Advanced Maximal Biclique Enumeration on GPUs Using Bitmaps

Zhe Pan, *Member, IEEE*, Shuibing He, *Member, IEEE*, Xu Li, Xuechen Zhang, *Member, IEEE*, Rui Wang, Yanlong Yin, and Gang Chen, *Member, IEEE*

***Abstract*—Maximal biclique enumeration (MBE) in bipartite graphs is an important problem in data mining with many real-world applications. Parallel MBE algorithms for GPUs are needed for MBE acceleration leveraging its many computing cores. However, enumerating maximal bicliques using GPUs has three main challenges including large memory requirement, thread divergence, and load imbalance. In this paper, we propose GMBE+, an advanced GPU solution for the MBE problem. To overcome the challenges, we design (1) a node-reuse approach to reduce GPU memory usage with advanced node pruning, (2) a bitmap-based set intersection approach to minimize thread divergence, and (3) a load-aware task scheduling framework to achieve load balance among threads within GPU warps, facilitated by a novel set union approach. Our experiments reveal that GMBE+ is 1.2× faster than the latest GPU-based MBE algorithm GMBE on average when running on the same NVIDIA A100 GPU.**

***Index Terms*—Graph computing, maximal biclique enumeration, GPU.**

I. Introduction

A *bipartite graph* $G = (U, V, E)$ contains two disjoint vertex sets U and V, where an edge $e \in E$ only occurs between two vertices in U and V, respectively. A *biclique* is a complete bipartite graph, i.e., there exists an edge between two vertices if and only if the two vertices are in different vertex sets. A *maximal biclique* in G is a subgraph of G, which is a biclique and can not be further enlarged to form a larger biclique. Maximal biclique enumeration (MBE) aims to find all maximal bicliques in G.

Received 29 July 2024; revised 31 March 2025; accepted 25 April 2025. Date of publication 5 May 2025; date of current version 11 July 2025. This work was supported in part by the National Science Foundation of China under Grant 62172361, in part by the National Key Research and Development Program of China under Grant 2023YFB4502100, in part by the Major Projects of Zhejiang Province under Grant LD24F020012, in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Grant 2023WNLOKF005, and in part by the Pioneer and Leading Goose R&D Program of Zhejiang Province under Grant 2024SSYS0002. Recommended for acceptance by C. Chen. *(Corresponding author: Shuibing He.)*

Zhe Pan, Shuibing He, Xu Li, Rui Wang, Yanlong Yin, and Gang Chen are with the State Key Laboratory of Blockchain and Data Security, and the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310051, China (e-mail: panzhe@zju.edu.cn; heshuibing@zju.edu.cn; fhxu@zju.edu.cn; rwang21@zju.edu.cn; yinyanlong@zju.edu.cn; cg@zju.edu.cn).

Xuechen Zhang is with the School of Engineering and Computer Science, Washington State University Vancouver, Vancouver, WA 98686 USA (e-mail: xuechen.zhang@wsu.edu).

Digital Object Identifier 10.1109/TC.2025.3566898

MBE is important in bipartite graph analysis, with widespread applications, such as anomaly detection in e-commerce networks [1], [2], social recommendation in social networks [3], gene expression analysis in expression datasets [4], and GNN information aggregation [5]. Consider an example in an e-commerce network, where the purchasing relationships can be modeled by a bipartite graph. It is suspicious for a large group of customers to buy a set of products together because online sellers are likely to make fake purchases through illegal platforms to improve their credibility and positive ratings [1], [2]. We can identify all these suspicious groups by enumerating all the maximal bicliques in the network, and then detect them.

Over the past few decades, many MBE algorithms have been proposed to speed up the enumeration of all maximal bicliques in bipartite graphs [3], [4], [6], [7], [8], [9], [10], [11], [12]. A mainstream approach is to use the set enumeration tree [13] to recursively enumerate all candidate subgraphs, and then judge whether they are maximal bicliques. The enumeration space of this method is a powerset of V or U, so the computational overhead is very high [14], especially for large graphs. Thus, many efforts are made to reduce the enumeration space using pruning [8], [9] and vertex ordering [4], [9]. Recent work accelerates enumeration node checking using the prefix tree [10]. However, they only achieve limited speedup for not exploring the parallelism of multi-core CPUs. Other works design parallelization strategies for the multi-core CPUs [7]. However, their performance speedup is constrained by the limited parallelism of CPUs.

Recent efforts speed up MBE with GPUs [15], [16], but they still face three major challenges. First, the existing MBE algorithms require large memory space and frequent memory allocations to store the intermediate data of each enumerated subgraph, thus they cannot efficiently run on GPUs with limited memory capacity and high dynamic memory allocation overhead [17], due to the severe shortage of memory resources.

Second, the performance of existing MBE algorithms suffers from irregular computation [18] while GPUs are suitable to perform regular computation with high parallelism. Specifically, different GPU threads in the same warp in the MBE algorithm may take different execution paths to access different vertex neighbors, causing the thread divergence problem [19]. GPUs will serialize these diverged thread operations [20], resulting in low thread utilization and poor memory access efficiency.

Lastly, existing MBE algorithms have a serious load imbalance problem on GPUs. The reason is that the maximal biclique sizes are varied significantly with real-world graphs for the power-law distribution of vertex degrees. As a result, threads assigned to each GPU core have different running times. When load imbalance happens, thousands of GPU threads have to wait for the slowest one to complete, causing degraded GPU performance.

Many existing studies have used GPUs to improve the performance of other graph enumeration problems, like maximal clique enumeration [21] and graph pattern mining [19]. The optimizations include data graph partitioning [22], two-level parallelism [23], adaptive buffering [24], hybrid order on GPU [19], etc. However, they are inefficient for MBE on GPUs, because the enumerated subgraphs for MBE generally contain much more vertices than those in other graph enumeration problems. For instance, it may enumerate maximal bicliques comprising several to thousands of vertices, while the triangle counting algorithm only considers subgraphs with three vertices. The larger subgraphs generated at runtime require larger memory and computation costs and lead to more serious problems of large memory requirement, thread divergence, and load imbalance mentioned above.

To address all the challenges and achieve advanced maximal biclique enumeration on GPUs, we design the GPU-based MBE algorithm (GMBE+) considering the characteristics of GPU architecture and MBE computation pattern. Specifically, first, by replacing recursion with iteration, we not only optimize memory usage by reusing the memory space of enumeration nodes but also enhance performance through node pruning based on intermediate results, such as local neighborhood sizes. Second, we utilize bitmaps to store vertex neighbors in cases where the current node contains a small number of vertices. This approach efficiently minimizes thread divergence by simplifying set intersections and accelerating them using bitwise operations between same-sized bitmaps. Finally, GMBE+ carefully manages the size of subtrees assigned to GPU threads and schedules the tasks using two-level queues to achieve load balance within GPU warps and blocks leveraging persistent thread programming models for GPUs. Notably, we devise a warp-wide intersect-path-based set union approach to help balance the workload. Compared to GMBE [15] (the conference version), GMBE+ further expedites set intersections using bitmaps and optimizes set unions based on Intersect Path (IP) [25].

We adopt the fast CUDA primitives [26] to implement the GMBE+ prototype and conduct extensive experiments on real-world datasets to demonstrate the efficiency of GMBE+. Our experimental results show that GMBE+ is 1.2× faster than the latest GPU-based MBE algorithm GMBE (the conference version) on average when run on the same NVIDIA A100 GPU.

The paper is organized as follows: Section II provides background on baseline MBE solutions, graph representations, and GPU architecture. Section III describes the challenges of MBE on GPUs, and Section IV presents our corresponding design solutions. Section V details the implementation of GMBE+. Section VI evaluates our design. Section VII discusses related work, and Section VIII concludes.

II. Background

In this section, we briefly review the recent MBE algorithms on CPUs, graph representations, and introduce the compute unified device architecture (CUDA) in modern GPUs.

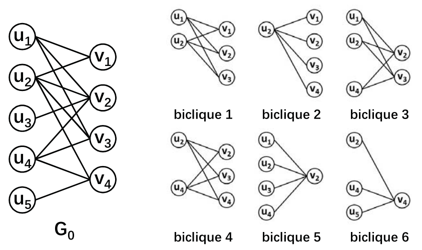


Fig. 1. A bipartite graph G_0 containing 6 maximal bicliques.

A. MBE Algorithms on CPUs

Notation definitions. Given a *bipartite graph* $G(U,V,E)$. U and V are two disjoint vertex sets in G. E is the edge set in G and $E \subseteq U \times V$. For each vertex u in U, $N(u)$ denotes the neighbors of u, i.e., $N(u) = \{v|(u,v) \in E\}$. $N_2(u)$ denotes the 2-hop neighbors of u, i.e., $N_2(u) = \cup_{v\in N(u)} N(v) - \{u\}$. For a vertex set X, $\Gamma(X)$ denotes the common neighbors of vertices in X, i.e., $\Gamma(X) = \cap_{u\in X} N(u)$. $\Delta(X)$ is the maximum degree of vertices in X, i.e., $\Delta(X) = \max_{u\in X}|N(u)|$. $\Delta_2(X)$ is the maximum 2-hop degree of vertices in X, i.e., $\Delta_2(X) = \max_{u\in X}|N_2(u)|$. We have a symmetrical definition for each vertex v in V. A *biclique* is a vertex set pair (L,R) in G s.t., $L \subseteq U$ and $R \subseteq V$ and $\forall u \in L$, $v \in R, (u,v) \in E$. A biclique (L,R) is a *maximal biclique* if there is no biclique (L',R') such that $(L\cup R) \subset (L' \cup R')$. Fig. 1 shows all maximal bicliques in a bipartite graph G_0.

Problem statement. The MBE problem aims to enumerate all maximal bicliques in a bipartite graph.

Baseline solution. To solve the MBE problem, recent works [3], [4], [6], [7], [8], [9], [10] **recursively** run a backtracking procedure to generate the powerset of V using a set enumeration tree [13] and then obtain all maximal bicliques correspondingly. In most works [3], [6], [7], [8], each enumeration node is represented as a 3-tuple (L,R,C), where $L \subseteq U$ and $R, C \subseteq V$. R and C are disjoint and used to generate the powerset of V. R stores the current subset of V, while C stores the candidate vertices for expanding R. (L,R) is the corresponding biclique where $L = \Gamma(R)$. In the following, we show the basic recursive procedure for each enumeration node in existing works using Algorithm 1. Then, we detail Algorithm 1 with an example.

The procedure starts at a root node initialized as $(U,\emptyset,V)$. In each enumeration node, each candidate vertex $v_c \in C$ is traversed sequentially (line #2) to generate a new biclique (L',R') (line #3). The parent node allows expanding R with untraversed candidate vertices in C (lines #5,6). The new candidate set C' contains all vertices in $C - R'$ that connect with any vertex in L' (lines #7,8). The new biclique is maximal if and only if R' is equal to $\Gamma(L')$ (line #9). All maximal bicliques are enumerated exactly once (line #10). The procedure recursively generates new nodes in a depth-first search (DFS) manner to enumerate all maximal bicliques (line #11). After processing a new node, we remove v_c from the current candidate set C (line #12). Besides, some works [4], [9] use a set Q to keep traversed candidate vertices for accelerating the node checking in line #9. Similar to many existing works [3], [6], [7], [8], we omit the set Q to reduce the memory consumption.

