;

Lemma 1. *Obtaining the optimal graph partitioning result is an NP-hard problem.*

Proof. The proof of Lemma 1 is straightforward with the knapsack problem and thus is omitted. □

We observe that since $C_L[l]$ is obtained through iterative intersections with $C_L[0]$, i.e., $N_2^q(u)$, the vertices in $C_L[l]$ necessarily belong to $N_2^q(u)$, i.e., $\forall u' \in C_L[l], u' \in N_2^q(u), 0 \leq l \leq p-1$. To construct $C_L[l]$, we intersect $C_L[l-1]$ with the 2-hop neighbors $N_2^q(u')$ of the newly added vertex u' in L. Hence, we only need to consider the 2-hop neighbors $N_2^q(u)$ of the starting vertex u and 2-hop neighbors $N_2^q(u')$ of u's 2-hop neighbors $u' \in N_2^q(u)$ for computing C_L of u. Similarly, C_R is determined by the 1-hop neighbors of the starting vertex and its 2-hop neighbors.

Thereafter, to retrieve bicliques for u, we collectively group the 1-hop and 2-hop neighbors of $\{u \cup N_2^q(u)\}$, rather than loading them on demand. As the neighbors of a vertex may be shared by others, we aim for maximal sharing of the neighbors among vertices within the same partition to minimize partition sizes and the number of partitions. However, achieving the optimal partitioning result is NP-hard (Lemma 1). We propose a greedy solution, namely BCPar, to address the partition problem, as shown in Algorithm 3.

BCPar takes as inputs the vertex set U of the selected layer, neighbor structure $N(\cdot)$ and $N_2^q(\cdot)$, the memory budget

M, and returns the partitioning result P. The algorithm first computes the weight w and the average weight avg_w for each vertex in U (lines 1–2). Then, the algorithm sorts the vertices in U in descending order and stores them in array L (line 3).

BCPar leverages a while-loop to partition the vertices (lines 5–22). In each iteration, four variables are initialized: p for storing the vertices in the current partition, p' for recording the vertices in p and their 2-hop neighbors, Q for storing the in-neighbors of vertices in p', and $cost$ for tracking partition size (line 6). The $cost$ of partition p is the total number of the vertices in p' and the corresponding 1-hop and 2-hop neighbors. Subsequently, BCPar prioritizes the vertex u with the maximal average weight to initialize p (line 7). The rationale behind this is that if the neighbors of the vertex with the maximal average weight can be shared by other vertices, there exists a heightened likelihood of achieving better compression gains. BCPar uses a hash table p' to record u and its 2-hop neighbors, subsequently adjusting the cost after the insertion of u into the current partition (lines 8–10). The algorithm iteratively appends vertices to the ongoing partition p (lines 11–23). For each vertex u' in p', the algorithm increments the weight of their corresponding in-neighbors by $w(u')$, signifying that if vertex v is inserted, the cost is reduced by $w(u')$ (lines 12–16). Q is established as a max-heap consisting of the in-neighbors of vertices in p'. The algorithm then selects the vertex with maximal weight in Q as a candidate for the current partition p (line 17). The reason behind this is that inserting the vertex with the maximum weight can reduce the partition size to the greatest extent possible. Subsequently, BCPar checks if inserting u' satisfies the memory requirement M. If not, the ongoing partition p is appended to the result set P, concluding the current search. Otherwise, u is inserted into p, and the exploration continues.

BCPar holds two primary advantages. Firstly, the sharing of multiple neighbors among vertices within the same partition significantly reduces the partition size. Secondly, it enhances load balancing, leading to improved performance.

VII. EVALUATION

In this section, we evaluate the performance of GBC and conduct a comparative evaluation with the state-of-the-art (p, q)-biclique counting algorithms.

A. Experimental Setup

Evaluated Methods. We compare GBC with three algorithms: (1) GBL: the GPU baseline designed in § III-B; (2) BCL: the state-of-the-art CPU solution [54]; (3) BCLP: the parallelized BCL on CPU [54].

