Response Letter for "Accelerating Biclique Counting on GPU"

Dear ICDE chairs, meta-reviewer, and reviewers,

We express our gratitude to all the reviewers. Our main changes in this revision are outlined as follows.

- **We have elevated the presentation** by (i) differentiating our framework from existing methods, (ii) clarifying the problem, applications, fundamental concepts, and motivation for optimizations, (iii) providing the link to codes/datasets, (iv) proofreading the manuscript to ensure clarity and correctness.
- **We have enhanced the experiments** by (i) elucidating the time components of experiments, (ii) examining the effects of individual techniques, (iii) testing memory consumption between DFS-BFS and DFS, (iv) comparing with existing graph systems, (v) detailing the experimental settings.

We addressed the reviewers' comments and delineated the changes in the following. In the revised paper, we have denoted the revisions in blue text to highlight the specific changes.

–Linshan, Zhonggen, Xiangyu, Lu and Yunjun

I. Response to meta-reviewer

Including discussions of GBC's methodology, their necessity, link to code/dataset, and experiments with Ligra (CPU) or Gunrock (GPU).

Response: Thanks for your support. We have thoroughly addressed all required revisions and outlined the main changes in a detailed point-by-point response provided below.

II. Response to reviewer #1

R1.O1: Issues about time components and Border's cost.

Response R1.O1: Thanks. The reported times are consistent with [51], only including the time for searching bicliques. We have enhanced our reporting by providing a comprehensive breakdown of the time spent on various components, including HTB transformation, Border execution, and biclique search (Table V in Appendix [?]). Notably, the time required for HTB transformations is minimal, typically ranging from hundreds to thousands of nanoseconds, proportional to the number of vertices. With the additional time cost for Border reordering, which typically ranges from 0.18s to 62.17s, we observe that the overall runtime decreases by up to 4.60×, highlighting the significant benefit of reordering in biclique searching. Furthermore, it's important to note that Border can be reused for different (p, q) parameters. Hence the amortized runtime of Border becomes practically negligible.

R1.O2: GBC resembles the established approach in [51].

Response R1.O2: Thanks for pointing this out. In the revised paper, we have clarified the distinctions between GBC and the parallel framework BCLP [51] in §IV. Although GBC adopt the layer-based approach as BCLP for efficiency consideration, they target different platforms and utilize distinct methodologies. First, BCLP relies on recursion on the CPU, leading to high memory consumption and challenging memory management. In contrast, GBC adopts an iterative approach 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 preallocate arrays to construct subgraphs for each vertex (i.e., task) on the selected layer to retrieve sub-cliques. This approach leads to high memory demand on the GPU due to the large number of parallel tasks. Additionally, the labeling technique of BCLP introduces significant 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 to achieve superior performance with GBC.

R1.O3: Speedup of each technique and core technique.

Response R1.O3: Thanks. In this revision, we have reorganized the paper and highlighted the effect of individual optimization in §VII.D. Overall, the DFS-BFS exploration, serving as the backbone, exhibits the highest average acceleration (3.7×). Although demonstrating a comparable speedup of 2.2× to the load balancing on all tested datasets, the collective enhancement of HTB and Border gradually amplifies with larger datasets. For instance, their average speedup reaches 2.5× (up to 3.1×), whereas the average speedup of load balancing is 2.0× (up to 2.1×) on *GH*, *SO*, and *YL*. This progression is attributed to the increasing workload of intersection computation emerging as the performance bottleneck.

R1.Required Changes: **(i)** *GCL*'s meaning; **(ii)** Memory comparison between DFS-BFS and DFS; **(iii)** (a) Applications, (b) Neighbor definition & intersection operation, (c) Recursive infeasibility on GPU; **(iv)** Applicability of existing load balancing methods.

Response R1.Required Changes: Thanks. In the revised paper, we have comprehensively addressed all required changes and conducted a thorough review to improve our presentations.

(i) We clarified in §V.C that *GCL* is an array of the same size as the number of blocks on the GPU, 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.

(ii) We conducted a comparison of the memory consumption between the DFS-BFS and DFS methods (Figure 11 in Appendix [?]). On average, DFS-BFS incurs 1.3× more memory overhead, remaining well below the GPU memory capacity.

However, DFS-BFS demonstrates superior performance, being on average 2.2× faster than DFS, attributed to the effective utilization of parallel threads offered by GPU.

(iii.a) In §I, we provided a comprehensive list of applications for (p, q)-bicliques, including densest subgraph detection [32], cohesive subgroup analysis [10] in bipartite graphs, and optimization of GNN information aggregation [51]. Notably, the problem of biclique counting has been extensively studied and established in [51]. Our paper concentrates on addressing the efficiency concerns associated with this established problem.

