Accelerating the Bron-Kerbosch Algorithm for Maximal Clique Enumeration Using GPUs

Yi-Wen Wei, Wei-Mei Chen, and Hsin-Hung Tsai

Abstract—Maximal clique enumeration (MCE) is a classic problem in graph theory to identify all complete subgraphs in a graph. In prior MCE work, the Bron-Kerbosch algorithm is one of the most popular solutions, and there are several improved algorithms proposed on CPU platforms. However, while few studies have focused on the related issue of parallel implementation, recently, there have been numerous explorations of the acceleration of general purpose applications using a graphics processing unit (GPU) to reduce the computing power consumption. In this article, we develop a GPU-based Bron-Kerbosch algorithm that efficiently solves the MCE problem in parallel by optimizing the process of subproblem decomposition and computing resource usage. To speed up the computations, we use coalesced memory accesses and warp reductions to increase bandwidth and reduce memory latency. Our experimental results show that the proposed algorithm can fully exploit the resources of GPU architectures, allowing for the vast acceleration of operations to solve the MCE problem.

Index Terms—Maximal clique, parallel computing, GPU

1 Introduction

Maximal clique enumeration (MCE) is a classic problem in graph theory, which identifies all complete subgraphs for a given graph. Maximal cliques are useful in many applications, such as social graph analysis, pattern detection in terrorist networks, financial network analysis, bioinformatics analysis, and email network analysis. Because these applications commonly deal with situations involving massive data, it is essential to devise an efficient algorithm to solve the MCE problem for large datasets.

Given an undirected graph $G = (V, E)$, a *clique* C is a nonempty subset of V with the property that every two distinct vertices in C are adjacent in G. A *maximal clique* is a clique that is not a subset of any clique except itself. Fig. 1 presents an illustrative example using eight maximal cliques. The sets $\{7, 8, 9\}$ and $\{7, 8, 9, 10\}$ are two cliques, and $\{7, 8, 9, 10\}$ is a maximal clique. Because $\{7, 8, 9\}$ is a subset of the clique $\{7, 8, 9, 10\}$, it is not a maximal clique. In [18], Moon and Moser proved that the exact upper bound of the number of cliques in graphs is $3^{|V|/3}$, which means that the number of cliques grows exponentially with the number of vertices in the worst case. The MCE problem is to list all maximal cliques in a given graph, which was proved to be an NP-hard problem [12].

The MCE problem and its dual problem of finding maximal independent sets have been widely studied [2], [3], [5], [6], [8], [16], [17], [20], [22], [24] for various classes of graphs. In previous studies, the Bron-Kerbosch algorithm [3] is one of the most popular algorithms because it is easy to understand and simple to implement for practical usage. The Bron-Kerbosch algorithm is a backtracking procedure that solves subproblems by dividing vertices into several subsets and determining for all maximal clique solutions recursively. In [3], another version of the Bron-Kerbosch algorithm was introduced based on pivot selection to reduce the expanding solution space. To optimize the running time, several studies have considered variations of pivot selection strategies [11], [19], [23]. The time complexity for the algorithm proposed in [23] is $O(3^{|V|/3})$, which is optimal for the function of the number of vertices. Because using the Bron-Kerbosch algorithm for large graphs may generate massive solutions and require a considerable amount of time, some researchers have made improvements to the Bron-Kerbosch algorithm [4], [9] for large graphs. In [9], a modified Bron-Kerbosch algorithm for large sparse graph was proposed, which ordered all vertices by degeneracy before searching for maximal cliques. While this new algorithm reduced the time complexity to $O(d|V|3^{d/3})$, where d is the degeneracy of the given graph, it still requires a considerable amount of time when d is large.

As technology rapidly advances, computers have progressed from the original single-core architecture to the current multi-core architecture. Recently, there are many studies on CPU-GPU frameworks for various applications, including analytical and practical investigations [14], [15], [27]. Some researchers have begun to examine parallel resolutions of the MCE problem [6], [7], [10], [21], [25], [26], [28]. The parallel MCE algorithm [21] is a widely used parallel approach, which involves handling some basic jobs before dividing the tasks into multi-thread processes to achieve parallelization. However, the number of cores is limited. Therefore, some researchers, such as [26], have attempted to solve the MCE problem by using distributed architectures to speed up the execution time. In general, the distributed system architecture must transmit data between separate computing devices to

• *The authors are with the Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan. E-mail: {D10202101, wmchen, M10802115}@mail.ntust.edu.tw.*

Manuscript received 22 June 2020; revised 2 Jan. 2021; accepted 8 Mar. 2021. Date of publication 18 Mar. 2021; date of current version 1 Apr. 2021. (Corresponding author: Wei-Mei Chen.) Recommended for acceptance by C. Ding. Digital Object Identifier no. 10.1109/TPDS.2021.3067053

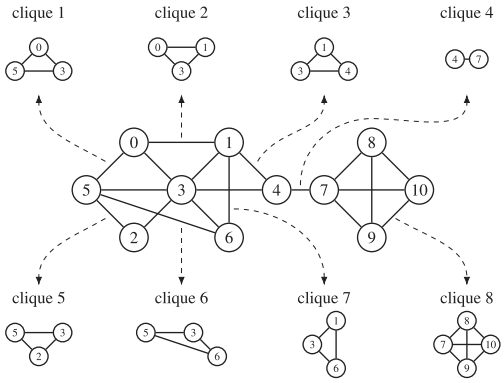

Fig. 1. A graph containing eight maximal cliques.

complete the whole task. Thus, the overall computation time increases due to the required communication cost. Recently, advances in graphics processing units (GPUs) have expanded their applications beyond graphics processing. Compared to multi-core central processing units (CPUs), GPUs contain more cores with a lower frequency for massive data processing. Therefore, many general problems can be solved more efficiently if the algorithm is designed to take full advantage of the GPU computing power to execute tasks in parallel.