Platform. All experiments are conducted on an Ubuntu 20.04.2 server, featuring an Intel Core i9-10900K 3.70GHz CPU and an Nvidia GeForce RTX 3090 GPU. The GPU has 82 SMs and $10,496$ cores. We implement GBC in C++ under Nvidia CUDA 11.2. BCLP is executed with 16 threads.

Datasets. Table II summarizes the 11 datasets used for experiments. The first 9 datasets[5] are real datasets with 8 of them extensively employed in related research [38], [54]. The
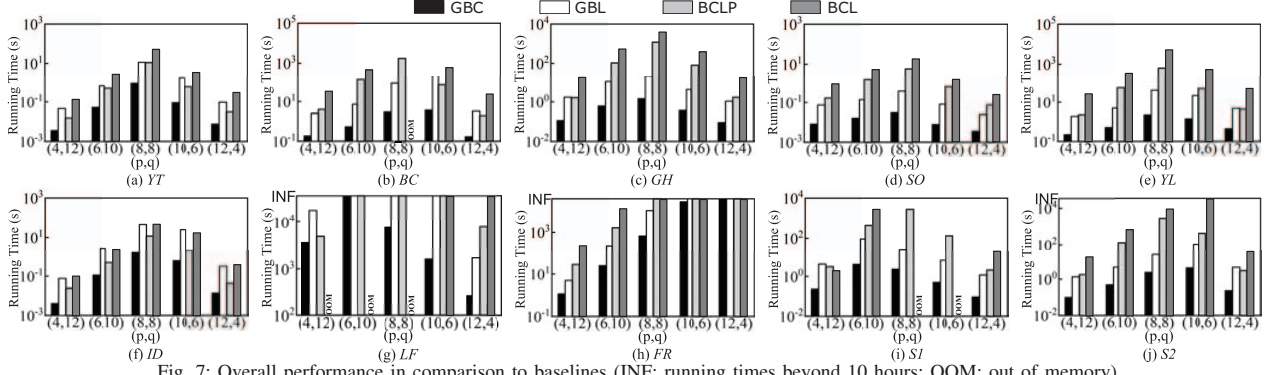
[5]http://konect.cc/networks

Fig. 7: Overall performance in comparison to baselines (INF: running times beyond 10 hours; OOM: out of memory).

TABLE II: Detailed descriptions of datasets.

Datasets	$|U|$	$|V|$	$|E|$	d_U	d_V
Youtube (*YT*)	94,238	30,087	293,360	3.11	9.75
Bookcrossing (*BC*)	77,802	185,955	433,652	5.57	2.33
Github (*GH*)	56,519	120,867	440,237	7.79	3.64
StackOverflow (*SO*)	545,196	96,680	1,301,942	2.39	13.47
Yelp (*YL*)	31,668	38,048	1,561,406	49.31	41.04
IMDB (*ID*)	303,617	896,302	3,782,463	12.46	4.22
Lastfm (*LF*)	359,349	160,168	17,559,162	48.86	109.63
Edit_fr (*FR*)	16,874	3,416,271	23,443,737	1,389.34	6.86
Orkut (*OR*)	2,783,196	8,730,857	327,037,487	117.50	37.45
Synthetic 1 (*S1*)	6,720	5,300	207,146	30.83	39.08
Synthetic 2 (*S2*)	12,720	11,100	220,651	17.35	19.88

large dataset Orkut (*OR*) is prepared for evaluating scalability where data cannot entirely fit into the global memory. Besides, we generate 2 synthetic datasets (*S1* and *S2*) with more 2-hop neighbors than those in the real datasets, resulting in increased computational load and uneven workloads. The synthetic datasets are generated as follows: (1) fix the size of U and V, (2) determine the number of 2-hop neighbors for layer U according to the power-law distribution, then artificially adjust it to be slightly larger than that of the real datasets used. (3) randomly select neighbors from V for U based on the generated number of 2-hop neighbors.

Queries. We set $(p + q)$ in the range of 8 to 24, with 16 serving as the default value.

B. Overall Performance

Figure 7 illustrates the performance of different methods. In this context, methods exceeding a runtime of 10 hours are terminated and their values are set to INF.

