Efficient Maximal Biclique Enumeration on GPUs

Zhe Pan
Zhejiang University
Hangzhou, China
panzhe@zju.edu.cn

Shuibing He*
Zhejiang University
Hangzhou, China
heshuibing@zju.edu.cn

Xu Li
Zhejiang University
Hangzhou, China
fhxu@zju.edu.cn

Xuechen Zhang
Washington State University
Vancouver
Vancouver WA, USA
xuechen.zhang@wsu.edu

Rui Wang
Zhejiang University
Hangzhou, China
rwang21@zju.edu.cn

Gang Chen
Zhejiang University
Hangzhou, China
cg@zju.edu.cn

ABSTRACT

Maximal biclique enumeration (MBE) in bipartite graphs is an important problem in data mining with many real-world applications. All existing solutions for MBE are designed for CPUs. 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, the first highly-efficient GPU solution for the MBE problem. To overcome the challenges, we design a node-reuse approach to reduce GPU memory usage, a pro-active pruning method using the vertex's local neighborhood size to alleviate thread divergence, and a load-aware task scheduling framework to achieve load balance among threads within GPU warps and blocks. Our experimental results show that GMBE on an NVIDIA A100 GPU can achieve 70.6× speedup over the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine.

CCS CONCEPTS

• **Mathematics of computing** → **Graph enumeration**; • **Computer systems organization** → **Multicore architectures**; • **Information systems** → *Data mining*.

KEYWORDS

Maximal biclique enumeration, GPU

ACM Reference Format:
Zhe Pan, Shuibing He, Xu Li, Xuechen Zhang, Rui Wang, and Gang Chen. 2023. Efficient Maximal Biclique Enumeration on GPUs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607059

*Shuibing He is the corresponding author.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607059

1 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.

MBE is important in bipartite graph analysis, with widespread applications, such as anomaly detection in e-commerce networks [30, 35], social recommendation in social networks [29], gene expression analysis in expression datasets [34, 39], and GNN information aggregation [38]. 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 [10, 30, 35]. 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 [8, 9, 14, 18, 22, 29, 32, 39]. A mainstream approach is to use the set enumeration tree [33] 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 [19], especially for large graphs. Therefore, many efforts are made to reduce the enumeration space using pruning [8, 14] and vertex ordering [14, 39]. 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 to speed up the enumeration process [18]. However, their performance speedup is still constrained by the limited parallelism of CPUs. For instance, the state-of-the-art parallel MBE algorithm ParMBE [18] costs over 40 minutes to finish running MBE on a medium-scale bipartite graph Github [25] on a 96-core CPU machine. In contrast, our GPU-based solution can reduce the running time to 132 seconds on an NVIDIA A100 GPU [4] because GPUs offer much higher computational throughput and parallelism than CPUs.

There are three major challenges in the design of a highly efficient MBE algorithm for GPUs. 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 [37], due to the severe shortage of memory resources.

Second, the performance of existing MBE algorithms suffers from irregular computation [13] 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 [15]. GPUs will serialize these diverged thread operations [1], 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 [36] and graph pattern mining [15]. The optimizations include data graph partitioning [23], two-level parallelism [17], adaptive buffering [16], hybrid order on GPU [15], etc. However, none of them is efficient for MBE using GPUs. This is 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 highly efficient maximal biclique enumeration on GPUs, we design the first GPU-based MBE algorithm (GMBE) considering the characteristics of GPU architecture and MBE computation pattern. Specifically, first, we replace the existing recursion with a stack-based iteration and reuse the memory of the root node throughout the iteration process. This approach effectively reduces memory usage because it eliminates the need to allocate additional memory space for new nodes. Second, we use the local neighborhood size of a vertex to reduce the enumeration space. The new pruning approach can reduce the number of sub-trees without visiting the nodes. Thus, it can significantly reduce thread divergence. 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.

We adopt the fast CUDA primitives [28] 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 on an NVIDIA A100 GPU can achieve up to 70.6× speedup over the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine.

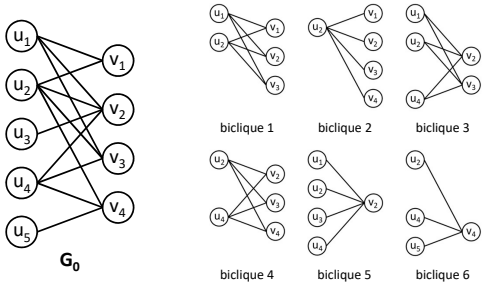


Figure 1: A bipartite graph G_0 containing 6 maximal bicliques.

2 BACKGROUND

In this section, we briefly review the state-of-the-art MBE algorithms on CPUs and introduce the compute unified device architecture (CUDA) in modern GPUs.

2.1 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., $N_2(u) = \bigcup_{v \in N(u)} N(v) - \{u\}$. For a vertex set X, $\Gamma(X)$ denotes the common neighbors of vertices in X, i.e., $\Gamma(X) = \bigcap_{u \in X} N(u)$. $\Delta(X)$ 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 $(L \cup R) \subset (L' \cup R')$. For instance, Figure 1 shows a bipartite graph G_0 and all maximal bicliques in G_0.

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

Baseline solution. To efficiently solve the MBE problem, recent works [8, 14, 18, 22, 24, 29, 32, 39] **recursively** run a backtracking procedure to generate the powerset of V using a set enumeration tree [33] and then obtain all maximal bicliques correspondingly. In most works [8, 18, 29, 32], 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 illustrate the basic recursive procedure for each enumeration node in existing works using Algorithm 1. Then, we provide a detailed example to demonstrate the application of Algorithm 1 to the bipartite graph G_0 in Figure 1.