In this paper, we propose a parallel algorithm for the MCE problem that effectively utilizes the GPU architecture. We replace recursion with iteration in the search process because executing recursive algorithms on GPU architecture is expensive. Further, we use a binary search to efficiently perform the set manipulations used in the searching process. To reduce memory latency, we coalesce global memory accesses and exchange data in registers. We also attempt to balance the workload for each thread to efficiently utilize streaming multiprocessors (SMs). As a result, the proposed algorithm can significantly increase the speed of MCE.

The remainder of this paper is organized as follows. In Section 2, we review previous works that improve the basic Bron-Kerbosch algorithm. In Section 3, we present the characteristics of GPU architectures. In Section 4, we introduce the proposed algorithm and optimization techniques for implementation. In Section 5, we evaluate the performance of the proposed algorithm for MCE. Section 6 concludes the paper.

2 Bron-Kerbosch Algorithm and its Variants

In this section, we briefly describe the basic Bron-Kerbosch algorithm [3] and three popular variants [9], [23], and we review some parallel implementation of the Bron-Kerbosch algorithm. Without loss of generality, we assume that expanding subtrees of a node follow by the lexicographical order of the vertices.

2.1 Bron-Kerbosch Algorithm

The basic Bron-Kerbosch algorithm [3] solves the MCE problem by examining subgraphs recursively as shown in Algorithm 1. In the algorithm, the status of the processing is recorded in three sets R, P, and X to search and expand maximal cliques without redundant examinations. The set R is the current clique obtained, the candidate set P is the set to be checked, and the duplicate set X is the set already checked. Initially, all vertices are in P, and both R and X are empty. In each recursive call, a vertex v in P is added to R, and P and X are restricted by intersecting with $N(v)$ where $N(v)$ is the set of vertices that are adjacent to v. The set X is used to avoid reporting duplicate maximal cliques because the vertex $v \in P$ is moved to X after v is included in R. The set R is reported as a maximal clique if $P \cup X = \varnothing$ because there is no vertex can be added to form a larger and non-reported clique. By recursively expanding the cliques, all maximal cliques are reported once. In the following, we illustrate the basic Bron-Kerbosch algorithm with the concept of recursion trees.

Algorithm 1. Bron-Kerbosch Algorithm

Require: Clique R, candidate set P, and duplicate set X.
Ensure: Identify all maximal cliques.
1: **procedure** BK(R, P, X)
2: **if** $P \cup X = \varnothing$ **then**
3: report R as a maximal clique
4: **return**
5: **end if**
6: **for** each vertex $v \in P$ **do**
7: BK$(R \cup \{v\}, P \cap N(v), X \cap N(v))$
8: $P \leftarrow P \backslash \{v\}$;
9: $X \leftarrow X \cup \{v\}$;
10: **end for**
11: **end procedure**

Fig. 2 shows the recursion tree of the basic Bron-Kerbosch algorithm for the example given in Fig. 1. Initially, the algorithm starts from the conditions: $R = \varnothing$, $P = V$, and $X = \varnothing$. Since we start with $P \cup X \neq \varnothing$, each vertex v in P is used to expand the current clique and search further. Thus, the root of the tree has 11 branches because $|P| = 11$. Without loss of generality, we process the vertices in numeric order. In the for loop of Algorithm 1, vertex 0 is checked first and the function call BK$(\{0\}, \{1,3,5\}, \varnothing)$ is invoked with $P \cap N(0) = \{1,3,5\}$ and $X \cap N(0) = \varnothing$. Because $P \cup X \neq \varnothing$ in the current status, R is not a maximal clique and the recursive expansion is continued. Next, vertex 1 is examined and a new recursive call BK$(\{0,1\}, \{3\}, \varnothing)$ is activated since $P \cap N(1) = \{1,3,5\} \cap \{0,3,4,6\} = \{3\}$ and $X \cap N(1) = \varnothing \cap \{0,3,4,6\} = \varnothing$. Then, BK$(\{0,1,3\}, \varnothing, \varnothing)$ is invoked recursively after vertex 3 is included. Now, the algorithm reports the first maximal clique founded $\{0,1,3\}$, and returns since $P \cup X = \varnothing$. Note that after each recursive call returns, the corresponding P and X are updated based on $P \leftarrow P \setminus \{v\}$ and $X \leftarrow X \cup \{v\}$. In the recursion tree of Fig. 2, a double-line rectangle indicates that the corresponding recursive call outputs a maximal clique.