First, GBC significantly outperforms the compared methods in all cases. On average, GBC achieves speed enhancements of 505.3× over BCL, 146.7× over BCLP and 15.7× over GBL. The considerable improvement of GBC stems from the full exploitation of GPU's computing resources. In the best cases, GBC achieves a remarkable 836.7× acceleration over BCLP ($p = q = 8$ on *GH*) on real datasets. Meanwhile, on synthetic datasets, GBC realizes an extraordinary 1217.6× acceleration over BCLP ($p = q = 8$ on *S2*). Compared to real datasets, GBC exhibits superior performance gain on synthetic datasets, owing to their elevated computational overheads. By harnessing the considerable parallelism of GPUs, GBC adeptly tackles demanding computations with notable efficacy.

Second, GBL's performance sometimes lags behind that of BCLP (e.g., on *TY* and *ID*). This discrepancy arises from

GBL being a ňaive migration of the basic model to the GPU platform, without advanced optimizations tailored for the GPU architecture. In contrast, through the incorporation of multiple innovative designs customized for GPUs, GBC attains optimal performances across all scenarios. The notable progress demonstrated by GBC underscores the importance of carefully adapting algorithms to harness the unique capabilities and parallel processing power of GPUs.

Third, varying parameters p and q yield disparate execution times. Generally, when either p or q is small, the depth of the search tree remains correspondingly shallow, leading to lower computation costs. Another noteworthy observation is that the speedup ratio of GBC tends to decrease as the disparity between the parameters increases. This is because the proposed techniques exhibit greater effectiveness with higher search trees, implying more computational workloads. Deeper levels in the high search tree result in smaller sizes of C_R and C_L, along with fewer rounds of intersection operations, thus making the optimizations considerably more impactful. Furthermore, workload imbalance is less pronounced when the height of the search trees is small.

Fourth, the improvement of GBC appears conservative on some datasets. For instance, GBC achieves a maximum speedup of only 7.4× faster than BCLP on *ID*. As observed in Figure 7, the running time of BCLP is relatively brief among those datasets with lower speedup ratios of GBC. This suggests that the CPU suffices to count (p, q)-bicliques in these datasets within a relatively short duration, thereby diminishing the necessity to offload the computation to the GPU. In contrast, datasets necessitating prolonged processing time on the CPU are more suited for GPU acceleration. The high parallelism offered by the GPU proves particularly advantageous for datasets with substantial computational demands, substantially reducing processing time.

C. Scalability Evaluation

We proceed to evaluate the scalability concerning different methods by varying the biclique size, i.e., the value of $(p+q)$, from 8 to 24 with an increment of 4, where $p = q = (p+q)/2$. Figure 8 shows the results on five representative datasets.

The first observation is that GBC consistently outperforms the compared methods across all test cases, exhibiting a substantial improvement ranging from 2.4× to 6298.1×. The
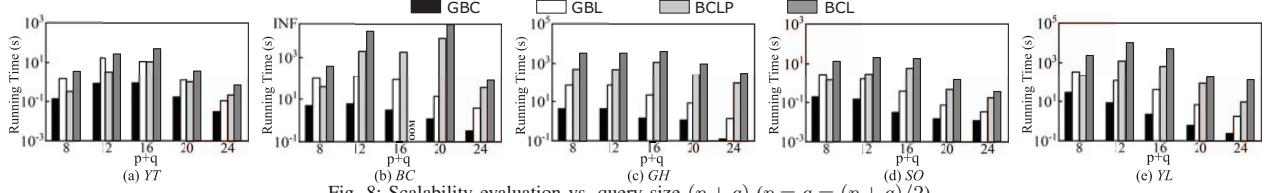
Fig. 8: Scalability evaluation vs. query size $(p + q)$ $(p = q = (p + q)/2)$.