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

Algorithm 1: Recursive MBE Algorithm

```
   Data: Bipartite graph G(U, V, E)
   Input: Set L ⊆ U, disjoint sets R, C ⊆ V
   Output: All maximal bicliques
1  procedure recursively_search(L, R, C):
2      foreach v' ∈ C do
3          L' ← L ∩ N(v'); R' ← R; C' ← ∅;
4          foreach v_c ∈ C do                    // Node generation
5              if L' ∩ N(v_c) == L' then
6                  R' ← R' ∪ {v_c};
7              else if L' ∩ N(v_c) ≠ ∅ then
8                  C' ← C' ∪ {v_c};
9          if R' == Γ(L') then                   // Maximality check
10             Output(L', R') as a maximal biclique;
11             recursively_search(L', R', C') ;  // Recursion
12         C ← C \ {v'};
```

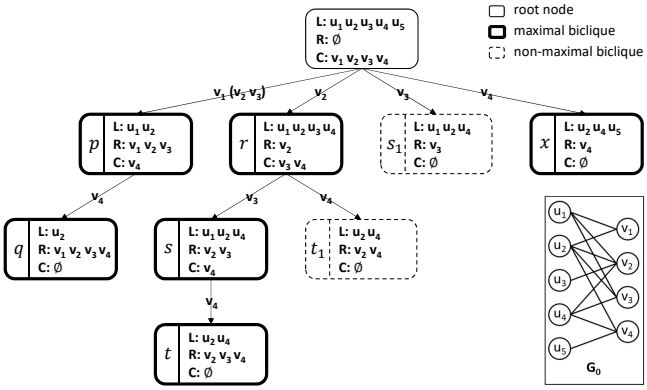


Figure 2: An enumeration tree for bipartite graph G_0.

#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 [14, 39] use a set Q to keep traversed candidate vertices for accelerating the node checking in line #9. Similar to many existing works [8, 18, 29, 32], we omit the set Q to reduce the memory consumption.

Example 2.1. *Figure 2 depicts an enumeration tree for a bipartite graph G_0 using Algorithm 1 without any optimization. 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.

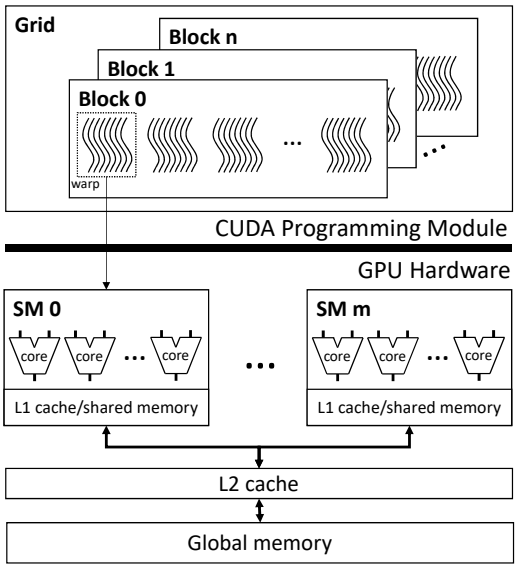


Figure 3: GPU architecture.

Recent optimizations. To reduce the computational overhead, researchers mainly focus on reducing the enumeration space, using various vertex ordering and pruning approaches [8, 14, 39]. To achieve further speedup, other works parallelized MBE algorithms on multicore CPUs [18] or distributed architectures [32]. Specifically, existing parallel MBE algorithms 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. For instance, the state-of-the-art parallel MBE algorithm ParMBE [18] costs over 40 minutes to enumerate all maximal bicliques on a medium-scale bipartite graph Github (180k vertices and 440k edges) on a 96-core CPU machine, as shown in Section 6.2.

2.2 GPU Architecture and Programming

GPU architecture. Figure 3 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 [4] 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 [1, 2] 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 [7] 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. In this way, 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 [37]. Therefore, we should carefully manage the valuable GPU memory 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. Because GPU has to serialize the different execution paths, thread divergence can severely degrade the execution performance [1]. 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.

3 CHALLENGES OF MBE ON GPUS

A naive approach to perform 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 Figure 2, we can divide the whole tree 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.

3.1 Large Memory Requirement

Existing MBE algorithms usually dynamically allocate memory for new enumeration nodes (lines #3-8 in Algorithm 1), which is very expensive on GPUs as mentioned in Section 2.2. 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. Therefore, the total amount of memory space that needs to be pre-allocated for each subtree traversal procedure is $\Delta(V) \times (\Delta(V) + \Delta_2(V)) \times$ **sizeof**($vertex$). For example, when using the NVIDIA A100 GPU with 40 GB memory to perform MBE on a real-world bipartite graph BookCrossing [25], the memory requirement for each subtree traversal procedure is $13,601 \times (13,601 + 53,915) \times$ **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.

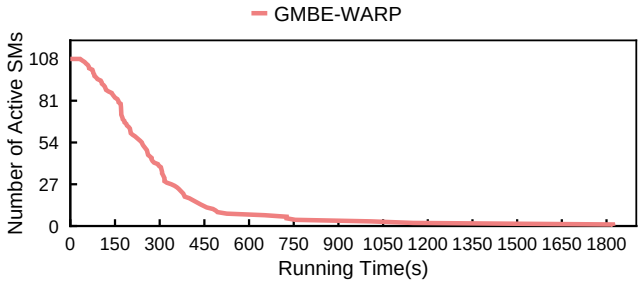


Figure 4: Load imbalance for the MBE problem. 86 SMs waste over 1,458 seconds waiting for the slowest one on BookCrossing if we assign each enumeration tree to a warp based on our new algorithm GMBE as shown in Section 6.3.

3.2 Massive Thread Divergence

As mentioned in Section 2.2, thread divergence can significantly degrade thread performance on GPUs. In the case of MBE on GPUs, thread divergence becomes more prominent due to two reasons. First, during the enumeration process of Algorithm 1, threads within the same warp may traverse different vertices to generate new nodes (line #2) and use different neighbors to compute different sets of L, R, and C (lines #3-9). This leads to each thread accessing different memory areas and executing different control flows. Second, several pruning techniques [8, 14, 39] have been proposed to reduce the enumeration space and accelerate the MBE process (refer to Section 2.1). For example, ooMBEA [14] traverses the 2-hop neighbors of all candidate vertices to identify its defined batch pivots and discard enumerated branches that do not belong to those batch pivots. However, this further exacerbates thread divergence by generating more divergent threads through the traversal of 2-hop neighbors. Hence, it is challenging to mitigate thread divergence during enumeration and crucial to develop a GPU-friendly pruning approach.