(iii.b) In §II.A, we refined the definitions of "1-hop and 2-hop neighbors" for better clarity. In our context, 1-hop neighbors refer to vertices directly connected to a given vertex u, denoted as $N(u, G) = \{v|(u, v) \in E(G)\}$, while 2-hop neighbors refer to vertices connected to u via one intermediate vertex, expressed as $N_2(u, G) = \{u'|(u, v) \in E(G) \wedge (u', v) \in E(G)\}$. 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.

In Figure 2 (§III.A), we elaborate on the intersection operation, which is conducted via linear search utilizing two pointers on the CPU, while employing binary search on the GPU, where elements in one list act as search keys to determine their existence in the other. As illustrated in Figure 1, intersection operation accounts for an average of 97% of the running time, thus serving as the primary performance bottleneck.

(iii.c) We apologize for the inaccurate statement and have rectified it as "directly using recursion to travel a tree to materialize Basic on GPUs is infeasible".

(iv) The load balancing methods in these papers are orthogonal to ours and employ static methods that allocate different-sized thread groups based on adjacency list lengths. For instance, Gunrock divides the adjacency list into equally sized chunks for processing across different blocks. The other two methods allocate threads, warps, and thread blocks based on the adjacency list size. They still cannot guarantee load balance due to the dynamic nature of the computations [?]. In contrast, we integrate dynamic work stealing to ensure load balance during execution. Furthermore, these methods incur additional overhead. For instance, Gunrock requires adjacency list partitioning, while Merrill et al's method necessitates synchronization before thread group allocation. In contrast, our method efficiently utilizes idle threads for work stealing.

III. Response to Reviewer #2

R2.O1: Elaboration on the applications of biclique counting.

Response R2.O1: Thanks for the suggestion. In this revision, we have incorporated additional details and examples to illustrate how biclique counting can be utilized in various applications in §I (please refer to R1.Required Changes (iii)(a)).

R2.O2: Writing errors.

Response R2.O2: Thanks for pointing this out. In the revised paper, we have corrected this error and meticulously proofread our manuscript to prevent any similar issues from arising.

R2.O3: Notation table.

Response R2.O3: Thanks. In this revision, we have included a notation table in §II.

R2.O4: Expired link to codes/datasets.

Response O4: Thanks. We have provided codes and datasets at https://github.com/ZJU-DAILY/GBC;

IV. Response to Reviewer #3

R3.O1: Feasibility of adapting existing graph systems.

ResponseR3.O1: Thank you for bringing up this point. We acknowledge that existing graph systems, such as Ligra and Gunrock, can be adapted to accommodate (p, q)-biclique counting. However, it's essential to note that their BFS-based graph traversal strategy may lead to substantial memory overhead, as discussed in Section IV. Our experimental results reveal that the memory usage of intermediate results exceeds 40GB at the fourth level of search trees (an early stage with large-scale bicliques), even for the relatively small dataset *GH*. More importantly, Ligra and Gunrock are designed with a broader scope, emphasizing the optimization of programming abstraction and traversal framework for general graph algorithms. As depicted in Figure 1, the bottleneck in (p, q)-biclique counting lies in the intersection operation, which constitutes the primary focus of our optimization efforts.

R3.O2: Connection and motivation of optimizations.

Response R3.O2: Thanks. In the revised paper, we have revised the paper to provide a more thorough explanation of the rationale behind our optimization methods (§IV and §V). Specifically, the DFS-BFS exploration is designed with consideration for GPU memory and thread utilization, while HTB and Border are devised to accelerate intersection computation (bottleneck of biclique counting), the joint load balancing to avoid the laggard in multi-threading.

R3.O3: Clarifying the (p, q)-biclique problem.

Response R3.O3: Thanks. In this revision, we have included an example and explanation of the computation workload to elucidate the (p, q)-biclique problem (Example 2 in §II.A).

R3.O4: Applications of the studied problem.

Response R3.O4: Thanks for the suggestion. In this revision, we have explicitly outlined the application scenarios in which our research findings can be utilized in §I (please refer to R1.Required Changes (iii)(a)).

R3.O5: Detailing experimental settings.

Response R3.O5: Thanks. In this revision, we have endeavored to provide comprehensive explanations of the experimental settings, including a detailed explanation of data generation, as well as information on their degree distribution in §VII.A. The synthetic datasets are generated as follows: (1) fix the size of U and V, (2) determine the number of 2-hop neighbors for vertices 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.

R3.D1: Writing errors.

Response R3.O5: Thanks for pointing this out. In the revised paper, we have rectified this error and conducted a thorough proofreading of our manuscript to prevent any similar issues.