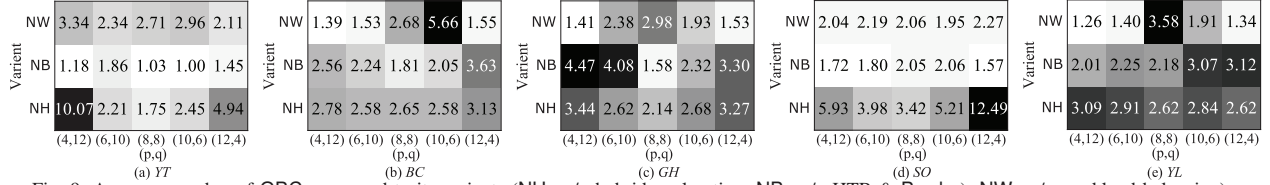
Fig. 9: Average speedup of GBC compared to its variants (NH: w/o hybrid exploration; NB: w/o HTB & Border); NW: w/o workload balancing).

(a) YT

Variant	(4,12)	(6,10)	(8,8)	(10,6)	(12,4)
NW	3.34	2.34	2.71	2.96	2.11
NB	1.18	1.86	1.03	1.00	1.45
NH	10.07	2.21	1.75	2.45	4.94

(b) BC

Variant	(4,12)	(6,10)	(8,8)	(10,6)	(12,4)
NW	1.39	1.53	2.68	5.66	1.55
NB	2.56	2.24	1.81	2.05	3.63
NH	2.78	2.58	2.65	2.58	3.13

(c) GH

Variant	(4,12)	(6,10)	(8,8)	(10,6)	(12,4)
NW	1.41	2.38	2.98	1.93	1.53
NB	4.47	4.08	1.58	2.32	3.30
NH	3.44	2.62	2.14	2.68	3.27

(d) SO

Variant	(4,12)	(6,10)	(8,8)	(10,6)	(12,4)
NW	2.04	2.19	2.06	1.95	2.27
NB	1.72	1.80	2.05	2.06	1.57
NH	5.93	3.98	3.42	5.21	12.49

(e) YL

Variant	(4,12)	(6,10)	(8,8)	(10,6)	(12,4)
NW	1.26	1.40	3.58	1.91	1.34
NB	2.01	2.25	2.18	3.07	3.12
NH	3.09	2.91	2.62	2.84	2.62

performance improvement of GBC becomes more notable with an increase in computational workload, as exemplified in datasets *BC* and *GH*. The performance enhancement stems from GBC's capability to fully unleash the computing resources of the GPU, which is specifically suitable for intensive computing. The second observation is that the running times of CPU solutions increase initially and then decrease with the growth of biclique size, whereas GPU methods generally exhibit comparable changes or continuous decreases as the queries grow in size. This is because GBC effectively balances the workload of parallel threads, preventing excessively long-running threads from becoming bottlenecks.

D. Effect of Individual Optimization

In this section, we evaluate the effect of the proposed optimizations by disabling specific modules of GBC, yielding three comparative variants: (1) NH omits the hybrid DFS-BFS exploration, (2) NB discards HTB and Border, and (3)NW excludes the workload balancing. Due to space constraints and similar experimental outcomes, we only present the results on five datasets (*YT*, *BC*, *GH*, *SO*, *YL*), as depicted in Figure 9.

Effect of Hybrid DFS-BFS Exploration. The hybrid DFS-BFS exploration plays a pivotal role in optimizing GBC, whose absence leads to a $3.7\times$ increase in average runtime. The integration of hybrid exploration allows us to fully harness the potential of GPU threads by concurrently applying intersections for multiple vertices during the BFS phase, resulting in more efficient thread utilization and thereby reducing the time required for intersections. This notably enhances GBC's performance on datasets with a low average degree or when reaching the bottom of the search trees with fewer candidates. For example, hybrid exploration leads to $4.2\times$ and $6.2\times$ reduction in runtimes for GBC on *YT* and *SO* (with average degrees less than 4), respectively. We further compare the performance between DFS and DFS-BFS (Appendix-B in [37]). On average, DFS-BFS incurs $1.3\times$ more memory overhead, remaining well below the GPU memory capacity. However, DFS-BFS demonstrates superior performance, being on average $2.2\times$ faster than DFS, attributed to the effective utilization of parallel threads offered by GPU.