3.3 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^2Miner [15] always assigns each enumeration tree to a warp in the GPU. However, Figure 4 shows that if we assign each enumeration tree to a warp based on our new algorithm GMBE, over 80% of SMs (i.e., 86 SMs / 108 SMs) waste 80% of running time (i.e., 1,458s / 1,822s) waiting for the slowest one on the BookCrossing dataset. It is necessary to balance workloads for MBE in a finer granularity.

4 DESIGN OF GMBE

We propose a new GPU-based MBE algorithm (GMBE) with three novel techniques. (1) We design a stack-based iterative MBE algorithm to replace the original recursive algorithm. The new MBE algorithm reuses the nodes of enumeration trees to reduce memory usage. (2) It uses local neighborhood sizes for pro-actively pruning for less thread divergence. And (3) it supports load-aware task scheduling to balance workloads among threads in warps and blocks. We describe them in detail in the following sections.

4.1 Stack-Based Iteration with Node Reuse

To reduce memory usage, we propose a stack-based iteration 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 Figure 2. We describe this stack-based MBE algorithm with node reuse in Algorithm 2.

Algorithm 2: Stack-based Iterative MBE Algorithm

```
   Data: Bipartite graph G(U, V, E)
   Input: Set L_r ⊆ U, disjoint sets R_r, C_r ⊆ V
   Output: All maximal bicliques
1  procedure iteratively_search(L_r, R_r, C_r):
2      node_buf.init_and_push((L_r, R_r, C_r));
3      while node_buf is not empty do                    // Iteration
4          (L_p, R_p, C_p) ← node_buf.pop() ;
5          if C_p is not empty then
6              v' ← the smallest vertex in C_p ;
7              node_buf.push((L_p, R_p, C_p \ {v'}));
8              L' ← L_p ∩ N(v'); R' ← R_p; C' ← ∅;
9              foreach v_c ∈ C_p do                   // Node generation
10                 if L' ∩ N(v_c) == L' then
11                     R' ← R' ∪ {v_c};
12                 else if L' ∩ N(v_c) ≠ ∅ then
13                     C' ← C' ∪ {v_c};
14             if R' == Γ(L') then                    // Maximality check
15                 Output(L', R') as a maximal biclique;
16                 node_buf.push((L', R', C')) ;
```

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 Figure 5. A node_buf consists of a root node (L_r, R_r, C_r), the attribute *depth* of each vertex in $L_r \cup R_r \cup C_r$, 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. Next, we illustrate the key functions designed for node reuse.

- **init_and_push(**(L_r, R_r, C_r)**)**: this function is used for creating and initializing a node_buf using the root node (L_r, R_r, C_r) of a subtree (line #2 in Algorithm 2). The node_buf stores all vertices in $L_r \cup R_r \cup C_r$ and actively tracks the *depth* for each vertex. We initialize the *depth* for vertices in $L_r \cup R_r$ to 0 and the *depth* for vertices in C_r 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_r to D if they are also in L', and then update the *depth*s of vertices in C_r 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, R_r, C_r) by the *depth* field, as L' contains all vertices in L_r whose *depth* is equal to D, and R' contains all vertices in $R_r \cup C_r$ whose *depth* is not greater than D, and C' contains all vertices in C_r 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_p, R_p, C_p) and then backtrack to its parent node using node_buf (line #4 in Algorithm 2). When we pop a node (L_p, R_p, C_p) at depth D, we first remove the latest traversed vertex v' in node_buf. Then, we update the *depth* for vertices in L_p to $D-1$ and update the *depth* for vertices in C_p with *depth* D to ∞.

Discussion. 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. For instance, an A100 GPU of 40 GB memory is adequate to run over 10k of subtree traversal procedures on the BookCrossing dataset [25] 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 3.1 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.

4.2 Pruning using Local Neighborhood Size

The existing pruning approaches are inefficient on GPUs because of thread divergence. To address this issue, we propose a new pruning approach to reduce enumeration space and thread divergence. 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).

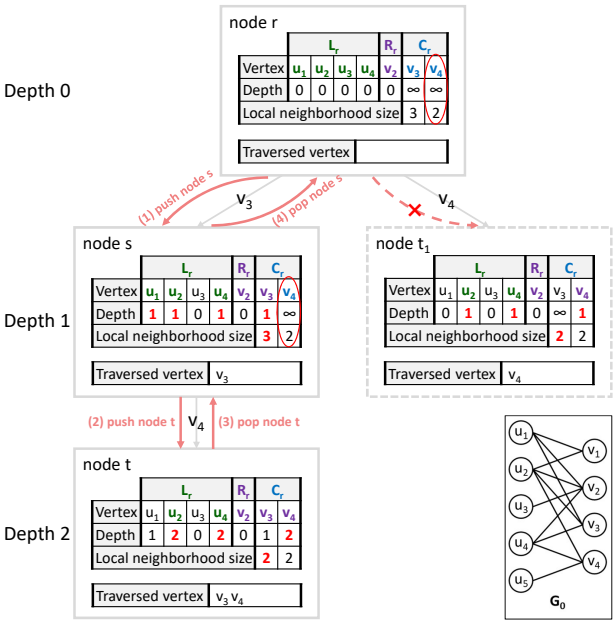


Figure 5: Illustration of GMBE with pruning using local neighborhood size.

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. We will formulate it in Theorem 4.1. 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. The new pruning approach has low thread divergence because threads in a warp always check elements in the same candidate set. We use an example to detail the stack-based iteration algorithm with pruning using local neighborhood size.