Algorithm 1: Recursive MBE Algorithm.

```
   Data: Bipartite graph $G(U,V,E)$
   Input: Set $L \subseteq U$, disjoint sets $R, C \subseteq V$
   Output: All maximal bicliques
1  procedure recur_search($L, R, C$):
2      foreach $v' \in C$ do
3          $L' \leftarrow L \cap N(v')$; $R' \leftarrow R$; $C' \leftarrow \emptyset$;
4          foreach $v_c \in C$ do        // Node generation
5              if $L' \cap N(v_c) = L'$ then
6                  $R' \leftarrow R' \cup \{v_c\}$;
7              else if $L' \cap N(v_c) \neq \emptyset$ then
8                  $C' \leftarrow C' \cup \{v_c\}$;
9          if $R' = \Gamma(L')$ then           // Node check
10             Output($L', R'$) as a maximal biclique;
11             recur_search($L', R', C'$) ;
12         $C \leftarrow C \setminus \{v'\}$;
```

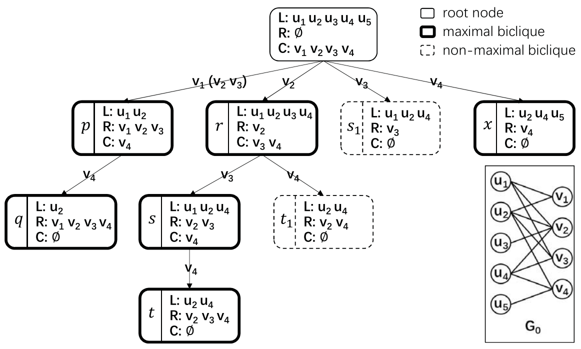

L: u1 u2 u3 u4 u5
R: ∅
C: v1 v2 v3 v4
root node
maximal biclique
non-maximal biclique
v1 (v2 v3)
v2
v3
v4
p L: u1 u2 R: v1 v2 v3 C: v4
r L: u1 u2 u3 u4 R: v2 C: v3 v4
s1 L: u1 u2 u4 R: v3 C: ∅
x L: u2 u4 u5 R: v4 C: ∅
v4
v3
v4
q L: u2 R: v1 v2 v3 v4 C: ∅
s L: u1 u2 u4 R: v2 v3 C: v4
t1 L: u2 u4 R: v2 v4 C: ∅
v4
t L: u2 u4 R: v2 v3 v4 C: ∅
u1 u2 u3 u4 u5
v1 v2 v3 v4
G0

Fig. 2. An enumeration tree for bipartite graph G_0.

Example 2.1: Fig. 2 depicts an enumeration tree for a bipartite graph G_0 using Algorithm 1. The vertices on the edge between two nodes are used to expand R of the new node, including a traversed vertex v_c and the other untraversed candidate vertices in parenthesis. For presentation convenience, we always use the subscript to denote the corresponding vertex set of an enumeration node. For instance, L_p denotes L set of node p.

We start from the root node and recursively search the subspaces in a DFS manner by traversing candidate vertices in C following a pre-imposed order. By traversing v_1, we enter node p. We know $L_p = L_{root} \cap N(v_1) = \{u_1, u_2, u_3, u_4, u_5\} \cap \{u_1, u_2\} = \{u_1, u_2\}$. We then expand R_p with candidate vertices v_1, v_2, and v_3 because $\Gamma(L_p) \cap C_{root} = \{v_1, v_2, v_3\} \cap \{v_1, v_2, v_3, v_4\} = \{v_1, v_2, v_3\}$. Node p generates a maximal biclique (L_p, R_p) because $R_p = \Gamma(L_p)$. v_4 is in the new candidate vertex set C_p because $L_p \cap N(v_4) = \{u_1, u_2\} \cap \{u_2, u_4, u_5\} = \{u_2\} \neq \emptyset$. Continuing this process, the root node can generate node s_1 by traversing v_3. We then know $L_{s_1} = \{u_1, u_2, u_4\}$ and $R_{s_1} = (\Gamma(L_{s_1}) \cap C_{root}) \cup R_{root} = \{v_3\}$. Compared to node s, node s_1 generates a non-maximal biclique because the parent node of node s_1 (i.e., the root node) fails to expand R_{s_1} with vertex v_2 since v_2 has been traversed by root node to generate node r. Other nodes can be generated similarly as shown in the figure.

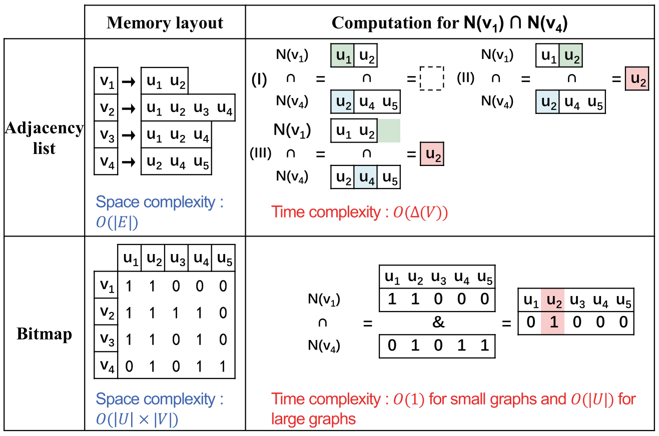


Fig. 3. Comparison of adjacency list and bitmap representations for bipartite graph $G_0(U,V,E)$ in Fig. 1. Adjacency list computes $N(v_1) \cap N(v_4)$ in 3 steps by sequentially comparing vertex neighbors, whereas the bitmap computes it using a bitwise and (&) operation.

Recent optimizations. To reduce the computational overhead, researchers mainly focus on reducing the enumeration space, using various vertex ordering, pruning approaches [4], [8], [9], and the prefix tree [10]. To achieve further speedup, other works parallelizes MBE algorithms on multicore CPUs [7] or distributed architectures [6]. Specifically, they distribute all vertices v in V across CPU threads, and each thread generates a subtree correspondingly using 1-hop and 2-hop neighbors of v. However, their performance speedup is constrained by the limited parallelism of CPUs. Recent studies [15], [16] accelerate MBE with GPUs, but their performance declines due to challenges outlined in Section III.

B. Graph Representations

Graph representations significantly impact MBE algorithms, especially set intersection efficiency (e.g., lines #3,5,7,9 in Algorithm 1). Fig. 3 depicts two popular representations for bipartite graphs. The adjacency list is favored in recent works for its minimal storage size (in $O(|E|)$) [8], [9], [10], [15]. However, each set intersection entails $O(\Delta(V))$ time for comparing two adjacency lists sequentially. The Bitmap enables fast set intersection via bitwise AND operations (i.e., $O(1)$ for small graphs) but demands excessive memory for large graphs (i.e., $O(|U| \times |V|)$). Therefore, we need to make a trade-off between memory usage and execution time when determining an optimal graph representation in memory.

C. GPU Architecture and Programming

GPU architecture. Fig. 4 shows the general architecture of a modern GPU. A GPU generally consists of a global memory, a shared L2 cache, and multiple streaming multiprocessors (SMs). Each SM contains an individual L1 cache, a programmable multi-bank shared memory, and multiple computing cores. A modern GPU can be equipped with thousands of lightweight cores in total, thus providing massive computing power. Unlike computing resources, memory resources on the GPU are relatively limited. For example, the recently popular NVIDIA A100

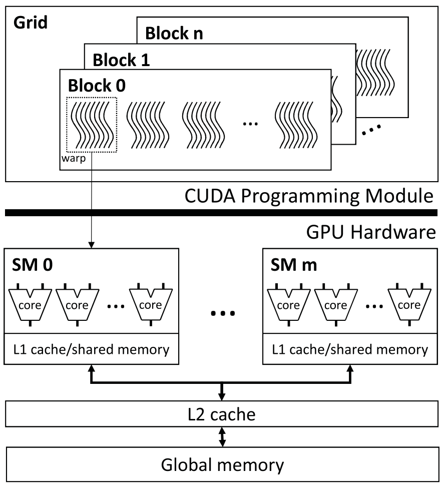


Fig. 4. GPU architecture.

[27] can provide up to 6,912 cores, but only up to 80 GB of memory.

CUDA programming model. The CUDA (Compute Unified Device Architecture) programming model [20] provides a parallel computing platform and a set of APIs that allows users to efficiently use GPUs for general-purpose processing. It adopts the SIMT (Single instruction, multiple threads) execution model [28] to manage numerous threads. CUDA divides GPU kernels into multiple grids, each consisting of multiple blocks. A block includes multiple threads and is assigned to an SM during execution. The SM groups 32 parallel threads into a warp and executes multiple warps concurrently. For each warp, all threads execute one common instruction at a time. Thus, thousands of GPU cores can efficiently work in parallel to achieve high performance.

GPU usage guidelines. There are several notable guidelines for improving the efficiency of GPU programs. First, dynamic memory allocations on GPUs are expensive because many threads may allocate new memory at the same time, causing problems such as thread contention, synchronization overhead, and memory fragmentation [17]. Hence, we should manage the valuable GPU memory wisely and avoid frequent memory allocations. Second, we should minimize the thread divergence, i.e., threads in a warp take different execution paths on the GPU. Since GPU needs to serialize these paths, thread divergence can severely degrade the execution performance [20]. Third, we should pay more attention to the load balancing among multiple cores on the GPU, because thousands of cores have to wait for the slowest thread to run on a lightweight core, which is costly.

III. Challenges of MBE on GPUs

A naive approach to performing MBE on the GPU is to divide the whole enumeration tree into multiple subtrees and assign each subtree to an individual SM for execution. For example, for the enumeration tree in Fig. 2, we divide it into four subtrees, then assign SM 0 to conduct the execution for enumeration nodes p and q, and assign SM 1 to conduct the execution for enumeration nodes r, s, t, and t_1, and so on. However, this naive approach faces the following three problems, thus making MBE on GPUs challenging.

A. Large Memory Requirement

MBE algorithms usually allocate memory for new enumeration nodes (lines #3-8 in Algorithm 1), which is expensive on GPUs as mentioned in Section II-C. To achieve high performance, a typical approach is to pre-allocate enough memory space on GPUs to accommodate all enumeration nodes before execution. In this case, each enumeration node requires $O(|L|+|R|+|C|)$ memory bounded by $O(\Delta(V)+\Delta_2(V))$, and each subtree activates at most $\Delta(V)$ nodes at the same time for backtracking. Thus, the total amount of memory space that needs to be pre-allocated for each subtree traversal procedure is $\Delta(V) \times (\Delta(V)+\Delta_2(V)) \times \mathbf{sizeof}(vertex)$. For example, when using the NVIDIA A100 GPU with 40 GB memory to perform MBE on a real-world bipartite graph BookCrossing [29], the memory requirement for each subtree traversal procedure is $13{,}601 \times (13{,}601+53{,}915) \times \mathbf{sizeof(int)}$ B $= 3.67$ GB. We need more than 108×3.67 GB $= 397$ GB memory to fully utilize the 108 SMs, which exceeds the maximum memory space on the GPU (i.e., 40 GB), thus facing a severe memory shortage.