2.2 Bron-Kerbosch Algorithm With a Pivot

Bron and Kerbosch also proposed an improved algorithm that reduces the number of expansions using a pivot, which is described in Algorithm 2 with the pivoting rule proposed in [23]. The time complexity of the Bron-Kerbosch algorithm with a pivot is $O(3^{|V|/3})$ in [23]. In general, selecting a pivot provides smaller branch factors of the recursion tree, so the number of recursive calls made by the algorithm can be reduced, saving significant running time. To further optimize the efficiency, several studies have suggested variations and pivot selection strategies in [19].

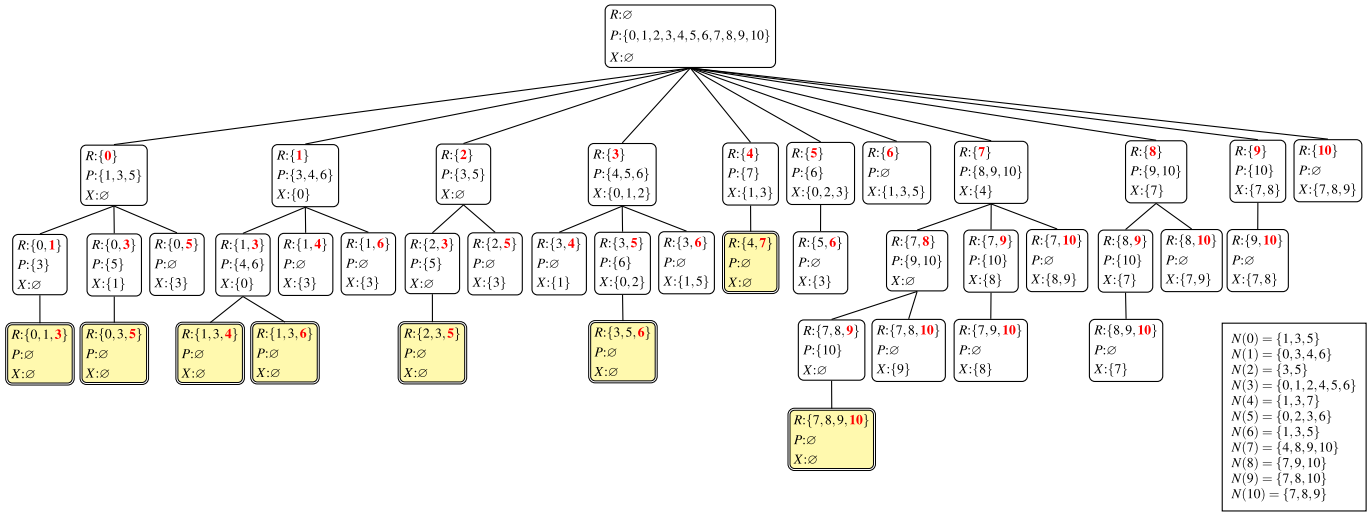

Fig. 2. The recursion tree of the basic Bron-Kerbosch algorithm.

Algorithm 2. Bron-Kerbosch Algorithm With a Pivot

Require: Clique R, candidate set P, and duplicate set X.
Ensure: Identify all maximal cliques.

```
1: procedure BKP(R, P, X)
2:     if P ∪ X = ∅ then
3:         report R as a maximal clique
4:         return
5:     end if
6:     u ← arg max_{v ∈ P∪X} |P ∩ N(v)|
7:     for each vertex v ∈ P\N(u) do
8:         BKP(R ∪ {v}, P ∩ N(v), X ∩ N(v))
9:         P ← P\{v}
10:        X ← X ∪ {v}
11:    end for
12: end procedure
```

Fig. 3 shows the recursion tree of the Bron-Kerbosch algorithm with a pivot for the graph in Fig. 1. The initialization is the same as the original Bron-Kerbosch algorithm. Vertex 3 is selected as a pivot because $|P \cap N(3)| = 6$ is the maximum number of all possible intersections for all vertices in $P \cup X$. Consequently, the algorithm does not expand the recursion tree for the vertices in $N(3) = \{0, 1, 2, 4, 5, 6\}$. In this situation, only the vertices in $P \backslash N(3) = \{3, 7, 8, 9, 10\}$ are checked for maximal cliques. Because the examinations of maximal clique for the neighbors of vertex 3 have already been performed in the process for vertex 3, these operations can be removed. In this example, the total number of recursive calls of the Bron-Kerbosch algorithm with a pivot, which is 19, is significantly less than that of the basic Bron-Kerbosch algorithm, which is 42.