Effect of HTB and Border. HTB and Border consistently enhance performance across all datasets, resulting in an av-erage speedup of $2.2\times$ over using CSR. This enhancement lies in that HTB and Border effectively compress adjacency lists, which reduces both element comparisons and memory accesses during binary search. This effect is particularly pronounced for datasets characterized by long adjacency lists, as a large proportion of elements can be compressed. For instance, GBC with HTB and Border achieves a speedup of more than $4\times$ on *GH* when $p = 4(6)$ and $q = 12(10)$.

We further explore Border's efficiency and compare it with a leading reordering method for unipartite graphs, i.e., Gorder [51]. We apply Border and Gorder to the graphs respectively and execute GBC with HTB on the reordered ones. Table III reports the experimental results. First, compared to no reordering, Gorder achieves an average speedup of $2.4\times$, whereas Border represents a notably higher average speedup of $3.1\times$, affirming the effectiveness of vertex reordering. Second, Border exhibits superior performance gain over Gorder across all datasets, averaging 37.0% faster than Gorder. This improvement is attributed to Border's specialization in optimizing the density of HTB and its tailored design for bipartite graphs. In contrast, Gorder prioritizes the hit rate of the CPU cache and reorders all vertices of the entire graph, potentially leading to inefficiencies in our specific problem domain.

Effect of Workload Balancing. The joint workload balancing yields a substantial optimization effect with an average speedup of $2.2\times$. Table IV further elaborates on the impact of different load balancing strategies. First, both pre-runtime and runtime strategies accelerate the algorithm, demonstrating their effectiveness in balancing workloads. Second, the pre-runtime strategy consistently outperforms the runtime strategy primarily due to its utilization of a fine-grained approach. This involves distributing vertices from the second layer to thread blocks, contrasting with the coarse-grained method of the runtime strategy, which dynamically reassigns unprocessed root vertices from occupied to idle blocks. Furthermore, the runtime strategy necessitates frequent global memory access for tasks such as block location and *GCL* rewriting. Lastly, the joint strategy demonstrates optimal performance in most scenarios, particularly under heavy workloads, underscoring the complementary nature of both strategies.

In conclusion, the DFS-BFS exploration, serving as the backbone, exhibits the highest average acceleration ($3.7\times$). Al-

TABLE III: Time costs (sec.) of GBC on (un)reordered graphs ($p = q = 8$).

Methods Datasets	No Reorder	Gorder	Border
YT	1.90	1.15	**0.87**
BC	14.00	3.29	**2.99**
GH	3.64	1.62	**1.45**
SO	0.15	0.04	**0.03**
YL	10.30	3.61	**2.25**
ID	5.64	2.19	**1.61**
LF	INF	17488.60	**7753.15**
FR	1251.01	946.36	**669.98**
S1	4.01	2.47	**2.29**
S2	4.65	2.59	**2.36**

TABLE IV: Time costs (sec.) of GBC varying load balancing strategies.

Methods \ Datasets	SO	S2	BC	LF	FR
No Balance	0.14	5.38	8.02	INF	3941.55
Pre-runtime Only	**0.03**	2.52	3.19	9071.55	804.31
Runtime Only	0.05	3.86	7.83	INF	2967.25
Joint	0.07	**2.36**	**2.99**	**7753.15**	**669.98**

though the collective impact of HTB and Border demonstrate a comparable speedup of $2.2\times$ to the workload balancing, their enhancement in performance gradually amplifies with larger datasets. For instance, on datasets *GH*, *SO*, and *YL*, their average speedup reaches $2.5\times$ (up to $3.1\times$), whereas the average speedup of load balancing is $2.0\times$ (up to $2.1\times$). This progression is attributed to the increasing workload of intersection computation emerging as the performance bottleneck.

E. Evaluating Graph Partition

Finally, we evaluate the effectiveness of BCPar through a comparative analysis with the widely-used graph partitioning algorithm, i.e., METIS [23], on the large dataset *OR*. Given that METIS is originally designed for unipartite graphs, we construct an auxiliary graph to serve as input for METIS. This auxiliary graph contains the vertices within the selected layer, with pairwise connections established if and only if two vertices are mutual 2-hop neighbors. Figure 10 reports the average throughput on the partitioned graphs, i.e., the number of bicliques traversed per second.