B. Massive Thread Divergence

As mentioned in Section II-C, thread divergence can significantly degrade thread performance on GPUs. In the case of MBE on GPUs, thread divergence mainly occurs when irregularly accessing neighborhoods of vertices on adjacency lists. Recent MBE algorithms typically represent bipartite graphs using adjacency lists due to their memory efficiency, as discussed in Section II-B. However, in this scenario, threads within the same warp may access neighborhoods of different vertices, and each neighborhood may contain a different number of vertices. This leads to each thread accessing different memory areas and executing different control flows. Therefore, it's crucial to develop a GPU-friendly approach to ensure regularized neighborhood accesses for each thread.

C. Load Imbalance

The parallel MBE algorithm is likely to generate severe imbalanced workloads on GPUs for two main reasons. First, the running time for processing each enumeration node in the enumeration tree varies greatly since nodes contain various numbers of candidate vertices. Second, the number of nodes in subtrees differs significantly because different maximal bicliques (L,R) contain various numbers of vertices in R. The enumeration tree grows as R increases as shown in Algorithm 1. As a result, most cores in the GPU will spend a large portion of time waiting for the processing of the largest enumeration tree, which aggravates the load imbalance. Related graph pattern mining algorithm G^{2}Miner [19] always assigns each enumeration tree to a warp in the GPU. However, Fig. 5 shows that if we assign each enumeration tree to a warp based on our new algorithm GMBE+, over 74% of SMs (i.e., 80 SMs / 108 SMs) waste 80% of running time (i.e., 217s / 270s) waiting for the slowest one on the

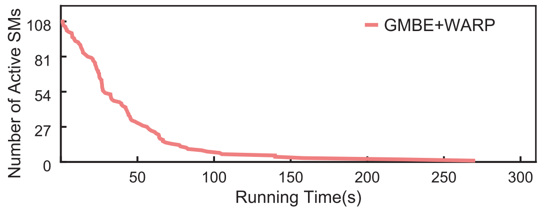


Fig. 5. Load imbalance for the MBE problem. 80 SMs waste over 217 seconds waiting for the slowest one on BookCrossing if we assign each enumeration tree to a warp based on our new algorithm GMBE+ as shown in Section VI-C.

BookCrossing dataset. It is necessary to balance workloads for MBE in a finer granularity.

IV. Design of GMBE+

We design a GPU-based MBE algorithm (GMBE+) with three novel techniques. (1) We develop an iterative MBE algorithm to replace the original recursive approach. It reduces memory usage by reusing the enumeration nodes and conducts node pruning based on local neighborhood sizes. (2) It enables bitmap-based set intersections to minimize thread divergence. And (3) it supports load-aware task scheduling to balance workloads among threads in warps and blocks. We propose a warp-wide intersect-path-based set union approach to aid in load balancing. We describe them in detail and analyze GMBE+ in depth.

A. Iteration With Node Reuse

1) Iterative MBE Algorithm With Node Reuse: To reduce memory usage, we propose an iterative MBE algorithm with node reuse. The key idea is that we can reuse a node x to represent its child nodes with additional metadata stored in the node x. By doing this, we save the memory space allocated for the child nodes of x. We can efficiently support node reuse because during enumeration the $L \cup R \cup C$ of the child nodes of x is always a subset of the $L \cup R \cup C$ of its parent node x. For instance, vertices $\{u_2, v_1, v_2, v_3, v_4\}$ in child node q ($\{u_2\}, \{v_1, v_2, v_3, v_4\}, \emptyset$) is the subset of vertices $\{u_1, u_2, v_1, v_2, v_3, v_4\}$ in its parent node p ($\{u_1, u_2\}, \{v_1, v_2, v_3\}, \{v_4\}$) as shown in Fig. 2. We detail this iterative algorithm in Algorithm 2.

Specifically, instead of dynamically creating and freeing nodes for recursions in Algorithm 1, we **replace recursions** (line #11 in Algorithm 1) **with iterations** (lines #2-5, 7, 16 in Algorithm 2) and explicitly manage nodes with a *stack-like structure* `node_buf`. The `node_buf` structure and its update process can refer to Fig. 6. A `node_buf` consists of a root node (L^*, R^*, C^*), the attribute *depth* of each vertex in $L^* \cup R^* \cup C^*$, and the *traversed vertices* from the root node to the current node. The *depth* of each vertex is updated according to the depth of the current node (i.e., the number of ancestor nodes of the current node). We can apply the node reuse strategy on `node_buf` by actively updating the *depth* field, and backtrack to ancestor nodes using the *traversed vertices* field. In this way, GMBE+ can derive all nodes in a fixed memory region during iteration, minimizing the need for dynamic allocation in GPUs. To minimize overhead, at the start and end of each node

Algorithm 2: Iterative MBE Algorithm.

```
Data: Bipartite graph $G(U, V, E)$
Input: Set $L^* \subseteq U$, disjoint sets $R^*, C^* \subseteq V$
Output: All maximal bicliques
1  procedure iter_search($L^*, R^*, C^*$):
2      $node\_buf$.init_and_push(($L^*, R^*, C^*$));
3      while $node\_buf$ is not empty do    // Iteration
4          $(L, R, C) \leftarrow node\_buf$.pop() ;
5          if $C$ is not empty then
6              $v' \leftarrow$ the smallest vertex in $C$ ;
7              $node\_buf$.push(($L, R, C \setminus \{v'\}$));
8              $L' \leftarrow L \cap N(v')$; $R' \leftarrow R$; $C' \leftarrow \emptyset$;
9              foreach $v_c \in C$ do
10                 if $L' \cap N(v_c) = L'$ then
11                     $R' \leftarrow R' \cup \{v_c\}$;
12                 else if $L' \cap N(v_c) \neq \emptyset$ then
13                     $C' \leftarrow C' \cup \{v_c\}$;
14             if $R' = \Gamma(L')$ then
15                 Output($L', R'$) as a maximal biclique;
16                 $node\_buf$.push(($L', R', C'$)) ;
```

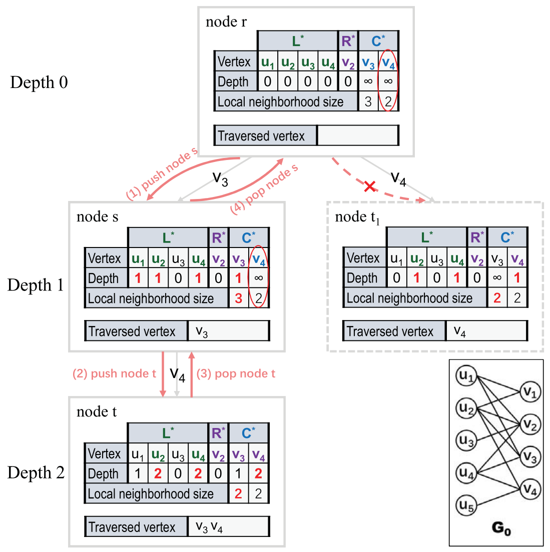


Fig. 6. Illustration of GMBE+ with pruning using local neighborhood size.

computation, we batch-check and update the 'depth' attribute with a complexity of $O(|L| + |R| + |C|)$, since each vertex is accessed once. This cost is negligible compared to the $O(|L| \times (|R| + |C|))$ complexity, which arises from computing the intersections between L and the neighbors of vertices in R and C. Next, we illustrate the key functions designed for node reuse.

- **init_and_push**$((L^*, R^*, C^*))$: this function is used for creating and initializing a `node_buf` using the root node (L^*, R^*, C^*) of a subtree (line #2 in Algorithm 2). The `node_buf` stores all vertices in $L^* \cup R^* \cup C^*$ and actively tracks the *depth* for each vertex. We initialize the *depth* for vertices in $L^* \cup R^*$ to 0 and the *depth* for vertices in C^* to ∞.
- **push**$((L', R', C'))$: we use this function to reuse the `node_buf` of a parent node to generate a child node (line #16 in Algorithm 2). When we push a new node (L', R', C') at

depth D, we first update the *depth*s of vertices in L^* to D if they are also in L', and then update the *depth*s of vertices in C^* to D if they are also in R' and their current *depth*s are ∞. Therefore, the new (L', R', C') can be found in the original (L^*, R^*, C^*) by the *depth* field, as L' contains all vertices in L^* whose *depth* is equal to D, and R' contains all vertices in $R^* \cup C^*$ whose *depth* is not greater than D, and C' contains all vertices in C^* whose *depth* is ∞. Finally, we append the *traversed vertices* with the chosen vertex v'.

- **pop**(): we use this function to get the current node (L, R, C) and then backtrack to its parent node using `node_buf` (line #4 in Algorithm 2). When we pop a node (L, R, C) at depth D, we first remove the latest traversed vertex v' in `node_buf`. Then, we update the *depth* for vertices in L to $D-1$ and update the *depth* for vertices in C with *depth* D to ∞.

Compared to the original node structure which only tracks the (L, R, C), our proposed `node_buf` tracks more information with more memory consumption for a single node, which is bounded by $3 \times \Delta(V) + 2 \times \Delta_2(V)$. Whereas, a single `node_buf` is adequate to be reused for all enumeration nodes for running a subtree traversal procedure shown in Algorithm 2, which significantly reduces the memory requirement and enables running thousands of MBE procedures on GPUs in parallel concurrently. For instance, an A100 GPU of 40 GB memory is adequate to run over 10k of subtree traversal procedures on the BookCrossing dataset [29] because each procedure requires only $(3 \times 13{,}601 + 2 \times 53{,}915) \times$ **sizeof**(int) B $= 595$ KB. Compared to the naive implementation discussed in Section III-A which requires $13{,}601 \times (13{,}601 + 53{,}915) \times$ **sizeof**(int) B $= 3.67$ GB, this node reuse approach saves 6,178× memory space for running each MBE procedure on BookCrossing.

2) Pruning Using Local Neighborhood Size: To enhance the efficiency, we propose a GPU-friendly pruning approach. Specifically, given a node (L, R, C), we define **local neighbors** of a vertex $v \in V$ as $N_L(v)$, where $N_L(v)$ is equal to $N(v) \cap L$. We further define *local neighborhood size* for a vertex v as the number of local neighbors of v. We can obtain local neighborhood sizes for candidate vertices without additional overhead because they are intermediate results for computing the candidate set (line #12 in Algorithm 2).

An interesting observation is that we can safely prune nodes generated by vertices whose local neighborhood size does not change after traversing any of its child nodes. More formally, suppose node p has traversed vertex v_q to generate node q and is now traversing vertex v_x to generate node x. If the local neighborhood sizes of vertex v_x are identical in nodes p and q, denoted as $|L_p \cap N(v_x)| = |L_q \cap N(v_x)| = |N(v_q) \cap L_p \cap N(v_x)|$, then we conclude that $L_x = L_p \cap N(v_x) \subseteq N(v_q)$ and v_q belongs to $\Gamma(L_x)$. As v_q has been traversed and cannot be added to R_x, we can confidently prune node x since it will fail the node checking at line #14 in Algorithm 2. Thus, we further optimize Algorithm 2 by maintaining *local neighborhood sizes* for all candidate vertices in `node_buf`. We can then prune useless candidates if their local neighborhood sizes do not change after popping a traversed child node. We use an example to demonstrate the application of this pruning approach.