Example 4.1. *Figure 5 illustrates that GMBE iteratively enumerates all maximal bicliques in the subtree rooted by node r in Figure 2 with the fixed memory in node_buf. Specifically, GMBE initializes node_buf using node r, where* $L_r = L_{(r)}$, $R_r = R_{(r)}$, *and* $C_r = C_{(r)}$. *Node_buf then initializes the depth for vertices in* $L_r \cup R_r$ *to 0 and initializes the depth for vertices in* C_r *to* ∞. *Node_buf initializes the local neighborhood size (i.e.,* $|N_L|$*) for vertices in* C_r *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_r *and* R_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.*

Algorithm 3: Naive approach for launching GPU tasks

```
foreach v_s ∈ V do  // Initialize |V| tasks independently
1   L_s ← N(v_s); R_s ← {v_s}; C_s ← ∅; // Node generation
2   foreach v_c ∈ N_2(v_s) do
3       if L_s ∩ N(v_c) == L_s then
4           R_s ← R_s ∪ {v_c};
5       else if v_c is with later order than v_s then
6           C_s ← C_s ∪ {v_c};
7   if v_s is the smallest vertex in R_s then  // Maximality check
        // Map each task to a warp or a block
8       iteratively_search(L_s, R_s, C_s) ;
```

Theorem 4.1. *Assume the current node is p in the enumeration tree. GMBE can safely prune its child nodes generated by choosing a candidate vertex* v_c *in* $C_{(p)}$ *if* $|N_L(v_c)|$ *for the current node p is equal to* $|N_L(v_c)|$ *for any of the child nodes of p.*

Proof. To prove the theorem, we assume that node s has traversed v_m to generate node t and node s will traverse v_n to generate node r. We further assume that $|N_L(v_n)|$ for node s is equal to $|N_L(v_n)|$ for node t. Because $|N_L(v_n)|$ for node s is equal to $|N_L(v_n)|$ for node t, $|N(v_n) \cap L_{(s)}| = |N(v_n) \cap L_{(t)}|$. Because $L_{(t)} = L_{(s)} \cap N(v_m)$, $|N(v_n) \cap L_{(s)}| = |N(v_n) \cap L_{(t)}| = |N(v_n) \cap L_{(s)} \cap N(v_m)|$. Then, we know $N(v_n) \cap L_{(s)} \subseteq N(v_m)$. Because $L_{(r)} = N(v_n) \cap L_{(s)}$, $L_{(r)} \subseteq N(v_m)$. Thus, we know that v_m connects with all vertices in $L_{(r)}$. Node r fails to expand $R_{(r)}$ with v_m because GMBE has traversed v_m generate node t and we can safely prune node r, which produces a non-maximal biclique. □

4.3 Load-Aware Task Scheduling

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 ($v_s \in V$). The naive approach is shown in Algorithm 3. We can then map these tasks to warps [15] or blocks [11] 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. We observe that 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 Figure 9 in Section 6.2.

To address this issue, we propose a **load-aware task-centric** approach for MBE using the persistent thread (PT) [21] 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

Algorithm 4: Load-aware task-centric scheme in GMBE

```
   processing_v is a global variable initialized as 0 ;
   SM_task_queue is a global concurrent queue for load balance ;
   // For each warp
1  procedure warp_kernel:
2      while true do
3          if SM_task_queue is not empty then
4              (L, R, C) ← SM_task_queue.dequeue() ;
5          else
6              v_s = atomicInc(processing_v) ;
7              if v_s ∈ V then
8                  L ← N(v_s); R ← {v_s}; C ← ∅;
9                  foreach v_c ∈ N_2(v_s) do
10                     if L ∩ N(v_c) == L then
11                         R ← R ∪ {v_c};
12                     else if v_c is with later order than v_s then
13                         C ← C ∪ {v_c};
14             else  // All tasks on GPU have been processed
15                 return;
16         if R == Γ(L) then                    // Maximality check
17             if min{|L|, |C|} × |C| > bound_size and
                min{|L|, |C|} > bound_height then
18                 foreach v_t ∈ C do           // Node generation
19                     L_t ← N(v_t); R_t ← R; C_t ← ∅;
20                     foreach v_c ∈ C do
21                         if L_t ∩ N(v_c) == L_t then
22                             R_t ← R_t ∪ {v_c};
23                         else if L_t ∩ N(v_c) ≠ ∅ then
24                             C_t ← C_t ∪ {v_c};
25                     SM_task_queue.enqueue((L_t, R_t, C_t)) ;
26                     C ← C \ {v_t};
27             else
28                 iteratively_search(L, R, C);
```

Section 6.4. We develop a GPU kernel, which can 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 *iteratively_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 #17), GMBE directly launches a GPU task (line #28). If the size of the enumeration tree is larger than the bounds, it enqueues the root node of sub-trees to be enumerated (lines #18-26). 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 #7-13) and launches its corresponding tree onto GPUs.

5 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 [14]. We then pre-process G by sorting all vertices in V using the increasing order of their neighborhood sizes [29, 39] and sorting all neighbor lists of each vertex using increasing order of vertex IDs similar to most of the related works [11, 15]. We finally 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 [28] in CUDA. We implement a two-level task queuing mechanism to further improve load balance. Specifically, we implement 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 can easily 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 [1]. 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. Experimental results in Figure 13 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. Theoretically, GMBE can also be extended to a distributed computing environment, where multiple machines (each with one or more GPUs) are connected by the network. Since this work focuses on the single-machine environment, we leave the exploration of GMBE on distributed multi-machine clusters as our future work.