As shown in Figure 10(a), the throughput achieved on the graph partitioned by BCPar consistently surpasses its METIS counterpart. The superiority of BCPar arises from its ability to partition a graph into mutually autonomous subgraphs, thereby confining the search within the current subgraph, avoiding the need for introducing communication overheads. Moreover, BCPar requires the vertices in the same partition to share as many common neighbors as possible, which facilitates load balancing. In contrast, the subgraphs partitioned by METIS introduce dependencies among them, necessitating frequent data transfers via low-bandwidth PCIe to enumerate bicliques across partitions, thereby diminishing the overall throughput. Figure 10(b) illustrates the detailed throughputs concerning bicliques found in the same partition (referred to intra-partition) and those spanning across partitions (referred to inter-partition). Undoubtedly, the throughputs for enumerating inter-partition bicliques are markedly inferior to those for enumerating intra-partition bicliques, constituting the bottleneck for METIS. Conversely, BCPar mitigates this bottleneck by incorporating the structural properties of bicliques.
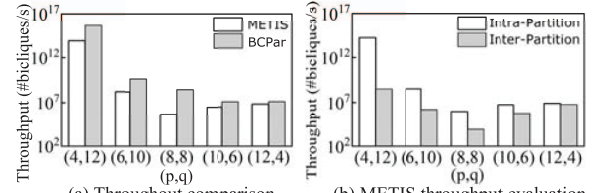

(a) Throughout comparison   (b) METIS throughput evaluation
Fig. 10: Throughput evaluation vs. different methods.

VIII. RELATED WORK

CPU-based Motif Counting on Bipartite Graphs. Wang et al. [46] initiate the exploration of butterfly counting and introduce wedge-based counting algorithms. Sanei-Mehri et al. [38] develop a random algorithm to estimate butterflies, while Wang et al. [38] formulate a vertex-priority strategy for butterfly listing. Recent studies extend butterfly counting to streaming graphs [39], [41], uncertain graphs [39], [41], and temporal graphs [11]. Over the years, several algorithms have been proposed for identifying maximal bicliques [14], [25]. Recently, maximal bicliques in signed bipartite graphs [43], uncertain graphs [47], and those with unique properties like fairness-aware maximal cliques [56] have been explored. Yang et al. [54] pioneers the (p, q)-biclique enumeration algorithm by introducing a recursive backtracking algorithm.

GPU-accelerated motif counting on bipartite and unipartite graphs. Xu et al. [53] pioneer GPU acceleration for butterfly counting, introducing a lock-free strategy to reduce synchronization and an adaptive strategy for workload balance. In unipartite graph motifs, triangle counting on GPUs [36] has seen various optimized algorithms, including workload estimation [18], bitmap-based intersection [8], and wedge-oriented methods [20], alongside preprocessing techniques [21]. In addition, there exists a wealth of literature on GPU acceleration for maximal clique enumeration [3], [52], k-clique counting [4], k-core decomposition [1], [31], [44] and k-truss decomposition [2], [12], [17]. To the best of our knowledge, accelerating (p, q)-biclique counting on GPUs has not been thoroughly investigated in the existing literature.

IX. CONCLUSION

We introduce GBC, a novel GPU-based approach for counting (p, q)-bicliques on large bipartite graphs. GBC enhances parallelism by adaptively consolidating tasks across multiple vertices during DFS. To facilitate efficient set intersection, we propose HTB, a novel data structure that reduces redundant comparisons and memory transactions. We further devise Border to compress HTB by reordering vertices. For scalability, we develop BCPar for handling large bipartite graphs beyond the GPU memory. Experimental results consistently demonstrate that GBC significantly surpasses existing algorithms, affirming its effectiveness, efficiency, and scalability.