Example 4.1: Fig. 6 depicts that GMBE+ iteratively enumerates all maximal bicliques in the subtree rooted by node r in Fig. 2 with the fixed memory in `node_buf`. Specifically, GMBE+ initializes `node_buf` using node r, where $L^* = L_r$, $R^* = R_r$, and $C^* = C_r$. `Node_buf` then initializes the depth for vertices in $L^* \cup R^*$ to 0 and initializes the depth for vertices in C^* to ∞. `Node_buf` initializes the local neighborhood size (i.e., $|N_L|$) for vertices in C^* according to the definition. For instance, $|N_L(v_3)|$ at node r is $|N(v_3) \cap L_r| = |\{u_1, u_2, u_4\} \cap \{u_1, u_2, u_3, u_4\}| = 3$.

Next, we generate node s by traversing v_3. We know $L_s = L_r \cap N(v_3) = \{u_1, u_2, u_4\}$. The depth of node s is 1, `node_buf` updates the depth for u_1, u_2, u_4, and v_3 to 1 because they belong to $L_s \cup R_s$. The depth for other vertices (i.e., u_3 and v_4) in L^* and R^* remains 0. The depth for v_4 in C_s remains as ∞.

We then update the local neighborhood size for v_3 and v_4 using L_s. We know $|N_L(v_3)| = |N(v_3) \cap L_s| = |\{u_1, u_2, u_4\} \cap \{u_1, u_2, u_4\}| = 3$. $|N_L(v_4)| = |N(v_4) \cap L_s| = |\{u_2, u_4, u_5\} \cap \{u_1, u_2, u_4\}| = 2$. We can generate other nodes similarly.

After processing the subtree rooted by node s, `node_buf` pops node s and recovers the parent node r. `node_buf` resets depth for u_1, u_2, and u_4 to 0 and resets the depth for v_3 to ∞ because their original depth is the same as the depth of node s. GMBE+ proactively prunes node t_1 by removing useless candidate vertex v_4 at node r because the local neighborhood size (i.e., 2) for v_4 does not change after popping node s.

B. Bitmap-Based Set Intersections

To minimize thread divergence, we introduce a bitmap-based set intersection approach. The key idea is that we directly store local neighborhoods of vertices using bitmaps when the current node holds only a small number of vertices. Instead of allocating memory for these neighborhoods, we carefully reuse memory in `node_buf`. This approach supports set intersections using bitwise operations between same-sized bitmaps, effectively reducing thread divergence. Consequently, different threads consistently follow the same control flow when performing set intersections.

Specifically, when the size of L in the current node (L, R, C) is below our defined threshold τ, we replace the *local neighborhood size* field in `node_buf` with multiple $|L|$-bit *bitmap*s and reuse these bitmaps for descendant nodes. Each bitmap illustrates the connection between a vertex v_c in C^* and all vertices in L, representing the local neighborhood of v_c. By storing these neighborhoods in bitmaps, we enable fast set intersections through bitwise AND (&) operations in lines #8, 10, 12 in Algorithm 2, with each set intersection taking $O(\tau)$ time. We practically set the threshold τ to 32 so that each set intersection requires only a single bitwise AND between two 32-bit integers in $O(1)$ time (See Section VI-D). Additionally, we proactively prune nodes generated by vertices whose local neighborhoods, in bitmap format, remain unchanged after traversing any child nodes, as demonstrated in Section IV-A2.

Example 4.2: Fig. 7 shows how GMBE+ employs bitmap-based set intersections. In comparison to Fig. 6, we only replace the local neighborhood size field in `node_buf` with the

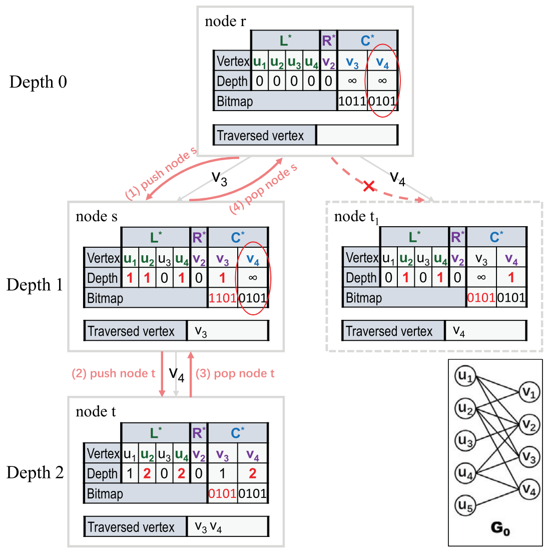


Fig. 7. Illustration of GMBE+ with bitmap-based set intersections.

Algorithm 3: Naive task scheduling on GPUs.

```
  foreach $v_s \in V$ do          // Initialize |V| tasks
1     $L_s \leftarrow N(v_s); R_s \leftarrow \{v_s\}; C_s \leftarrow \emptyset$;
2     foreach $v_c \in N_2(v_s)$ do
3         if $L_s \cap N(v_c) = L_s$ then
4             $R_s \leftarrow R_s \cup \{v_c\}$;
5         else if $v_c$ is with later order than $v_s$ then
6             $C_s \leftarrow C_s \cup \{v_c\}$;
7     if $v_s$ is the smallest vertex in $R_s$ then
          // Task Mapping
8         iter_search($L_s, R_s, C_s$) ;
```

bitmap field. Each bitmap represents the local neighborhood of a vertex. For instance, in node t, the local neighborhood of v_3 is determined by $L_t \cap N(v_3) = \{u_2, u_4\} \cap \{u_1, u_2, u_4\} = \{u_2, u_4\}$, corresponding to the second and fourth vertices in $L^* = \{u_1, u_2, u_3, u_4\}$. Thus, the bitmap for v_3 in node t is 0101, and we can derive other bitmaps similarly. Besides, node r can proactively prune node t_1 as the bitmaps for v_4 remain unchanged after node s is accessed.

C. Load-Aware Task Scheduling

1) Task-Centric Scheme: For exploring the massive parallelism of GPUs, a naive approach is assigning a task to manage each enumeration tree whose root node is v_s ($v_s \in V$). We show the naive approach in Algorithm 3. Then we map these tasks to warps [19] or blocks [30] in GPUs. We denote these schemes as the *warp-centric scheme* and the *block-centric scheme*, respectively. Our research shows that the naive approach is insufficient to balance the loads among GPU SMs for the MBE problem because the parallel GPU tasks created at line #8 in Algorithm 3 are highly unbalanced and their loads are determined by the various sizes of the enumeration trees assigned to the tasks. Specifically, the slowest tasks running in a warp or a block frequently block other tasks, resulting in up to 97.8% performance degradation on the EuAll dataset. More results can be found in Fig. 13 in Section VI-B.

To address this issue, we propose a **load-aware task-centric scheme** for MBE using the persistent thread (PT) [31] programming model for GPUs. Specifically, we create thread groups, each of which consists of multiple warps. Each thread group is mapped to a GPU SM. We denote the number of warps on each SM as `WarpPerSM` and will discuss its impact on system performance in Section VI-D. We develop a GPU kernel to create load-aware tasks. Any tasks of processing larger enumeration trees are divided into smaller tasks recursively at runtime. The load-aware tasks are then added to a global structure *SM_task_queue* for each SM. When a task is finished, the software scheduler of PT dequeues a task from SM_task_queue and executes `iter_search()` on its corresponding SM. We denote it as the *task-centric scheme*.

The key question to answer is how to detect tasks having heavier loads than others. We use the tree heights and the total number of nodes in a tree. Specifically, the height of an enumeration tree with the root node (L, R, C) is $\min\{|L|, |C|\}$. The number of nodes of the tree can be estimated as $\min\{|L|, |C|\} \times |C|$, where $|C|$ indicates the maximum number of child nodes for each node in the enumeration tree. We empirically set two thresholds including *bound_height* and *bound_size*. Only when $\min\{|L|, |C|\}$ is larger than *bound_height* and $\min\{|L|, |C|\} \times |C|$ is larger than *bound_size*, we divide the task into multiple subtasks to achieve better load balance.

Putting them together. Algorithm 4 describes GMBE+, which is a load-aware task-centric algorithm for GPUs. When SM_task_queue is not empty, it obtains a node from the queue (lines #3-4). If the size of the enumeration tree generated from the node is within bounds (line #16), GMBE+ directly launches a GPU task (line #27). If the size of the enumeration tree is larger than the bounds, it enqueues the root node of sub-trees to be enumerated (lines #17-25). These nodes will be processed later when they are dequeued. If SM_task_queue is empty, it will use *processing_v* to obtain the current v_s that needs to be processed (line #6). Then, it generates a node (L, R, C) (lines #9-14) and launches its corresponding tree onto GPUs.

2) Warp-Wide IP-Based Set Union Approach: Compared to Algorithm 1, Algorithm 4 necessitates the computation of 2-hop neighbors of vertices (line #10), which can be done using set unions since $N_2(u) = \cup_{v \in N(u)} N(v) - \{u\}$. To accelerate set unions, inspired by intersect-path(IP)-based set intersection on GPUs [25], we introduce a warp-wide IP-based set union on GPUs. The key idea is to give each thread its own assignment, allowing us to use many threads at the same time to speed up set unions.

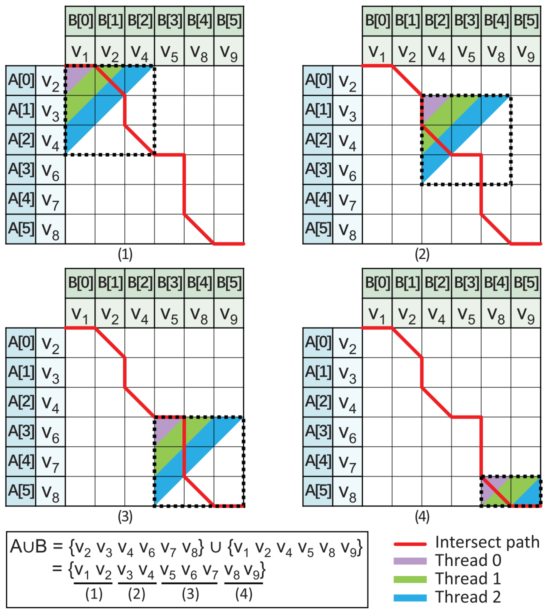
Specifically, given two ordered sets A and B, we consider the IP as a traversal of a 2D grid of size $|A| \times |B|$. IP always starts at the coordinate $(0, 0)$. Suppose IP is at (x, y), IP can move to $(x+1, y)$ (when $A[x] < B[y]$), $(x, y+1)$ (when $A[x] > B[y]$), or $(x+1, y+1)$ (when $A[x] = B[y]$) until IP reaches $(|A|, |B|)$. IP is a visual approach to parallelizing the computation. To parallelize the set union operation, we assign each thread i to find an IP segment independently. Subsequently, we merge all the results generated by these IP segments to obtain the final result of $A \cup B$. Each thread i checks all pairs of $(A[x], B[y])$ such that $x +$