Table 1: 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

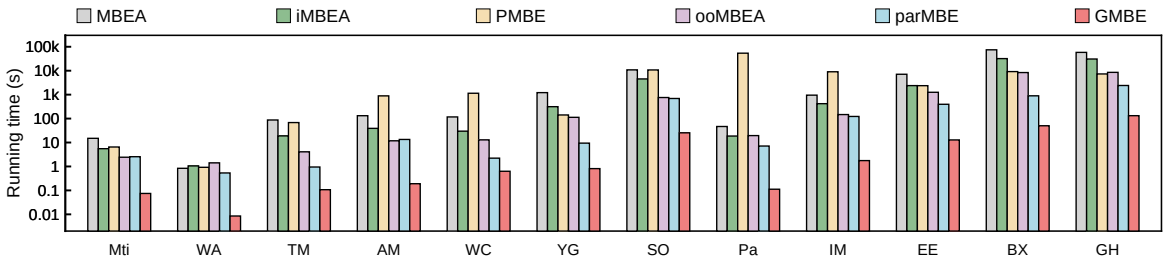


Figure 6: Overall evaluation (log scaled).

6 EVALUATION

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

6.1 Experimental Setup

Platform. By default, we evaluate our GPU implementations on an NVIDIA A100 GPU [4] with 108 streaming multiprocessors (SMs) and 40 GB of global memory. For the 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 12 real-world datasets to justify the performance of GMBE as shown in Table 1. 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 [26] and the other datasets from the KONECT repository [25]. 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. Since there is no existing MBE algorithm working on GPUs, we compare GMBE to the CPU-oriented MBE algorithms, including the recent serial versions, i.e., MBEA [39], iMBEA [39], PMBE [8], and ooMBEA [14], and the cutting-edge parallel MBE algorithm, i.e., ParMBE [18]. 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 iteratively enumerates all maximal bicliques with node reuse, prunes useless nodes using local neighborhood sizes, and applies the load-aware task-centric scheme for load balance. By default, GMBE 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 evaluate the techniques proposed in this paper. We will detail those variants in the corresponding experiments.

6.2 Overall Evaluation

Figure 6 compares the running time of GMBE to state-of-the-art MBE algorithms on real-world datasets. The experimental results show that GMBE is 3.5×–69.8× faster than any next-best competitor on CPUs on all testing datasets because GMBE efficiently utilizes extensive computational resources on GPUs. Specifically, GMBE on a single A100 GPU outperforms the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine by up to 70.6× on ActorMovies. Compared to all the existing MBE algorithms that cost over 40 minutes to enumerate all maximal bicliques on Github, GMBE needs only 132 seconds and thus is helpful for MBE on the large datasets in practice. In addition, we conduct a performance analysis of GMBE using the NVIDIA Nsight Compute software [5]. The profiling results indicate that the average warp execution efficiency is 64%, and the memory utilization is 12% across all real-world datasets. These outcomes can be attributed to the inherent irregularity present in the MBE problem [13].

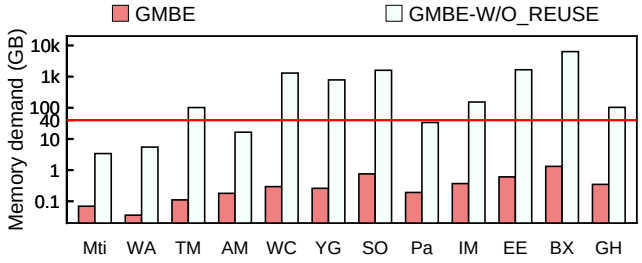


Figure 7: Effect of the node reuse approach (log scaled). The red line indicates the GPU memory capacity of NVIDIA A100.

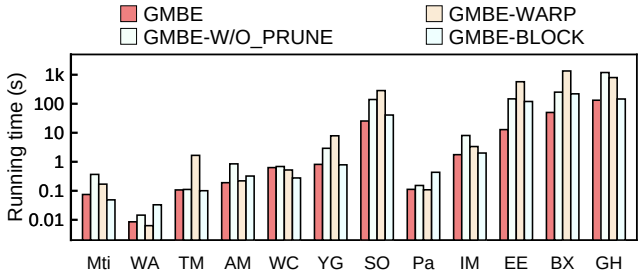


Figure 8: Effect of pruning approach and task scheduling approach (log scaled).

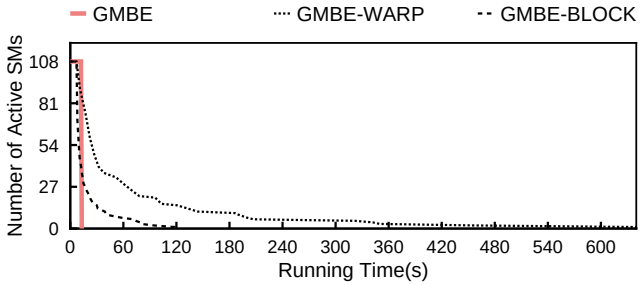
Table 2: Comparison of the ratio of generated non-maximal bicliques to maximal bicliques between GMBE and GMBE-w/o_PRUNE.

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

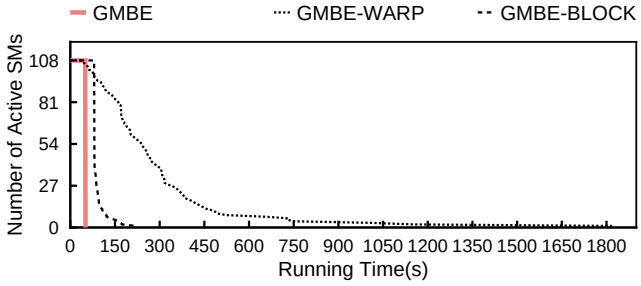
6.3 Effect of Optimizations

Effect of the node reuse approach. To study the effect of the node reuse approach in Section 3.1, we design a variant GMBE-w/o_REUSE that pre-allocates memory on GPUs according to Section 3.1. 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. Figure 7 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 requirements exceed the memory capacity of the A100 GPU on multiple datasets.

Effect of the pruning approach. To study the effect of the local-neighborhood-size-based pruning approach in Section 4.2, we design a variant GMBE-w/o_PRUNE that only disables the pruning function of GMBE. As shown in Figure 8, GMBE constantly outperforms GMBE-w/o_PRUNE because the pruning approach prunes enumeration space for MBE at runtime with less thread divergence.