2.3 Bron-Kerbosch Algorithm With Degeneracy

To make the Bron-Kerbosch algorithm with a pivot more efficient in practice for large sparse graphs, the study in [9] suggested that the outer-most recursive calls can be expanded according to the *degeneracy ordering*. This algorithm, called BKD, is rephrased and given in Algorithm 3,

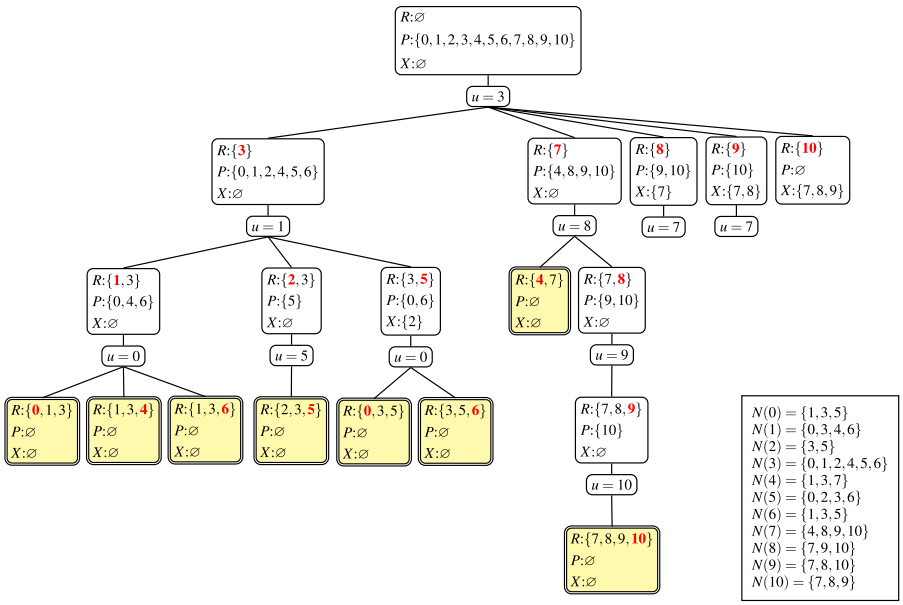

Fig. 3. The recursion tree of the Bron-Kerbosch algorithm with a pivot.

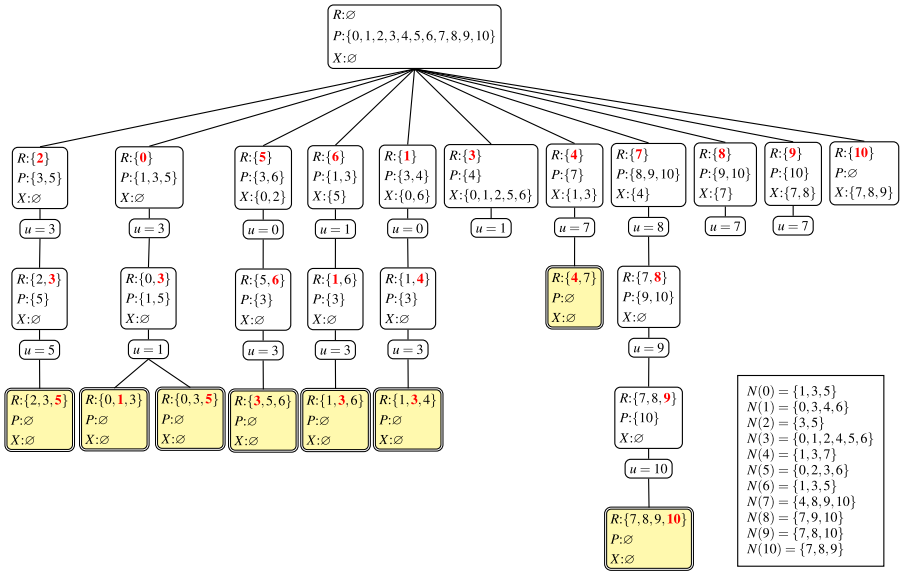

Fig. 4. The recursion tree of the Bron-Kerbosch algorithm with degeneracy.

which quickly divides the recursion tree into smaller subtrees in the first-level and minimizes the number of maximal cliques reported in each recursion subtree. The degeneracy of a graph G is defined as the smallest value d such that every nonempty subgraph of G includes a vertex of degree at most d. Degeneracy ordering can be obtained through a recursive procedure: picking the vertex with the smallest degree in the graph, removing it and all edges adjacent to it, and repeating the previous steps until no vertex remains. If we have a degeneracy ordering $v_0, v_1, \ldots, v_{|V|-1}$, the algorithm invokes BKP in Algorithm 2 for each vertex v_i in V with $R = \{v_i\}$, $P = N(v_i) \cap \{v_{i+1}, v_{i+2}, \ldots, v_{|V|-1}\}$, and $X = N(v_i) \cap \{v_0, v_1, \ldots, v_{i-1}\}$. Thus, for vertex v_i, all vertices listed before it in the degeneracy ordering are directly skipped. Consequently, the number of expansions determined by $|P|$ is minimized, and the time complexity is $O(d|V|3^{d/3})$ where d is the degeneracy of the graph.