Algorithm 4: Load-aware scheduling in GMBE+.

```
$processing\_v$ is a global variable initialized as 0 ;
$SM\_task\_queue$ is a concurrent queue for load balance;
// For each warp
procedure $warp\_kernel$:
    while true do
        if $SM\_task\_queue$ is not empty then
            $(L, R, C) \leftarrow SM\_task\_queue$.dequeue() ;
        else
            $v_s$ = atomicInc($processing\_v$) ;
            if $v_s \notin V$ then        // No task exists
                return;
            $L \leftarrow N(v_s); R \leftarrow \{v_s\}; C \leftarrow \emptyset$;
            foreach $v_c \in N_2(v_s)$ do
                if $L \cap N(v_c) = L$ then
                    $R \leftarrow R \cup \{v_c\}$;
                else if $v_c$ is with later order than $v_s$ then
                    $C \leftarrow C \cup \{v_c\}$;
        if $R = \Gamma(L)$ then
            if $\min\{|L|, |C|\} \times |C| > bound\_size$ and
              $\min\{|L|, |C|\} > bound\_height$ then
                foreach $v_t \in C$ do
                    $L_t \leftarrow N(v_t); R_t \leftarrow R; C_t \leftarrow \emptyset$;
                    foreach $v_c \in C$ do
                        if $L_t \cap N(v_c) = L_t$ then
                            $R_t \leftarrow R_t \cup \{v_c\}$;
                        else if $L_t \cap N(v_c) \neq \emptyset$ then
                            $C_t \leftarrow C_t \cup \{v_c\}$;
                    $SM\_task\_queue$.enqueue($(L_t, R_t, C_t)$);
                    $C \leftarrow C \setminus \{v_t\}$;
            else
                iter_search($L, R, C$);
```

$y = i$ using binary search. After that, thread i can output $B[y]$ (when the IP segment is horizontal) or $A[x]$ (when the IP segment is vertical or the upper half of a diagonal, i.e., $A[x] = B[y]$). Thread i does not output when the IP segment is the bottom half of a diagonal, i.e., $A[x-1] = B[y]$. Further elaboration is provided in Example 4.3. In view of GPU architectures, we use shared memory and warp-level primitives to implement warp-wide IP-based set unions on GPUs. We traverse IP using a $w \times w$ sliding window in parallel. In each window, we always fetch w consecutive elements from either array A or B into shared memory to coalesce global memory. Then, each thread conducts a binary search to find an IP segment using shared memory. Threads in a warp synchronize partial results using `__balloc_sync` and `__popc` primitives. Finally, the last thread (i.e., thread #$(w-1)$) computes the position of the next sliding window and broadcasts such position to all other threads in a warp using the `__shfl_sync` primitive. We use an example to clarify our design.

Example 4.3: To begin with, we illustrate how to obtain the IP for sets A and B. In Fig. 8, IP begins from the top left corner at $(0,0)$. IP firstly moves to the right to $(0,1)$ because $A[0] > B[0]$. IP then moves diagonally to $(1,2)$ because $A[0] =$



B[0] B[1] B[2] B[3] B[4] B[5]
v_1 v_2 v_4 v_5 v_8 v_9
A[0] v_2
A[1] v_3
A[2] v_4
A[3] v_6
A[4] v_7
A[5] v_8
(1)
(2)
(3)
(4)
$A \cup B = \{v_2\ v_3\ v_4\ v_6\ v_7\ v_8\} \cup \{v_1\ v_2\ v_4\ v_5\ v_8\ v_9\}$
$= \{\underbrace{v_1\ v_2}_{(1)}\ \underbrace{v_3\ v_4}_{(2)}\ \underbrace{v_5\ v_6\ v_7}_{(3)}\ \underbrace{v_8\ v_9}_{(4)}\}$
Intersect path
Thread 0
Thread 1
Thread 2

Fig. 8. Warp-wide IP-based set union approach. When computing $A \cup B$ with three threads in a warp, this approach traverses IP via four 3×3 sliding windows as shown in Fig. 8(1)-(4). Each thread independently finds an IP segment in a region of different colors, producing partial results in each window.

$B[1]$. After that IP moves downwards to $(2,2)$ because $A[1] < B[2]$. Continuing this processing, we can iteratively track IP until IP reaches the bottom right corner at $(6,6)$.

Fig. 8 depicts the procedure of warp-wide IP-based set union for sets A and B using three threads in a warp. At step(1), thread i checks all pairs $(A[x], B[y])$ such that $x + y = i$ using binary search. Thread #0 checks $(A[0], B[0])$ and knows that $A[0] > B[0]$. Then thread #0 knows that the IP segment is horizontal and outputs $B[0] = v_1$. Thread #1 first checks $(A[0], B[1])$ and knows that $A[0] = B[1]$. Then thread #1 knows that the IP segment is the upper half of a diagonal and outputs $A[0] = v_2$. Based on the binary search, thread #2 first checks $(A[1], B[1])$ and knows that $A[1] > B[1]$. Then thread #2 checks $(A[0], B[2])$ and knows that $A[0] < B[2]$. After checking $(A[1], B[1])$ and $(A[0], B[2])$, thread #2 knows that the IP segment is the bottom half of a diagonal because two minimum vertices of two pairs are the same (i.e., $A[0] = B[1]$). Finally, the last thread (i.e., thread #2) keeps two maximum indices of two last-checked pairs(i.e., A[1], B[2]), selects their indices $(i.e., (1,2))$ as the position of the next sliding window, and broadcasts such position to other threads. Continuing this processing, the procedure will finally obtain the results of the set union for sets A and B.

Lemma 4.1: Given a warp with w threads, the warp-wide IP-based set union for sets A and B runs in $O\left(\frac{(|A|+|B|)\log(w)}{w}\right)$.

Proof: In each sliding window, w threads execute simultaneously. Using the binary search, each thread takes $O(\log(w))$. The number of sliding windows is equal to $\frac{|A|+|B|}{w}$ because each sliding window processes w independent assignments independently and the number of such assignments is up to $|A| + |B|$. Thus, such approach runs in $O\left(\frac{(|A|+|B|)\log(w)}{w}\right)$. □

Our efficient set union method is significant as it provides a more effective GPU-based solution for computing 2-hop neighbors [7], [9], [16], component merging [32], and other graph mining problems [33], [34].

D. Algorithm Analysis

Time Complexity. Given a bipartite graph $G = (U, V, E)$ with $\mathcal{B}$ maximal bicliques, GMBE+ runs in $O\left(\frac{|E|\Delta_2(V)\mathcal{B}}{p}\right)$, where p is the parallel ratio. Processing each enumeration node takes $O(|E|)$, as both node generation and checking only require accessing each edge in E once. The number of nodes is bounded by $O(\Delta_2(V)\mathcal{B})$, as each node with a maximal biclique has at most $\Delta_2(V)$ child nodes. The parallel ratio p is the product of the number of GPUs, the number of SMs per GPU, and the number of warps per SM.

Recent MBE algorithms express time complexity differently. OOMBEA [9] takes $O(|E|\varsigma(V)\mathcal{B})$, where $\varsigma(V)$ is the unilateral order defined by OOMBEA. MBET [10] takes $O(\mathcal{B}\Delta^3\log(\Delta_2))$. cuMBE [16] takes at least $O\left(\frac{|U||V|\Delta_2(V)\mathcal{B}}{p}\right)$ by parallelizing MBEA [4] p times. While comparing time complexities is not trivial and the upper bound cannot fully capture the real performance, GMBE+ is much more scalable with a large parallel ratio p.

Space Complexity. GMBE+ requires $O(p(\Delta(V) + \Delta_2(V)) + |G|)$ space, as it concurrently runs p independent tasks, each needing $O(\Delta(V) + \Delta_2(V))$ space, as detailed in Section IV-A. In contrast, OOMBEA uses $O(\varsigma(V)|U| + |G|)$, MBET needs $O(\Sigma(|V(\mathcal{B})|) + |G|)$, and cuMBE requires $O(p(|U| + |V|) + |G|)$ space. Since $\Delta(V) + \Delta_2(V)$ is much smaller than $\varsigma(V)|U|$, $\Sigma(|V(\mathcal{B})|)$, or $|U| + |V|$, GMBE+ benefits from node reuse, enabling superior performance with a large parallel ratio p.

Discussion. The high performance of GMBE+ is driven by its significant parallelism. OOMBEA, MBET, and cuMBE require massive storage for intermediate results, such as batch pivoting, prefix trees, and the full vertex set. These large storage demands limit the number of parallel computation units that can be used. Additionally, graph computation irregularities [18] cause thread divergence, which is exacerbated by OOMBEA's 2-hop DFS, MBET's prefix tree maintenance, and cuMBE's frequent swap operations. Our techniques, including neighborhood size-based pruning, bitmap-based set intersection, and warp-wide IP-based Set Union, effectively mitigate these issues.

V. Implementation Issues

Pre-processing. We represent the input bipartite graph G in the compressed sparse row (CSR) format. We first load graph G into CPU memory and quickly extract important characteristics from G, such as $|U|$, $|V|$, $|E|$, $\Delta(V)$, and $\Delta_2(V)$. Since U and V are symmetrical in the bipartite graph, we always select the vertex set with fewer vertices as V similar to [9]. We then pre-process G by sorting all vertices in V using the increasing order of their degrees [3], [4] and sorting neighbor lists of each vertex using increasing order of vertex IDs similar to most of the related works [19], [30]. We transfer the whole bipartite graph G to the global memory of the GPU and enumerate all maximal bicliques on GPUs without transferring any extra data from hosts.

Lock-free task queue. To reduce the synchronization overhead, we manage the task queue in a lock-free manner using the `atomicCAS` primitive [26] in CUDA. We apply a two-level task queuing mechanism to further improve load balance. Specifically, we devise a local task queue for each block so that all warps in the block can balance workloads by accessing the local task queue. In addition, we implement a global task queue to balance workloads between different blocks. Each block only allows one proxy warp to manage tasks between the local task queue and the global task queue. We implement the local task queues using the shared memory and implement the global task queue in the global memory because atomic operations on shared memory are faster than atomic operations on the global memory.

MBE on multiple GPUs. A high-performance machine may consist of multiple GPUs to accelerate application execution performance. To support this scenario, we extend GMBE+ algorithm to multi-GPU machines. The main idea is sharing the global variable *processing_v* in Algorithm 4 on all GPU devices and replacing the `atomicInc` primitive in line #6 with `atomicInc_system` [20]. Consequently, the MBE problem is divided into multiple independent sub-problems, and each GPU independently processes these sub-problems. The overall running time is determined by the GPU with the longest execution time. Fig. 18 shows that GMBE+ is efficient on multiple GPUs because each warp on multiple GPUs can automatically balance workloads using atomic primitives with little synchronization overhead.

VI. Evaluation

In this section, we conduct experiments to evaluate the performance of GMBE+ and the proposed techniques.

A. Experimental Setup

Platform. By default, we evaluate our GPU implementations on an NVIDIA A100 GPU [27] with 108 streaming multiprocessors (SMs) and 40 GB of global memory. For comparison, we run the other CPU-based MBE algorithms on a Linux server with 96 Xeon(R) Gold 5318Y CPU @ 2.10GHz CPU cores. The operating system is Linux kernel-5.4.0.