(a) EuAll



(b) BookCrossing

Figure 9: Comparison of runtime loads on SMs among GMBE, GMBE-WARP, and GMBE-BLOCK.

In addition, the pruning approach enhances memory access coalescence by comparing the sizes of local neighborhoods among vertices in the same candidate vertex set. To further explore the pruning efficiency, we use α to represent the number of maximal bicliques and δ to represent the number of pruned non-maximal bicliques generated 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 GMBE and GMBE-w/o_PRUNE, as presented in Table 2. By comparing the ratio δ/α of both GMBE and GMBE-w/o_PRUNE, we observe 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, the pruning approach significantly reduces the running time on Github from 1,191 seconds to 132 seconds.

Effect of the task scheduling approach. To study the effect of the load-aware task scheduling approach in Section 4.3, we design two variants GMBE-WARP and GMBE-BLOCK that apply warp-centric and block-centric schemes respectively. As shown in Figure 8, GMBE is significantly faster than GMBE-WARP and GMBE-BLOCK on large datasets, including EuALL, Github, BookCrossing, StackOverflow, and IMDB, and spends less than one second on the other datasets. GMBE is more performant on large datasets because it dynamically detects and partitions the tasks having heavy loads and manages lock-free task queues to rebalance the workloads in finer granularity. As a result, GMBE is 44.7× and 9.3× faster than GMBE-WARP and GMBE-BLOCK on EuAll, respectively.

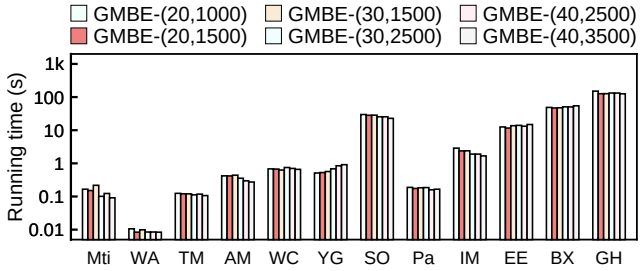


Figure 10: Impact of thresholds *bound_height* and *bound_size* for load-aware task scheduling (log scaled).

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. Figure 9 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 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 80% of running time (98s / 118s) 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.

6.4 Sensitivity Analysis

Impact of thresholds for load-aware task scheduling. To explore the efficient configuration for thresholds *bound_height* and *bound_size* in Section 4.3, 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 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. Figure 10 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 4.3, 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. Figure 11 shows that the variant GMBE(16) outperforms the other variants by up to 3.83× on most large datasets, such as BookCrossing, StackOverflow, IMDB, DBLP, and EuAll. In addition, GMBE(16) is 0.94× slower than GMBE(32) 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.

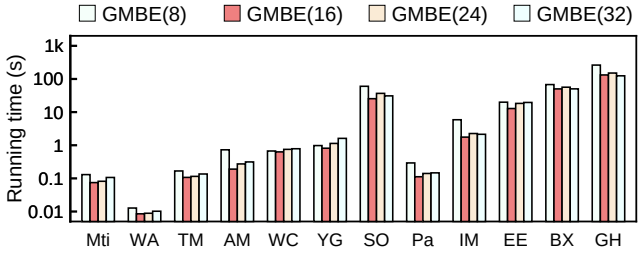


Figure 11: Impact of the parameter WarpPerSM (log scaled).

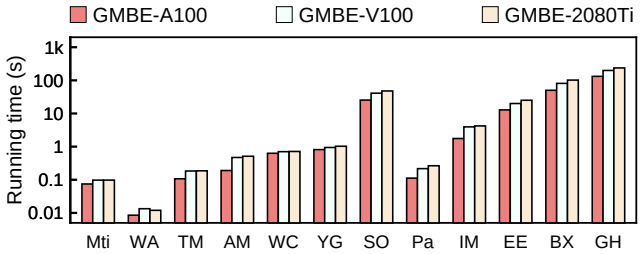
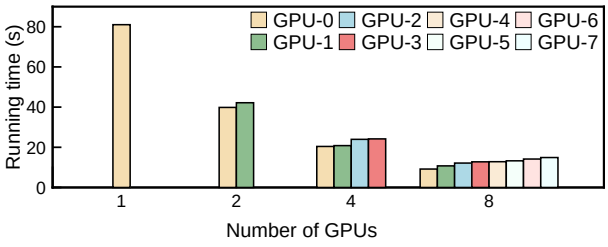


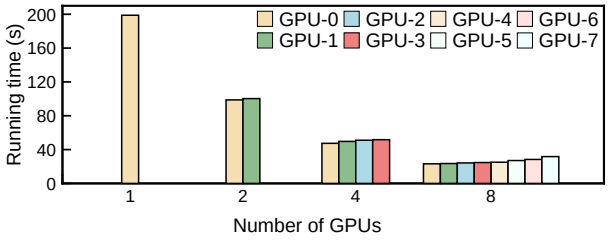
Figure 12: Adaptability on different GPU (log scaled).

Adaptability on different GPUs. To explore the adaptability of GMBE, we evaluate GMBE on an NVIDIA A100 GPU (108 SMs, 40 GB global memory), an NVIDIA V100 GPU (80 SMs, 32 GB global memory) [6], and an NVIDIA 2080Ti GPU (68 SMs, 11 GB global memory) [3], respectively. Figure 12 shows that GMBE is adaptive on all three GPUs. GMBE-A100 is slightly faster than both GMBE-V100 and GMBE-2080Ti because an A100 GPU contains more computational resources 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 V100 GPUs. 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. Figure 13 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 Github dataset within 31 seconds, meaning a performance speedup of 77× compared to the state-of-the-art parallel MBE algorithm ParMBE on a 96-core CPU machine (i.e., 2,411 seconds).



(a) BookCrossing



(b) Github

Figure 13: Scalability of GMBE on a machine with multi-GPU.