Algorithm 3. Bron-Kerbosch Algorithm With Degeneracy

Require: Graph $G = (V, E)$.
Ensure: Identify all maximal cliques in G.
1: **procedure** BKD(G)
2: **for** each vertex v_i in V **do** // degeneracy ordering $v_0, v_1, \ldots, v_{|V|-1}$
3: $P \leftarrow \{v_{i+1}, v_{i+2}, \ldots, v_{|V|-1}\}$
4: $X \leftarrow \{v_0, v_1, \ldots, v_{i-1}\}$
5: BKP($\{v_i\}, P \cap N(v_i), X \cap N(v_i)$)
6: **end for**
7: **end procedure**

The Bron-Kerbosch algorithm with degeneracy shows great improvement for large sparse graphs that contain a number of vertices with low degeneracy. By degeneracy ordering, the given graph is divided into smaller subgraphs, which reduces the computation of intersections in recursive calls. Fig. 4 shows the recursion tree of the Bron-Kerbosch algorithm with degeneracy for the graph in Fig. 1. Note that the total number of recursive calls in the Bron-Kerbosch algorithm with degeneracy is 26, which is slightly higher than that of the Bron-Kerbosch algorithm with a pivot because this example graph is not sparse.

2.4 Parallel Variant of the Bron-Kerbosch Algorithm

To explore the power of parallelism, the literature has investigated MCE on multi-core architectures, distributed architectures, and GPU architectures. The multi-core version of the Bron-Kerbosch algorithm was introduced in [21]. The original Bron-Kerbosch algorithm enumerates maximal cliques by visiting vertices recursively. The search paths of the Bron-Kerbosch algorithm are expanded as a recursion tree structure. In the multi-core variant algorithm, the recursion tree is decomposed into several recursion subtrees, and each subtree is explored by one thread. Thus, each thread maintains a subgraph and enumerates all maximal cliques in its corresponding subtree. To ensure the exploration of the recursion tree is divided evenly among threads, the recursion subtree might be reassigned among threads. Recently, shared-memory parallel algorithms for MCE are proposed in [6] with high work-efficiency for static and dynamic graphs.

The Bron-Kerbosch algorithm can also be implemented on GPU architectures, as proposed in [10]. The search tree is decomposed into several recursion subtrees, and subtrees are explored in parallel. However, in that example, the GPU was only used to accelerate the intersection computations, whereas the exploration scheduling and decomposition of the recursion tree were handled by CPU. Thus, a large amount of data was transferred between the CPU and GPU, which does not fully utilize the GPU architecture. The challenge of accelerating the Bron-Kerbosch algorithm using GPU is considerable due to irregular expansions and unbalanced computations, both of which depend on characteristics of the graphs.

3 GPU Architectures

In this section, we briefly introduce the compute unified device architecture (CUDA), which is a single instruction,

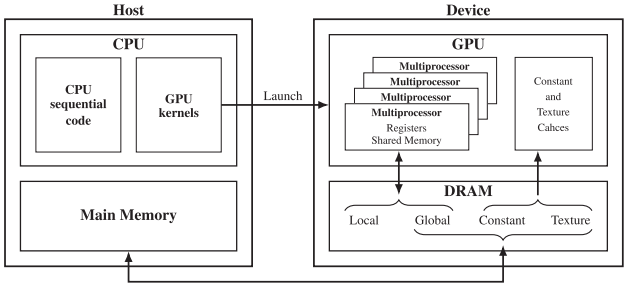

Fig. 5. The compute unified device architecture.

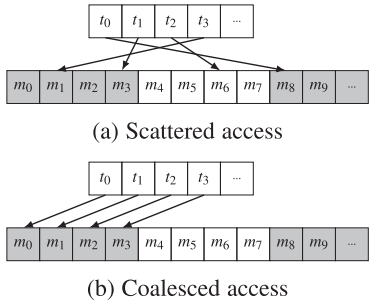

Fig. 7. Accessing global memory.

multiple data (SIMD) computing platform and application programming interface created by NVIDIA, as shown in Fig. 5.

In CUDA, a CPU and GPU can work together in a master/slave mode, where the CPU is called the host and the GPU is called the device. The GPU kernels are the parallel code executed by a grid on the GPU, which is launched by the CPU. As shown in Fig. 6, a grid consists of multiple blocks, and a block consists of multiple threads. Both blocks and threads can be identified using one-dimensional, two-dimensional, or three-dimensional indices. However, the multiprocessors create, manage, schedule, and execute threads in groups of 32 parallel threads called warps. Since a warp executes a common instruction simultaneously, the GPU is working at full efficiency when all 32 threads agree on their execution path. If the threads of a warp diverge via a conditional branch, the warp executes each path taken and disables threads not on that path.

To increase I/O bandwidth and reduce memory latency, GPUs offer different kinds of memory spaces, as shown in Fig. 5. First, data transferred from the host are stored in three kinds of off-chip memory: global memory, constant memory, and texture memory, where constant memory and texture memory are read-only. These memory spaces can be accessed by all threads in a grid, where global memory is the biggest memory space with the highest latency. To maximize the bandwidth of global memory, accessing global memory in a coalesced way is more efficient than a random way, as shown in Fig. 7. The data loaded from global memory is first stored in a cache before processing, and the chunks of memory handled by the cache are called cache lines. Assume that the cache line size is four bytes and the scattered access reads three cache lines, while the coalesced access reads only one cache line. The GPU also provides two kinds of on-chip memory with lower latency: shared memory and a register. Shared memory is used to exchange data between threads in the same block. Note that bank conflicts should be avoided when using shared memory, which seriously reduces memory bandwidth. A register is the fastest memory for each thread. Although a register can only be used by its own thread in most cases, we can exchange the contents of registers in a warp using shuffle instructions. This dramatically accelerates computations for synchronizations, such as prefix-sum operations or broadcasts.