Datasets. We use 13 real-world datasets, including a large one with 19 billion maximal bicliques, to justify the performance of GMBE+ as shown in Table I. It's worth noting that for datasets that allow multiple edges between two vertices, such as MovieLens (Mti), we only retain one unique edge between each vertex pair for MBE analysis. The number of these unique edges is denoted by $|E|$. Since U and V are symmetrical in the bipartite graph, we always denote the vertex set with fewer vertices as V, i.e., $|U| > |V|$. We obtain datasets Amazon and EuAll from the SNAP repository [35] and the other datasets from the KONECT repository [29]. Since the MBE time mainly depends on the number of maximal bicliques of the dataset, we sort all datasets in ascending order of their maximal biclique count. Datasets with more than two million maximal bicliques are referred to as large datasets in subsequent sections.

Compared algorithms. We compare GMBE+ to the CPU-oriented MBE algorithms, including the recent serial versions, i.e., OOMBEA [9] and MBET [10], the cutting-edge parallel

TABLE I
DATASET STATISTICS

Datasets	$\|U\|$	$\|V\|$	$\|E\|$	$\Delta(U)$	$\Delta_2(U)$	$\Delta(V)$	$\Delta_2(V)$	Max. bicliques
MovieLens (Mti)	16,528	7,601	71,154	640	5,817	146	3,217	140,266
Amazon (WA)	265,934	264,148	925,873	168	635	546	903	461,274
Teams (TM)	901,130	34,461	1,366,466	17	18,516	2,671	2,838	517,943
ActorMovies (AM)	383,640	127,823	1,470,404	646	3,956	294	7,798	1,075,444
Wikipedia (WC)	1,853,493	182,947	3,795,796	54	47,190	11,593	4,629	1,677,522
YouTube (YG)	94,238	30,087	293,360	1,035	37,513	7,591	7,356	1,826,587
StackOverflow (SO)	545,195	96,680	1,301,942	4,917	146,089	6,119	31,636	3,320,824
DBLP (Pa)	5,624,219	1,953,085	12,282,059	287	7,519	1,386	2,119	4,899,032
IMDB (IM)	896,302	303,617	3,782,463	1,590	15,451	1,334	15,233	5,160,061
EuAll (EE)	225,409	74,661	420,046	930	135,045	7,631	23,844	12,306,755
BookCrossing (BX)	340,523	105,278	1,149,739	2,502	151,645	13,601	53,915	54,458,953
Github (GH)	120,867	59,519	440,237	3,675	29,649	884	15,994	55,346,398
TVTropes (DBT)	87,678	64,415	3,232,134	12,400	37,493	6,507	47,459	19,636,996,096

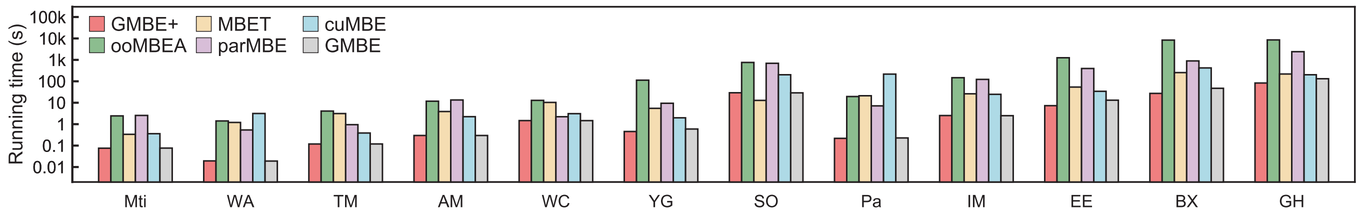


Fig. 9. Overall evaluation on general datasets (log scaled).

MBE algorithm, i.e., ParMBE [7], and the GPU-oriented MBE algorithms, i.e., GMBE [15] (the conference version) and cuMBE [16]. For fair comparisons, we obtain well-optimized codes of all competitors from the authors and run them on the same platform. We run ParMBE with 96 threads because our machine contains 96 CPU cores.

Measures. We measure the running time of each algorithm excluding the time spent reading the graph from the disk. Without specification, GMBE+ leverages all proposed techniques in Section IV. By default, GMBE+ sets threshold τ for bitmaps to 32, sets the thresholds for *bound_height* and *bound_size* to 20 and 1,500 respectively, sets `WarpPerSM` to 16, and sorts V in ascending order based on the vertex degree before the enumeration. We also implement other variants to further evaluate proposed techniques. We will detail those variants in the corresponding experiments. The reported running time is the average of five trials per configuration.

B. Overall Evaluation

Evaluation on general datasets. Fig. 9 compares the running time of GMBE+ to state-of-the-art MBE algorithms on real-world datasets. The experimental results show that GMBE+ is 1.2× faster than GMBE on average and surpasses any other next-best competitor by 1.5×–32.8× across all datasets, with an average improvement of 9.0×. The only exception is the StackOverflow dataset, where MBET's prefix tree approach is highly efficient due to its complex control flow. However, this complexity exacerbates thread divergence on GPUs, making it difficult to integrate into GMBE+. The Analysis of Variance (ANOVA) further demonstrates that GMBE+ runs consistently across all datasets, with variances below 0.1 in each of the five repetitions, and running times vary significantly across datasets. This is because GMBE+ efficiently uses GPU resources. Specifically, GMBE+ speeds up GMBE by 1.8× on the BookCrossing dataset, thanks to its utilization of bitmap-based intersections and advanced set unions. In addition, we conduct a performance analysis of GMBE+ using the NVIDIA Nsight Compute software [36]. The profiling results indicate that the average warp execution efficiency is 64%, and the memory utilization is 12% across all real-world datasets. These results can be attributed to the inherent irregularity present in the MBE problem [18]. In our experiments, a 24-core CPU consumes 165W, while an A100 GPU consumes 300W. With an average performance improvement of 9.0×, the energy efficiency improves by 9.0 × 165 / 300 = 5.0×. Although GPUs are more expensive than CPUs, GMBE+ utilizes the GPU's parallel scalability to meet high-performance demands, a capability that CPUs may not provide.

Evaluation on the large dataset. We further evaluate GMBE+ on the large TVTropes dataset, which contains 19 billion maximal bicliques. Fig. 11 shows the change in the number of bicliques over time, with the red line representing the total number in the dataset. We present results for GMBE+ and MBETM, the space-optimized version of MBET, as other competitors fail to generate 10% of bicliques within 48 hours. GMBE+ produces 98% of the maximal bicliques in 18 hours, while MBETM only produces 58% in 48 hours, thanks to GMBE+'s efficient parallelization.

C. Effect of Optimizations

To verify the effect of proposed optimizations, we compare the runtime of GMBE+ with various variants, each disabling a single optimization, as depicted in Fig. 10.

Effect of the node reuse approach. To study the impact of the node reuse approach in Section IV-A, we design a variant

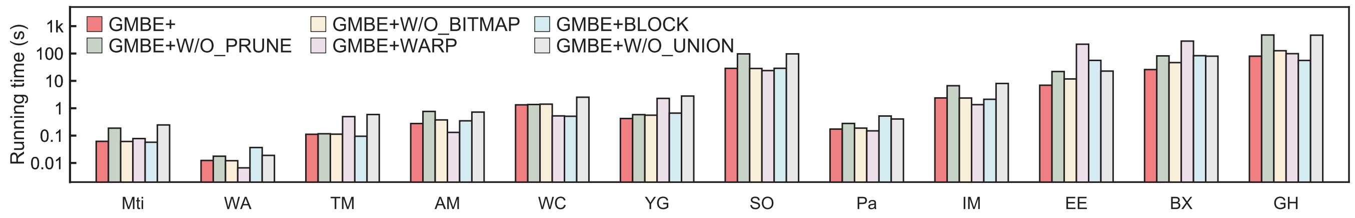


Fig. 10. Effect of optimizations, including pruning, bitmap-based set intersection, task scheduling, and set union approaches (log scaled).

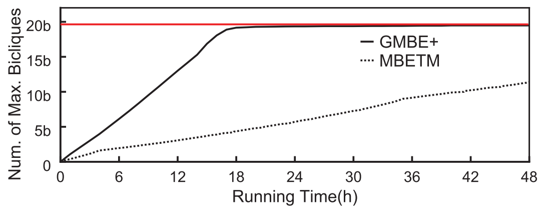


Fig. 11. Overall evaluation on the TVTropes dataset.

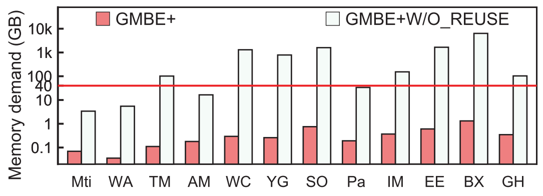


Fig. 12. Effect of the node reuse approach (log scaled). The red line indicates the GPU memory capacity of NVIDIA A100.

GMBE+w/o_REUSE that pre-allocates memory on GPUs according to Section III-A. We estimate the memory requirement allocated by the cudaMalloc primitive for GMBE+ with and without the node reuse approach. This memory requirement includes the pre-allocated memory for the input bipartite graph and the runtime subtrees. Fig. 12 shows that the node reuse approach significantly reduces the memory requirement by 49×–4,819× on all testing datasets while GMBE+w/o_REUSE is impractical because its memory requirement exceeds the memory capacity of the A100 GPU on multiple datasets.

Effect of the pruning approach. To evaluate the local-neighborhood-size-based pruning approach in Section IV-A2, we design a variant GMBE+w/o_PRUNE that disables the pruning function of GMBE+. Fig. 10 shows that GMBE+ constantly outperforms GMBE+w/o_PRUNE, because the pruning approach prunes enumeration space for MBE at runtime. To further explore the pruning efficiency, we use α to represent the number of maximal bicliques and δ to represent the number of non-maximal bicliques removed by node checking (line #14 in Algorithm 2). Since α remains constant for each dataset, we use the ratio δ/α to indicate the pruning efficiency for both approaches in Table II. This analysis reveals that the proposed pruning approach can avoid 48.7%-92.8% non-maximal biclique checks among all testing datasets. This pruning technique plays a crucial role, particularly for larger datasets where the enumeration space grows with an increasing number of maximal bicliques. Consequently, it significantly reduces the running time on Github from 475 seconds to 80 seconds.

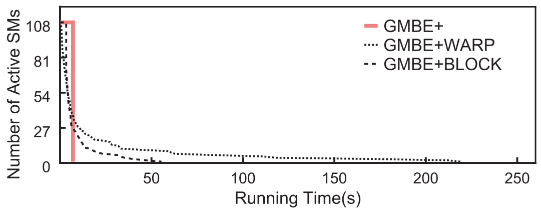
TABLE II
COMPARISON OF THE RATIO OF GENERATED NON-MAXIMAL BICLIQUES TO MAXIMAL BICLIQUES BETWEEN GMBE+ AND GMBE+w/o_PRUNE

Datasets	Mti	WA	TM	AM	WC	YG
GMBE+	9.04	0.734	1.63	12.9	0.71	2.11
GMBE+w/o_PRUNE	66.0	3.68	3.88	53.0	2.89	20.1
Datasets	SO	Pa	IM	EE	BX	GH
GMBE+	89.4	0.362	15.5	4.04	3.40	11.1
GMBE+w/o_PRUNE	174	1.43	74.4	56.0	27.3	51.4

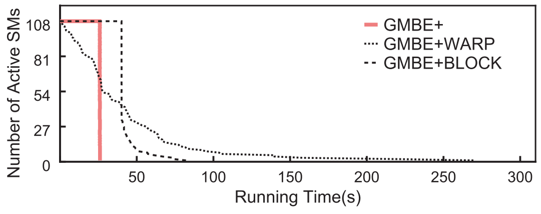
Effect of the bitmap-based set intersection. To study the impact of the bitmap-based set intersection approach in Section IV-B, we design a variant GMBE+w/o_BITMAP, which disables this bitmap-based optimization in GMBE+. Fig. 10 shows that GMBE+ outperforms GMBE+w/o_BITMAP in most cases, achieving a performance improvement of 1.80× and 1.58× on the BookCrossing and Github datasets, respectively. This improvement is attributed to the efficient set intersections facilitated by bitmaps during enumeration. However, GMBE+ exhibits slightly slower performance compared to GMBE+w/o_BITMAP on the MovieLens, Amazon, StackOverflow, and IMDB datasets, primarily due to the overhead associated with creating multiple bitmaps, which offsets the benefits of bitmap-based set intersection.