X. ACKNOWLEDGEMENTS

REFERENCES

[1] A. Ahmad, L. Yuan, D. Yan, G. Guo, J. Chen, and C. Zhang. Accelerating k-core decomposition by a gpu. In *ICDE*, pages 1818–1831, 2023.

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

[3] M. Almasri, Y.-H. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu. Parallelizing maximal clique enumeration on gpus. *arXiv preprint arXiv:2212.01473*, 2022.

[4] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu. Parallel k-clique counting on gpus. In *ICS*, pages 1–14, 2022.

[5] K. Andreev and H. Räcke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.

[6] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[7] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *IPDPS*, pages 22–31, 2016.

[8] M. Bisson and M. Fatica. High performance exact triangle counting on gpus. *TPDS*, 28(12):3501–3510, 2017.

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

[10] S. P. Borgatti and M. G. Everett. Network analysis of 2-mode data. *Social networks*, 19(3):243–269, 1997.

[11] X. Cai, X. Ke, K. Wang, L. Chen, T. Zhang, Q. Liu, and Y. Gao. Efficient temporal butterfly counting and enumeration on temporal bipartite graphs. *arXiv preprint arXiv:2306.00893*, 2023.

[12] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo. Accelerating truss decomposition on heterogeneous processors. *PVLDB*, 13(10):1751–1764, 2020.

[13] J. Chen, Z. Chen, M. Wang, G. Fan, G. Zhong, O. Liu, W. Du, Z. Xu, and Z. Gong. A neural inference of user social interest for item recommendation. *DSE*, 8(3):223–233, 2023.

[14] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li. Efficient maximal biclique enumeration for large sparse bipartite graphs. *PVLDB*, 15(8):1559–1571, 2022.

[15] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *TOPC*, 5(3):1–39, 2019.

[16] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *PVLDB*, 13(8):1190–1205, 2020.

[17] S. Diab, M. G. Olabi, and I. El Hajj. Ktrussexplorer: Exploring the design space of k-truss decomposition optimizations on gpus. In *HPEC*, pages 1–8, 2020.

[18] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader. Logarithmic radix binning and vectorized triangle counting. In *HPEC*, pages 1–7, 2018.

[19] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*, pages 1067–1082, 2020.

[20] L. Hu, N. Guan, and L. Zou. Triangle counting on gpu using fine-grained task distribution. In *ICDEW*, pages 225–232, 2019.

[21] L. Hu, L. Zou, and Y. Liu. Accelerating triangle counting on GPU. In *SIGMOD*, pages 736–748, 2021.

[22] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on gpus. In *SC*, pages 171–182, 2018.

[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput*, page 359–392, 1998.

[24] P. Kumar and R. Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *CSUR*, 51(6):1–35, 2019.

[25] J. Li, G. Liu, H. Li, and L. Wong. Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: A one-to-one correspondence and mining algorithms. *TKDE*, 19(12):1625–1637, 2007.

[26] Z. Li, X. Shen, Y. Jiao, X. Pan, P. Zou, X. Meng, C. Yao, and J. Bu. Hierarchical bipartite graph neural networks: Towards large-scale e-commerce applications. In *ICDE*, pages 1677–1688, 2020.

[27] W. Lin, X. Xiao, X. Xie, and X.-L. Li. Network motif discovery: A gpu approach. *TKDE*, 29(3):513–528, 2016.

[28] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: An index-based approach. In *WWW*, pages 1130–1141, 2019.

[29] H. Liu and H. H. Huang. Enterprise: Breadth-first graph traversal on gpus. In *SC*, pages 1–12, 2015.

[30] S. Lu, B. He, Y. Li, and H. Fu. Accelerating exact constrained shortest paths on gpus. *PVLDB*, 14(4):547–559, 2020.

[31] A. Mehrafsa, S. Chester, and A. Thomo. Vectorising k-core decomposition for gpu acceleration. In *SSDBM*, pages 1–4, 2020.

[32] X. Meng, H. Huo, X. Zhang, W. Wang, and J. Zhu. A survey of personalized news recommendation. *DSE*, 8(4):396–416, 2023.