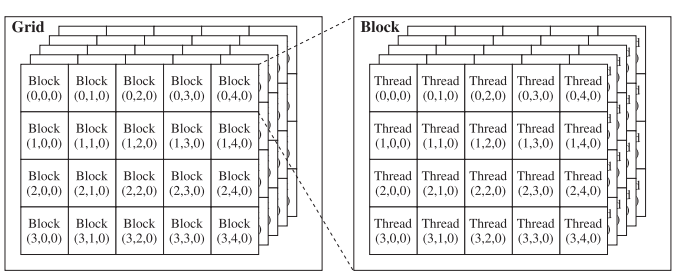

Fig. 6. Grid of thread blocks.

4 Parallel MCE Algorithm

In this section, we first introduce a parallel Bron-Kerbosch algorithm that generates all and only maximal cliques without duplication for a given graph. Then, we demonstrate our GPU-based Bron-Kerbosch algorithm, which fully utilizes the computing power of GPU architectures in parallel to accelerate the algorithm computations.

4.1 Parallel Bron-Kerbosch Algorithm

According to the process of BKP in Algorithm 2, a parallel version of the BKP section of the algorithm can be developed based on expanding behavior for the recursive calls. Algorithm 4 describes the parallel BKP algorithm (called PBKP), where the order of recursive calls invoked by the BKP algorithm is the order of nodes visited for the depth-first search on the recursive tree. Each recursive call of the BKP section considers expansion of the vertices in $P \backslash N(u)$, one by one in some implicit order (see Line 7 in Algorithm 2). For a given graph, if following the processing order of vertices for every *for* loop in the BKP section, our proposed PBKP expands the same recursion tree that is produced by the BKP algorithm.

PBKP contains three stages: maximal clique examination, pivot selection, and clique expansion. The tasks of each stage can be executed in parallel. After a pivot u is picked in pivot selection, the main goal of the *for* loop is to extract the vertices from $Q(= P \backslash N(u))$ to efficiently expand the same recursion tree produced by BKP and eliminate duplicate tasks. Suppose that Q is arranged in the order $v_0, v_1, \ldots, v_{|Q|-1}$. For each vertex $v_i \in Q$, we update the P_i and X_i as $P_i \leftarrow N(u) \cup \{v_i, v_{i+1}, \ldots, v_{|Q|-1}\}$ and $X_i \leftarrow X \cup \{v_0, v_1, \ldots, v_{i-1}\}$ because the vertices listed before v_i in the order have been examined and can be removed later in BKP. Here, we set the boundary case of $\{v_0, v_1, \ldots, v_{i-1}\}$ as $\varnothing$ if $i = 0$. Initially, PBKP solving the MCE problem for a graph G starts with the function call PBKP$(\varnothing, V, \varnothing)$.

Consider the sequential version of PBKP (called PBKP$_S$), which is devised by removing the keyword "parallel" in Line 8 of Algorithm 4.

Algorithm 4. Parallel Bron-Kerbosch Algorithm With a Pivot

Require: Graph G, clique R, candidate set P, and duplicate set X.
Ensure: Identify all maximal cliques in parallel.
1: **procedure** PBKP(R, P, X)
2: **if** $P \cup X = \varnothing$ **then** //1. *maximal clique examination*
3: report R as a maximal clique
4: **return**
5: **end if**
6: $u \leftarrow \arg\max_{v \in P \cup X} |P \cap N(v)|$ //2. *pivot selection*
7: $Q \leftarrow P \backslash N(u)$
8: **for** all vertex $v_i \in Q$ **parallel do** //3. *clique expansion*
// Suppose that the order of Q is $v_0, v_1, \ldots, v_{|Q|-1}$
9: $P_i \leftarrow N(u) \bigcup \left(\bigcup_{j=i}^{|Q|-1} \{v_j\} \right)$
10: $X_i \leftarrow X \bigcup \left(\bigcup_{j=0}^{i-1} \{v_j\} \right)$
11: PBKP($R \cup \{v_i\}, P_i \cap N(v_i), X_i \cap N(v_i)$)
12: **end for**
13: **end procedure**

Next, we will show that PBKP$_S$ and PBKP complete the same operations as BKP.

Lemma 1. *Given a graph G, BKP generates all and only maximal cliques without duplication.*

Proof. See [23] by Tomita *et al*. □

Lemma 2. *Consider the* for *loop in BKP. Suppose that $Q = P \backslash N(u)$ and the vertices of Q are processed following the order $v_0, v_1, \ldots, v_{|Q|-1}$. For the iteration of vertex v_k where $k = 0, 1, \ldots, |Q| - 1$, the corresponding sets P and X before entering the recursive call BKP(·) are $N(u) \bigcup (\bigcup_{j=k}^{|Q|-1} \{v_j\})$ and $X_o \bigcup (\bigcup_{j=0}^{k-1} \{v_j\})$, respectively, where X_o is the set X before entering the* for *loop.*