7 RELATED WORK

Maximal Biclique Enumeration on CPUs. The most efficient class of MBE algorithms [8, 14, 18, 29, 31, 32, 39] is based on backtracking with recursions on CPUs. Most of the research efforts [8, 14, 29, 39] applied various sorting and pruning techniques on a serial MBE algorithm to reduce the search space for the MBE problem. Some other works [18, 32] 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 3, 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) [12, 27, 36] and graph pattern mining (GPM) [11, 15, 20], it is still challenging for MBE on GPUs. Specifically, MCE on GPUs suffers from similar performance issues as MBE described in Section 3. As a result, the latest GPU-based MCE algorithm GBK [36] only obtained a suboptimal performance, which is comparable 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^2Miner [15] cannot address the memory issue and severe load imbalance for MBE on GPUs.

8 CONCLUSION

In this paper, we present GMBE, the first highly-efficient 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, a pro-active pruning method using local neighborhood size, and a load-aware task scheduling framework to achieve load balance among threads within warps and blocks. We conduct comprehensive evaluations using 12 real-world datasets and three different GPUs. Our experimental results show that GMBE outperforms the state-of-the-art parallel MBE algorithm ParMBE by 70.6×.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their constructive feedback and suggestions. This work was supported by the National Key Research and Development Program of China under Grant 2021ZD0110700, the National Science Foundation of China under Grant 62172361, the Program of Zhejiang Province Science and Technology under Grant 2022C01044, the Zhejiang Lab Research Project under Grant 2020KC0AC01, and the US National Science Foundation under CNS 2216108, CNS 1906541, and OAC 2243980.

REFERENCES

[1] 2022. CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[2] 2023. CUDA from Wikipedia. https://en.wikipedia.org/wiki/CUDA.
[3] 2023. GeForce RTX 20 Series. https://www.nvidia.com/en-gb/geforce/20-series/.
[4] 2023. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-gb/data-center/a100/.
[5] 2023. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute/.
[6] 2023. NVIDIA V100 Tensor Core GPU. https://www.nvidia.com/en-gb/data-center/v100/.
[7] 2023. Single instruction, multiple threads (SIMT) from Wikipedia. https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads.
[8] Aman Abidi, Rui Zhou, Lu Chen, and Chengfei Liu. 2020. Pivot-based Maximal Biclique Enumeration.. In *IJCAI*. 3558–3564.
[9] Gabriela Alexe, Sorin Alexe, Yves Crama, Stephan Foldes, Peter L Hammer, and Bruno Simeone. 2004. Consensus algorithms for the generation of all maximal bicliques. *Discrete Applied Mathematics* 145, 1 (2004), 11–21.
[10] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Elisa Bertino, and Norman Foo. 2013. Collusion detection in online rating systems. In *Web Technologies and Applications: 15th Asia-Pacific Web Conference, APWeb 2013, Sydney, Australia, April 4-6, 2013. Proceedings 15*. Springer, 196–207.
[11] Mohammad Almasri, Izzat El Hajj, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2022. Parallel k-clique counting on gpus. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.
[12] Tariq Alusaifeer, Sheela Ramanna, Christopher J Henry, and James Peters. 2013. GPU implementation of MCE approach to finding near neighbourhoods. In *International Conference on Rough Sets and Knowledge Technology*. Springer, 251–262.
[13] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.
[14] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, and Jianxin Li. 2022. Efficient maximal biclique enumeration for large sparse bipartite graphs. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1559–1571.
[15] Xuhao Chen et al. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.
[16] Xuhao Chen, Li-Wen Chang, Christopher I Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *2014 47th Annual IEEE/ACM international symposium on microarchitecture*. IEEE, 343–355.
[17] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the ACM International Conference on Supercomputing*. 378–391.
[18] Apurba Das and Srikanta Tirthapura. 2019. Shared-memory parallel maximal biclique enumeration. In *2019 IEEE 26th International Conference on High Performance Computing (HiPC)*. IEEE, 34–43.
[19] David Eppstein. 1994. Arboricity and bipartite subgraph listing algorithms. *Information processing letters* 51, 4 (1994), 207–211.
[20] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting reuse for GPU subgraph enumeration. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 34, 9 (2020), 4231–4244.
[21] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel*

Computing (InPar). 1–14.

[22] Danny Hermelin and George Manoussakis. 2021. Efficient enumeration of maximal induced bicliques. *Discrete Applied Mathematics* 303 (2021), 253–261.

[23] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[24] Kyle Kloster, Blair D Sullivan, and Andrew van der Poel. 2019. Mining maximal induced bicliques using odd cycle transversals. In *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM, 324–332.

[25] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[26] Jure Leskovec and Andrej Krevl. 2014. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[27] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E Wes Bethel. 2017. Maximal clique enumeration with data-parallel primitives. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 16–25.

[28] Yuan Lin and Vinod Grover. 2018. Using cuda warp-level primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[29] Guimei Liu, Kelvin Sim, and Jinyan Li. 2006. Efficient mining of large maximal bicliques. In *Data Warehousing and Knowledge Discovery: 8th International Conference, DaWaK 2006, Krakow, Poland, September 4-8, 2006. Proceedings 8*. Springer, 437–448.

[30] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum biclique search at billion scale. *Proceedings of the VLDB Endowment* (2020).

[31] Ziyi Ma, Yuling Liu, Yikun Hu, Jianye Yang, Chubo Liu, and Huadong Dai. 2022. Efficient maintenance for maximal bicliques in bipartite graph streams. *World Wide Web* 25, 2 (2022), 857–877.

[32] Arko Provo Mukherjee and Srikanta Tirthapura. 2016. Enumerating maximal bicliques from a large graph using mapreduce. *IEEE Transactions on Services Computing (TSC)* 10, 5 (2016), 771–784.

[33] Ron Rymon. 1992. Search through systematic set enumeration. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 539–550.