[33] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[34] M. Mitzenmacher, J. Pachocki, R. Peng, C. Tsourakakis, and S. C. Xu. Scalable large near-clique detection in large-scale networks via sampling. In *SIGKDD*, pages 815–824, 2015.

[35] S. Pandey, Z. Wang, S. Zhong, C. Tian, B. Zheng, X. Li, L. Li, A. Hoisie, C. Ding, D. Li, et al. Trust: Triangle counting reloaded on gpus. *TPDS*, 32(11):2646–2660, 2021.

[36] A. Polak. Counting triangles in large graphs on GPU. In *IPDPS Workshops*, pages 740–746, 2016.

[37] L. Qiu, Z. Li, X. Ke, L. Chen, and Y. Gao. Accelerating biclique counting on gpu. *arXiv preprint arXiv:2403.07858*, 2024.

[38] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura. Butterfly counting in bipartite networks. In *SIGKDD*, pages 2150–2159, 2018.

[39] S.-V. Sanei-Mehri, Y. Zhang, A. E. Sariyüce, and S. Tirthapura. Fleet: Butterfly estimation from a bipartite graph stream. In *SIGMOD*, pages 1201–1210, 2019.

[40] M. Sha, Y. Li, and K.-L. Tan. Self-adaptive graph traversal on gpus. In *SIGMOD*, pages 1558–1570, 2021.

[41] A. Sheshbolouki and M. T. Özsu. sgrapp: Butterfly approximation in streaming graphs. *TKDD*, 16(4):1–43, 2022.

[42] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang. Realtime top-k personalized pagerank over large graphs on gpus. *PVLDB*, 13(1):15–28, 2019.

[43] R. Sun, Y. Wu, C. Chen, X. Wang, W. Zhang, and X. Lin. Maximal balanced signed biclique enumeration in signed bipartite graphs. In *ICDE*, pages 1887–1899, 2022.

[44] A. Tripathy, F. Hohman, D. H. Chau, and O. Green. Scalable k-core decomposition for static graphs using a dynamic graph data structure. In *BigData*, pages 1134–1141, 2018.

[45] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang. Sep-graph: finding shortest execution paths for graph processing under a hybrid framework on gpu. In *PPoPP*, pages 38–52, 2019.

[46] J. Wang, A. W.-C. Fu, and J. Cheng. Rectangle counting in large bipartite graphs. In *BigData*, pages 17–24, 2014.

[47] J. Wang, J. Yang, Z. Ma, C. Zhang, S. Yang, and W. Zhang. Efficient maximal biclique enumeration on large uncertain bipartite graphs. *TKDE*, 2023.

[48] K. Wang, X. Lin, L. Qin, W. Zhang, and Y. Zhang. Efficient bitruss decomposition for large-scale bipartite graphs. In *ICDE*, pages 661–672, 2020.

[49] K. Wang, W. Zhang, X. Lin, Y. Zhang, L. Qin, and Y. Zhang. Efficient and effective community search on large-scale bipartite graphs. In *ICDE*, pages 85–96, 2021.

[50] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *PPoPP*, pages 1–12, 2016.

[51] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *SIGMOD*, pages 1813–1828, 2016.

[52] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai. Accelerating the bron-kerbosch algorithm for maximal clique enumeration using gpus. *TPDS*, 32(9):2352–2366, 2021.

[53] Q. Xu, F. Zhang, Z. Yao, L. Lu, X. Du, D. Deng, and B. He. Efficient load-balanced butterfly counting on gpu. *PVLDB*, 15(11):2450–2462, 2022.

[54] J. Yang, Y. Peng, and W. Zhang. (p, q)-biclique counting and enumeration for large sparse bipartite graphs. *PVLDB*, 15(2):141–153, 2021.

[55] R. Yang, J. Shi, K. Huang, and X. Xiao. Scalable and effective bipartite network embedding. In *SIGMOD*, pages 1977–1991, 2022.

[56] Z. Yin, Q. Zhang, W. Zhang, R.-H. Li, and G. Wang. Fairness-aware maximal biclique enumeration on bipartite graphs. *arXiv preprint arXiv:2303.03705*, 2023.