Proof. From Lines 9 and 10 in BKP, we see that each checked vertex is removed from P and inserted into X at the end of its iteration. For vertex v_k, the vertices listed before it that have been examined can be removed from P and inserted into X after their recursive call BKP(·). Then, the corresponding sets P and X before entering the recursive call BKP(·) are $P_o \backslash (\bigcup_{j=0}^{k-1} \{v_j\})$ and $X_o \bigcup (\bigcup_{j=0}^{k-1} \{v_j\})$, respectively, where P_o is the set P before entering the *for* loop. Furthermore, since $P_0 = N(u) \cup \{v_0, v_1, \ldots, v_{|Q|-1}\}$, we have

$$P_o \backslash \left(\bigcup_{j=0}^{k-1} \{v_j\} \right) = N(u) \bigcup \left(\bigcup_{j=k}^{|Q|-1} \{v_j\} \right).$$

□

Lemma 3. *For a given graph, the sequential version of PBKP (called PBKP$_S$), which is devised by removing the keyword "parallel" in Line 8 of Algorithm 4, reports the same maximal cliques as BKP.*

Proof. The two algorithms, BKP and PBKP$_S$, do the same actions except in their *for* loop. By Lemma 2, two sequential algorithms invoke the same recursive calls in their *for* loop. Note that the last two statements of the last iteration of the loop do not change the output of the maximal cliques. Thus, the maximal cliques reported by PBKP$_S$ are exactly what is produced by BKP. □

Theorem 1. *PBKP generates all and only maximal cliques without duplication.*

Proof. (Sketch) Consider the operations in the *for* loop. Suppose that the vertices of Q are given in some order. For any iteration, P_i and X_i are ready to be computed because all parameters needed are known. Since the operations of an iteration are independent of those of other iterations, all of them can be executed in parallel to improve performance. Thus, by Lemmas 1 and 3, the result is obtained. □

PBKP expands the corresponding recursive tree in parallel. Starting from the root, all child nodes of a node can be examined simultaneously. Every non-root node can be checked to see if its ancestors have all been checked. Moreover, the set P_i is expressed as the union of two disjoint sets, $N(u)$ and $\bigcup_{j=k}^{|Q|-1} \{v_j\}$, in Line 9 of Algorithm 4. This operation can be simplified as a merge operation instead of a more time-consuming union operation when we later implement the algorithm on a GPU.

To increase the degree of parallelism of PBKP, we propose PBKD in Algorithm 5 to perform the first-level recursive calls in degeneracy order, which can quickly split the original problem into several subproblems. Thus, the recursion tree is quickly divided into smaller subtrees, and the number of expansions for each subtree is reduced. The strategies of memory management and workload distribution are significant features impacting efficient implementation of the algorithm on the GPU. Therefore, we introduce the data structure used in the GPU and the implementation details in the following subsection.

Algorithm 5. Parallel Bron-Kerbosch Algorithm With Degeneracy

Require: Graph $G = (V, E)$. // V is in degeneracy order.
Ensure: Identify all maximal cliques in G in parallel.
1: **procedure** PBKD(G)
2: **for** all vertex $v_i \in \{v_0, v_1, \ldots, v_{|V|-1}\}$ **parallel do**
3: $P_i \leftarrow \{v_{i+1}, v_{i+2}, \ldots, v_{|V|-1}\}$
4: $X_i \leftarrow \{v_0, v_1, \ldots, v_{i-1}\}$ // Let $\{v_0, v_1, \ldots, v_{i-1}\}$ be $\varnothing$ if $i = 0$
5: PBKP($\{v_i\}, P_i \cap N(v_i), X_i \cap N(v_i)$)
6: **end for**
7: **end procedure**

4.2 GPU-Based Bron-Kerbosch Algorithm

When processing problems involving irregular and complex data structures on a GPU, parallelization becomes more challenging. Finding efficient data partitioning for a task-parallel program is critical to improving the overall performance. In PBKP, the tasks among iterations of the *for* loop are independent such that multiple subgraphs can be searched simultaneously. However, considering the memory constraints of the GPU and the transfer latency between

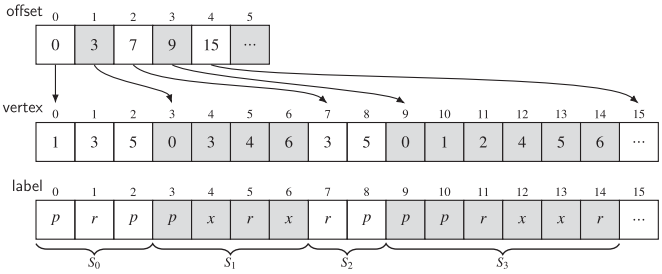

Fig. 8. The representation of subgraphs on GPU.

the CPU and GPU, we need to decompose the given graph to reduce the amount of data passed for recursive calls. In fact, for the recursive call invoked by the iteration of v_i, the useful information of the subgraph associated with v_i can be restricted to the induced subgraph of G, denoted by $G[\{v_i\} \cup N(v_i)]$, which is induced by the set $\{v_i\} \cup N(v_i)$. The graph decomposition of the induced subgraphs supports the parallel processing of the clique expansion on a GPU and provides higher data parallelism for GPU operations. Thus the utilization of GPU computing through load-balancing schemes is maximized.