Effect of the task scheduling approach. To study the effect of the load-aware task-centric scheme in Section IV-C, we design two variants GMBE+WARP and GMBE+BLOCK that apply warp-centric and block-centric schemes respectively. Fig. 10 shows that GMBE+ surpasses both variants, with a 4.88× improvement over GMBE+WARP and 2.07× over GMBE+BLOCK. Notably, on the EuAll dataset, GMBE+ is 31.5× and 8× faster than GMBE+WARP and GMBE+BLOCK, respectively. This superior performance of GMBE+ is due to its dynamic task detection and workload rebalancing using lock-free task queues. However, in some cases with balanced workloads, like the Wikipedia dataset, GMBE+ lags behind GMBE+WARP or GMBE+BLOCK since the task-centric scheme still needs to rebalance workloads, incurring overhead even when not needed.

To further explore the load imbalance problem for MBE on GPUs, we record the number of active SMs while running GMBE+, GMBE+WARP, and GMBE+BLOCK. Fig. 13 reports the comparison of runtime loads on SMs on the BookCrossing and EuAll datasets. Because of the imbalanced workloads, SMs with light workloads may finish early and wait for the SM with the heaviest workload, which is costly. GMBE+WARP gains the worst performance because the number of active SMs decreases rapidly due to the load imbalance. GMBE+BLOCK



(a) EuAll



(b) BookCrossing

Fig. 13. Comparison of runtime loads on SMs among GMBE+, GMBE+WARP, and GMBE+BLOCK.

obtains better performance than GMBE+WARP because GMBE+BLOCK could spend more resources on each workload than GMBE+WARP (i.e., a block vs. a warp) which reduces the time waiting for the SM with the heaviest workload. However, GMBE+BLOCK is insufficient because the workloads for the MBE problem could be severely imbalanced. For instance, over 80% of SMs (86 SMs / 108 SMs) waste over 85% of running time (47 seconds / 55 seconds) waiting for the slowest SM on EuALL. GMBE+ always achieves the best performance because GMBE+ works in the finest granularity and each SM finishes its work roughly at the same time. GMBE+ completes even before the number of active SMs of GMBE+BLOCK starts to decrease on BookCrossing because GMBE+ activates all warps in each SM while GMBE+BLOCK may use only use a small portion of warps in each SM at runtime.

Effect of the set union approach. To study the effect of the intersect-path-based set union approach in Section IV-C, we design a variant GMBE+W/O_UNION that executes the set union operation using only one thread in a warp. Fig. 10 shows that GMBE+ constantly beats GMBE+W/O_UNION since our proposed approach fully utilizes parallel computational resources on GPUs. As a result, GMBE+ speeds up GMBE+W/O_UNION by 6.6× on YouTube.

D. Sensitivity Analysis

Impact of threshold τ for bitmap-based set intersections. To determine the threshold τ in the bitmap-based set intersection approach in Section IV-B. We assess the performance of GMBE+ with τ values of 8, 16, 32, 64, and 128, respectively. Fig. 14 shows that GMBE+ achieves peak performance when τ is set to 32. Thus, GMBE+ sets τ to 32.

Impact of thresholds for task scheduling. To explore the efficient configuration for thresholds *bound_height* and *bound_size* in Section IV-C, we design multiple variants GMBE+ (m, n), where m and n represent *bound_height* and *bound_size* respectively. We always set m larger than n^2 because $|L| \times |C|$ is always greater or equal to $(min\{|L|,|C|\})^2$. The selection of

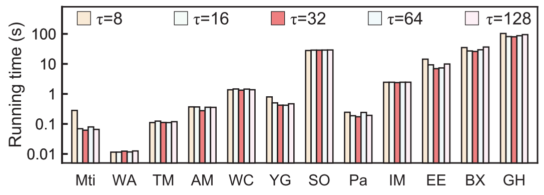


Fig. 14. Impact of threshold τ for bitmap-based set intersections (log scaled).

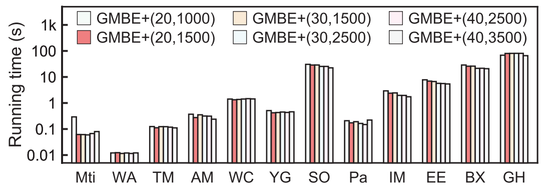


Fig. 15. Impact of thresholds *bound_height* and *bound_size* for load-aware task scheduling (log scaled).

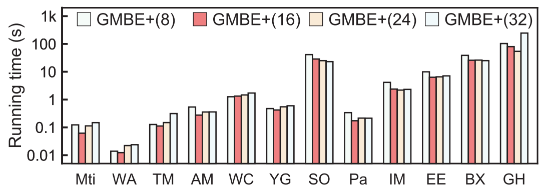


Fig. 16. Impact of the parameter `WarpPerSM` (log scaled).

thresholds is a trade-off between the parallel granularity and synchronization overhead. We require smaller thresholds to balance workloads in finer granularity. However, the thresholds should not be too small. Otherwise, we have to manage more tasks with the huge synchronization overhead. Fig. 15 shows that the variant GMBE+(20, 1500) is empirically better than the others in most cases. Thus, GMBE+ applies this configuration by default.

Impact of the number of warps in each SM. To determine the parameter `WarpPerSM` in the PT model in Section IV-C, we design variants that set `WarpPerSM` to 8, 16, 24, and 32, respectively. The selection of `WarpPerSM` is a trade-off between parallelism and the resources for each warp. Intuitively, we expect `WarpPerSM` to be larger so that we can run more MBE tasks in parallel. However, `WarpPerSM` should not be too large since the computational resources (e.g., registers) in each SM are limited. A larger `WarpPerSM` may decrease the performance of GMBE+ because each warp will have fewer resources to run MBE tasks. Fig. 16 shows that the variant GMBE+(16) outperforms the other variants by up to 2.8× on most large datasets, such as BookCrossing, IMDB, DBLP, and EuAll. In addition, GMBE+(16) is 0.66× slower than GMBE+(24) on Github due to its extensive enumeration space that requires more warps to enumerate maximal bicliques in parallel. Considering its efficiency in most cases, GMBE+ sets `WarpPerSM` to 16 by default.

Adaptability on different GPUs. To explore the adaptability of GMBE+, we evaluate GMBE+ on an NVIDIA A100 GPU (108 SMs, 40 GB memory), a V100 GPU (80 SMs, 32 GB memory) [37], a 2080Ti GPU (68 SMs, 11 GB memory), and a 4090 GPU (128 SMs, 24 GB memory) [38], respectively.

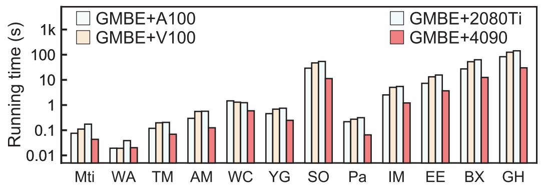
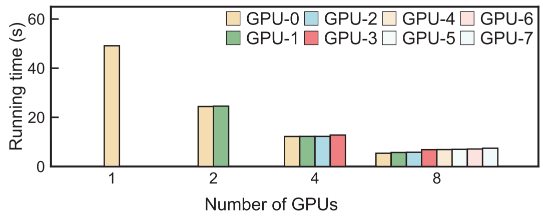
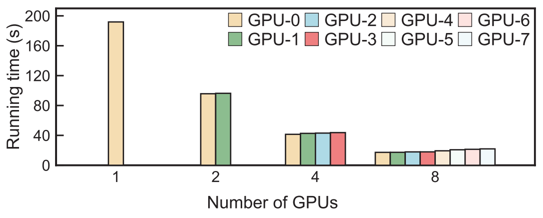


Fig. 17. Adaptability on different GPU (log scaled).



(a) BookCrossing



(b) Github

Fig. 18. Scalability of GMBE+ on a machine with multi-GPU.

Fig. 17 shows that GMBE+ is adaptive on all four GPUs. GMBE + 4090 is faster than the others because a 4090 GPU contains more SMs than other GPUs.

Scalability on multi-GPU. To explore the scalability of GMBE+ on multi-GPU, we conduct experiments on a machine with 8 NVIDIA A30 GPUs [39]. To optimize GMBE+ for multi-GPU configurations, we divide the problem into multiple independent sub-problems, and the total execution time is determined by the longest-running sub-problem. Fig. 18 shows that GMBE+ scales out linearly on Github and BookCrossing datasets as we increase the number of GPUs because each GPU finishes its execution almost at the same time. With the help of multiple GPUs, GMBE+ can enumerate over 55 million maximal bicliques on the Github dataset within 22 seconds. It achieves speedups of 110× and 1.41× respectively compared to the state-of-the-art CPU-based parallel MBE algorithm ParMBE on a 96-core CPU machine (i.e., 2,411 seconds) and the state-of-the-art GPU-based MBE algorithm GMBE (i.e., 31 seconds).

VII. Related Work

Maximal Biclique Enumeration on CPUs. The most efficient class of MBE algorithms [3], [4], [6], [7], [8], [9], [10], [40] is based on backtracking with recursions on CPUs. Most of the research efforts [3], [4], [8], [9], [10] applied various sorting, pruning techniques, and the prefix tree on a serial MBE algorithm to reduce the search space for the MBE problem. Some other works [6], [7] improved the efficiency of MBE by parallelizing MBE algorithms on multicore CPUs or distributed architectures. However, the existing MBE algorithms do not work on GPUs due to the challenges in Section III, and thus achieve limited performance with limited computational resources on CPUs.

Related Problems on GPUs. Although GPUs are widely used to accelerate related graph algorithms, such as maximal clique enumeration (MCE) [21], [41], [42] and graph pattern mining (GPM) [19], [30], [43], it is still challenging for MBE on GPUs. Specifically, MCE on GPUs suffers from similar performance issues as MBE described in Section III. As a result, the latest GPU-based MCE algorithm GBK [21] only obtained a comparable performance to a single-thread sequence algorithm on the CPU. Because the enumerated subgraphs (i.e., maximal bicliques) for MBE generally contain more vertices than those for other GPM problems, optimizations in the latest GPU-based GPM framework G^{2}Miner [19] cannot address the memory issue and severe load imbalance for MBE on GPUs. Recent GPU-based MBE algorithms [15, 16] are suboptimal as they suffer from thread divergence when dealing with vertex neighbors of different sizes and fail to balance extensive workloads.