[34] Amos Tanay, Roded Sharan, and Ron Shamir. 2002. Discovering statistically significant biclusters in gene expression data. In *Proceedings of the Tenth International Conference on Intelligent Systems for Molecular Biology*. 136–144.

[35] Kai Wang, Wenjie Zhang, Xuemin Lin, Lu Qin, and Alexander Zhou. 2022. Efficient personalized maximum biclique search. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 498–511.

[36] Yi-Wen Wei, Wei-Mei Chen, and Hsin-Hung Tsai. 2021. Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 32, 9 (2021), 2352–2366.

[37] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. 2021. Are dynamic memory managers on gpus slow? a survey and benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 219–233.

[38] Jianye Yang, Yun Peng, Dian Ouyang, Wenjie Zhang, Xuemin Lin, and Xiang Zhao. 2023. (p, q)-biclique counting and enumeration for large sparse bipartite graphs. *The VLDB Journal* (2023), 1–25.

[39] Yun Zhang, Charles A Phillips, Gary L Rogers, Erich J Baker, Elissa J Chesler, and Michael A Langston. 2014. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. *BMC bioinformatics* 15 (2014), 1–18.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

https://doi.org/10.5281/zenodo.8079259

ARTIFACT IDENTIFICATION

In this paper, we propose GMBE, the first highly-efficient GPU solution for the maximal biclique enumeration (MBE) problem. The computational artifact provides the source code and scripts to reproduce all experimental results.GMBE is implemented in C++ and CUDA. The artifact includes all comparison baselines (i.e., MBEA, iMBEA, PMBE, ooMBEA, and ParMBE) for justifying the performance of GMBE and all variants of GMBE for evaluating all proposed techniques. This artifact enables others to conduct their own studies on any Linux machine with NVIDIA GPUs.

REPRODUCIBILITY OF EXPERIMENTS

1 ARTIFACT CHECKLIST

- **Platforms:**
  - An NVIDIA A100 GPU
  - An NVIDIA 2080Ti GPU
  - 8 NVIDIA V100 GPUs
  - A machine with 96 Xeon(R) Gold 5318Y CPU @ 2.10GHz CPU cores
- **System Details:**
  - 18.04-Ubuntu x86_64 GNU/Linux
  - Linux kernel: 5.4.0
- **Software Dependencies:**
  - GNU Make 4.2.1
  - CMake 3.22.0
  - CUDA toolkit 11.7
  - GCC/G++ 10.3.0
  - Python 2.7.18
  - Python packages: zplot 1.41, pathlib 1.0.1
  - C++ library: libtbb-dev 2020.1-2
  - Ubuntu apt package: Ghostscript, Texlive-extra-utils
  - Nvidia driver 510.85.02
  - Docker 20.10.10

2 DESCRIPTION

The computational artifact provides scripts to reproduce all experimental results of GMBE, including Figures 6-13 and Table 2. Users need to download the source code and scripts from https://github.com/fhxu00/MBE-GPU.git. The following is the directory structure of the repository:

- **README.md:** This file contains a detailed step-by-step "Try out GMBE" guide.
- **src/:** This directory contains the core source code of GMBE implementation.
- **scripts/:** This directory has scripts to run experiments.
- **baselines/:** This directory contains the zipped source codes of the comparison baselines, i.e., MBEA, iMBEA, PMBE, ParMBE, and ooMBEA.
- **preprocess/:** This directory has scripts to preprocess the graph datasets.
- **fig/:** This directory contains the scripts to generate figures.

After downloading the source code and scripts, users need to prepare graph datasets and compile the code using our provided scripts. We provide a docker image in the repository https://hub.docker.com/r/fhxu00/gmbe for the convenience to deploy this project. Please refer to the corresponding *README.md* for a detailed guide.

3 EXPERIMENTAL WORKFLOW

Step 1: Prepare datasets.

We use 12 real-world datasets for our evaluation. Users can download and preprocess them with the script in the directory *preprocess/*, which will take roughly 10 minutes. After executing the script, users can find the preprocessed datasets in the new directory *datasets/*.

Step 2: GPU Setup.

We provide the specifications for setting up the project on A100, V100, and 2080Ti GPUs in the CMakeList.txt file located in the project directory. Users can deploy GMBE on other GPUs by setting the "CUDA_NVCC_FLAGS" and "DMAX_SM" flags for the designated GPU, following the guidelines provided in the GPU manufacturer's documentation. The "DMAX_SM" flag indicates the maximum number of SMs on the GPU.

Step 3: Execute scripts to generate results.

We provide the scripts to automatically generate the experimental results of Figures 6-13 and Table 2 in *scripts/*. It will take roughly 9 hours to generate all the results. The running progress and the expected running time of each experiment would be printed to the file *scripts/progress.txt* automatically. All the experimental results would be printed to the file *scripts/results.txt* and the results required to generate the figure would be printed to the data file in the corresponding subdirectory in *fig/*.

Step 4: Execute scripts to generate figures.

We provide the script to generate figures in the directory *fig/*, executing which will take roughly 30 seconds. The generated figures would be found under the directory *fig/*.

4 EVALUATION AND EXPECTED RESULTS

Users can reproduce all experimental results with the scripts, which should roughly match the figures from the paper. Note that, the results may be not the exact ones presented in the paper. Because in our subsequent experiments, we found that different states of the GPUs may also impact the performance.

ARTIFACT DEPENDENCIES REQUIREMENTS

To conduct the experiments for GMBE, a Linux machine with Nvidia GPUs is required. The experiments of ParMBE require a machine with multi-core CPU and the C++ library libtbb-dev. The input datasets used can be obtained from two open-source repositories:

SNAP and KONECT. These datasets were collected from the real world and are widely used in MBE research.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

The process about how to install, compile and deploy the code is detailed in the 'README.md' of our source code. It takes no more than 10 minutes to complete the process.

OTHER NOTES

The project has also been deployed on ChameleonCloud, which can be found at https://chi.uc.chameleoncloud.org/project/instances/b28ae6f2-aec0-4583-8126-789efcaefb2f/.