VIII. Conclusion

In this paper, we present GMBE+, an advanced GPU solution for the MBE problem. MBE on GPUs faces serious challenges, including large memory requirement, thread divergence, and severe load imbalance. To address these problems, we design a node-reuse approach to reduce GPU memory usage with advanced node pruning, a bitmap-based set intersection approach to minimize thread divergence, and a load-aware task scheduling framework to achieve load balance among threads within GPU warps facilitated by a novel set union approach. We conduct comprehensive evaluations using 13 real-world datasets and three different GPUs. Our experimental results show that GMBE+ outperforms the state-of-the-art GPU-based parallel MBE algorithm GMBE by 1.2× on average.

Data Availability Statement

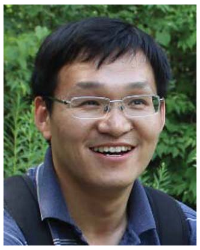
The code of GMBE+ is available at https://github.com/fhxu00/MBE-GPU.

References

[1] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou, “Maximum and top-k diversified biclique search at scale,” *VLDB J.*, vol. 31, no. 6, pp. 1365–1389, 2022.
[2] K. Wang, W. Zhang, X. Lin, L. Qin, and A. Zhou, “Efficient personalized maximum biclique search,” in *Proc. IEEE 38th Int. Conf. Data Eng. (ICDE),* 2022, pp. 498–511.
[3] G. Liu, K. Sim, and J. Li, “Efficient mining of large maximal bicliques,” in *Proc. Int. Conf. Data Warehousing Knowl. Discovery (DaWaK),* 2006, pp. 437–448.
[4] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston, “On finding bicliques in bipartite graphs: A novel algorithm and its application to the integration of diverse biological data types,” *BMC Bioinf.*, vol. 15, pp. 1–18, 2014.
[5] J. Yang, Y. Peng, and W. Zhang, “(p,q)-Biclique counting and enumeration for large sparse bipartite graphs,” *Proc. VLDB Endowment (PVLDB)*, vol. 15, no. 2, pp. 141–153, 2021.

[6] A. P. Mukherjee and S. Tirthapura, “Enumerating maximal bicliques from a large graph using mapreduce,” *IEEE Trans. Services Comput. (TSC)*, vol. 10, no. 5, pp. 771–784, 2016.
[7] A. Das and S. Tirthapura, “Shared-memory parallel maximal biclique enumeration,” in *Proc. IEEE 26th Int. Conf. High Perform. Comput., Data, Anal. (HiPC),* 2019, pp. 34–43.
[8] A. Abidi, R. Zhou, L. Chen, and C. Liu, “Pivot-based maximal biclique enumeration,” in *Proc. 29th Int. Conf. Int. Joint Conf. Artif. Intell. (IJCAI),* 2020, pp. 3558–3564.
[9] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li, “Efficient maximal biclique enumeration for large sparse bipartite graphs,” *Proc. VLDB Endowment (PVLDB)*, vol. 15, no. 8, pp. 1559–1571, 2022.
[10] J. Chen, K. Wang, R. H. Li, H. Qin, X. Lin, and G. Wang, “Maximal Biclique enumeration: A prefix tree based approach,” in *Proc. IEEE 40th Int. Conf. Data Eng. (ICDE),* pp. 2544–2556, 2024.
[11] Z. Pan et al., “Ambea: Aggressive maximal biclique enumeration in large bipartite graph computing,” *IEEE Trans. Comput.*, vol. 73, no. 12, pp. 2664–2677, 2024.
[12] Z. Pan et al., “Enumeration of billions of maximal bicliques in bipartite graphs without using gpus,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2024.
[13] R. Rymon, “Search through systematic set enumeration,” in *Proc. 3rd Int. Conf. Princ. Knowl. Representation Reasoning (KR)*, 1992, pp. 539–550.
[14] D. Eppstein, “Arboricity and bipartite subgraph listing algorithms,” *Inf. Process. Lett.*, vol. 51, no. 4, pp. 207–211, 1994.
[15] Z. Pan, S. He, X. Li, X. Zhang, R. Wang, and G. Chen, “Efficient maximal biclique enumeration on GPUs,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC),* 2023, pp. 1–13.
[16] C.-Y. Hsieh, C.-M. Chang, P.-H. Cheng, and S.-Y. Kuo, “Accelerating maximal biclique enumeration on GPUs,” 2024, *arXiv:2401.05039*.
[17] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, “Are dynamic memory managers on GPUs slow? A survey and benchmarks,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, 2021, pp. 219–233.
[18] M. Burtscher, R. Nasre, and K. Pingali, “A quantitative study of irregular programs on GPUs,” in *Proc. IEEE Int. Symp. Workload Characterization (IISWC),* 2012, pp. 141–151.
[19] X. Chen et al., “Efficient and scalable graph pattern mining on GPUs,” in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implement. (OSDI),* 2022, pp. 857–877.
[20] “CUDA C++ programming guide,” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024.
[21] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai, “Accelerating the bron-kerbosch algorithm for maximal clique enumeration using GPUs,” *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, vol. 32, no. 9, pp. 2352–2366, 2021.
[22] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Scientific Comput.*, vol. 20, no. 1, pp. 359–392, Jan. 1998.
[23] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, “Sandslash: A two-level framework for efficient graph pattern mining,” in *Proc. ACM Int. Conf. SuperComput. (ICS),* 2021, pp. 378–391.
[24] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, “Adaptive cache management for energy-efficient GPU computing,” in *Proc. IEEE/ACM Int. Symp. Microarchit. (MICRO)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 343–355.
[25] O. Green, P. Yalamanchili, and L.-M. Munguía, “Fast triangle counting on the GPU,” in *Proc. 4th Workshop Irregular Appl.: Archit. Algorithms*, 2014, pp. 1–8.
[26] Y. Lin and V. Grover, “Using CUDA warp-level primitives,” https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, 2018.
[27] “NVIDIA A100 Tensor Core GPU,” https://www.nvidia.com/en-gb/data-center/a100/, 2024.
[28] “Single instruction, multiple threads (SIMT) from wikipedia,” https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads, 2024.
[29] J. Kunegis, “Konect: The koblenz network collection,” in *Proc. 22nd Int. Conf. World Wide Web (WWW),* 2013, pp. 1343–1350.
[30] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W-m Hwu, “Parallel k-clique counting on GPUs,” in *Proc. 36th ACM Int. Conf. SuperComput. (ICS),* 2022, pp. 1–14.
[31] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style GPU programming for GPGPU workloads,” in *Proc. Innovative Parallel Comput. (InPar)*, 2012, pp. 1–14.
[32] D. Tench et al., “Graphzeppelin: How to find connected components (even when graphs are dense, dynamic, and massive),” *ACM Trans. Database Syst.*, vol. 49, no. 3, May 2024.
[33] Q. Chen, B. Tian, and M. Gao, “Fingers: Exploiting fine-grained parallelism in graph mining accelerators,” in *Proc. 27th ACM Int. Conf. Architect. Supp. Program. Lang. Oper. Syst., Ser. (ASPLOS)*. ACM, 2022, pp. 43–55.
[34] Z. Lin, K. Meng, C. Shui, K. Zhang, J. Xiao, and G. Tan, “Exploiting fine-grained redundancy in set-centric graph pattern mining,” in *Proc. 29th ACM SIGPLAN Annu. Symp. Princ. Pract. Parallel Program. (PPoPP)*. ACM, 2024, pp. 175–187.
[35] J. Leskovec and A. Krevl, “Snap datasets: Stanford large network dataset collection,” Jun. 2014. [Online]. Available: http://snap.stanford.edu/data
[36] “Nvidia nsight compute,” 2024. [Online]. Available: https://developer.nvidia.com/nsight-compute/
[37] “NVIDIA V100 Tensor Core GPU,” 2024. [Online]. Available: https://www.nvidia.com/en-gb/data-center/v100/
[38] “NVIDIA GeForce RTX 4090 GPU,” 2025. [Online]. Available: https://www.nvidia.com/en-gb/geforce/graphics-cards/40-series/rtx-4090/
[39] “NVIDIA A30 Tensor Core GPU,” 2024. [Online]. Available: https://www.nvidia.com/en-gb/data-center/products/a30-gpu/
[40] Z. Ma, Y. Liu, Y. Hu, J. Yang, C. Liu, and H. Dai, “Efficient maintenance for maximal bicliques in bipartite graph streams,” *World Wide Web (WWW*, vol. 25, no. 2, pp. 857–877, 2022.
[41] T. Alusaifeer, S. Ramanna, C. J. Henry, and J. Peters, “GPU implementation of MCE approach to finding near neighbourhoods,” in *Proc. Int. Conf. Rough Sets Knowl. Technol.*, 2013, pp. 251–262.
[42] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, “Maximal clique enumeration with data-parallel primitives,” in *Proc. IEEE 7th Symp. Large Data Anal. Vis. (LDAV),* 2017, pp. 16–25.
[43] W. Guo, Y. Li, and K.-L. Tan, “Exploiting reuse for GPU subgraph enumeration,” *IEEE Trans. Knowl. Data Eng. (TKDE)*, vol. 34, no. 9, pp. 4231–4244, 2020.

Zhe Pan (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and technology from Zhejiang University, Hangzhou, China. Currently, he is a Postdoctoral Researcher with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include computer architecture, graph computing, and hardware and software co-design.



Shuibing He (Member, IEEE) received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology, in 2009. Currently, he is a ZJU100 Young Professor with the College of Computer Science and Technology, Zhejiang University, China. His research interests include intelligent computing, high-performance computing, graph computing, memory, and storage systems. He is a member of ACM.

Xu Li received the B.S. degree in intelligent science and technology from Xiamen University, Xiamen, China, in 2021. He received the M.S. degree in electronic information from Zhejiang University, Hangzhou, China, in 2024. His research interests include graph computing, parallel algorithms, and hardware-aware software optimization.

Xuechen Zhang (Member, IEEE) received the M.S. and Ph.D. degrees in computer engineering from Wayne State University. Currently, he is an Associate Professor with the School of Engineering and Computer Science, Washington State University Vancouver. His research interests include the areas of file and storage systems, operating systems, and high-performance computing. He is a member of ACM.

Rui Wang received the Ph.D. degree in computer science from the University of Science and Technology of China (USTC), in 2021. Currently, she is a ZJU100 Research Fellow with the School of Software Engineering, Zhejiang University (ZJU). Before that, she worked as a Postdoctoral Fellow with the School of Computer Science and Technology, ZJU, from 2021 to 2023. She is interested in graph computing and graph storage.

Yanlong Yin received the Ph.D. degree in computer science from Illinois Institute of Technology, USA. Currently, he is a Researcher working with the College of Computer Science and Technology, Zhejiang University, China. His research interests include parallel storage, parallel computing, and I/O optimization.

Gang Chen (Member, IEEE) received the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China. Currently, he is a Professor with the College of Computer Science, Zhejiang University. He has led successful research projects focused on building China’s own database management systems. His research interests include relational database systems, large-scale data management technologies, and intelligent computing.