Considering the limitation of memory spaces on a GPU, we need to design an efficient data structure to store the given graph and some of its subgraphs simultaneously during the process. For flexibility and efficiency of memory usage, we adopt two arrays, offset and vertex, represented as adjacency lists, as shown in Fig. 8. Array offset contains the starting index of the neighbors of the corresponding vertices in array vertex, and array vertex contains the neighboring vertices. For example, offset[0] is 0 and offset[1] is 3, then we have three neighbors of the first vertex, $\{1, 3, 5\}$, are stored in array vertex with the indices from 0 to 2. Beacuse $\text{offset}[2] - \text{offset}[1] = 4$, four neighbors of the second vertex are stored in array vertex with the indices from 3 to 6.

In the searching process, we store the vertices of subgraphs in the same way as the given graph is stored and use an array label to record the status of vertices in the subgraphs, as shown in Fig. 8. If v is in set R, the corresponding field of v is set as r. Similary, if the corresponding field of v is p or x, then $v \in P$ or $v \in X$. For instance, subgraph S_1 contains four vertices $\{0, 3, 4, 6\}$, stored in vertex with indice from 3 to 6, and corresponding labels $\{p, x, r, x\}$, stored in the label array with indice from 3 to 6. That is, the status of S_1 consists of $R = \{5\}$, $P = \{3\}$, and $X = \{4, 6\}$. In our GPU implementation, the given graph and all related subgraphs are expressed as array-based representations and arranged in lexicographical order of vertex IDs. Thus, the efficiency of the set operations can be improved by applying a binary search.

The description of the GPU-based Bron-Kerbosch algorithm (called GBK) is given in Algorithm 6, which is a GPU implementation of the parallel Bron-Kerbosch algorithm with degeneracy, as shown in Algorithm 5. The major difference is that the recursive functions are transformed as iteration operations because the recursion overhead on a GPU is very high. As proposed in [9], the recursion tree can be divided into several smaller subtrees based on the degeneracy ordering. Then, the algorithm can parallelly expand more subtrees simultaneously in the first level if there is still enough available space in memory. By Moon and Moser's theorem [18], we can estimate the maximum number of maximal cliques that could be found in a graph $G = (V, E)$ as $3^{|V|/3}$ where $|V|$ is the number of vertices in V. With the maximum number of maximal cliques and the size of the subgraph, we can decide the amount of memory usage for further expansion. To fully utilize the features of GPU architectures, we plan to parallelly expand as many cliques as possible if the amount of memory space required does not exceed the available memory provided by the GPU. Here, GBK uses the criteria to schedule as many tasks as possible to concurrently run on the GPU.

In Line 6 of the algorithm for GBK, MEM(C) computes the amount of memory space required for all subgraphs in set C. If the total maximum amount of memory space required by C and the induced subgraph $G[\{v_i\} \cup N(v_i)]$ are less than the capacity of memory, subgraph $G[\{v_i\} \cup N(v_i)]$ is added to C. When no more induced subgraphs can be inserted in C, we start to search all subgraphs in C and expand all cliques simultaneously by the GPU. Here, GBK is composed of three stages: EXAMINECLIQUE, SELECTPIVOT, and EXPANDCLIQUE. EXAMINECLIQUE examines and reports maximal cliques, SELECTPIVOT selects the pivot for each subgraph in C, and EXPANDCLIQUE expands the cliques and produces the induced subgraphs according to the selected pivots. At the end of the stage EXPANDCLIQUE, set C is updated to contain only the subgraphs to be further expanded. After the three stages are complete, the algorithm goes to the next iteration if C is not empty. The task of the *while* loop reloads more subgraphs to be checked if system memory is available. In all stages of GBK, balanced workload, coalesced memory access, and warp reduction are considered to achieve a high degree of parallelism on the GPU, including data parallelism and task parallelism.

Algorithm 6. GPU-Based Bron-Kerbosch Algorithm

Require: Graph $G = (V, E)$.
Ensure: Identify all maximal cliques in G.
1: **procedure** GBK(G)
2: Sort $V = \{v_0, v_1, \ldots, v_{|V|-1}\}$ in degeneracy order
3: $C \leftarrow \varnothing$ // the set of subgraphs
4: $i \leftarrow 1$
5: **repeat**
6: **while** $i \leq |V|$ **and** MEM(C) + MEM($G[\{v_i\} \cup N(v_i)]$) $<$ *Capacity* **do**
7: Insert $G[\{v_i\} \cup N(v_i)]$ to C
8: $i \leftarrow i + 1$
9: **end while**
10: EXAMINECLIQUE(C)
11: pivot $\leftarrow$ SELECTPIVOT(C)
12: $C \leftarrow$ EXPANDCLIQUE(C, pivot)
13: **until** $C = \varnothing$
14: **end procedure**

4.2.1 Maximal Clique Examination

In GBK, each subgraph is examined parallelly by EXAMINECLIQUE, where the vertices of a subgraph are examined in a batch by a warp. In a subgraph, a maximal clique is reported if $P \cup X = \varnothing$, which means all vertices in the subgraph are labeled r. As shown in Algorithm 7, threads of

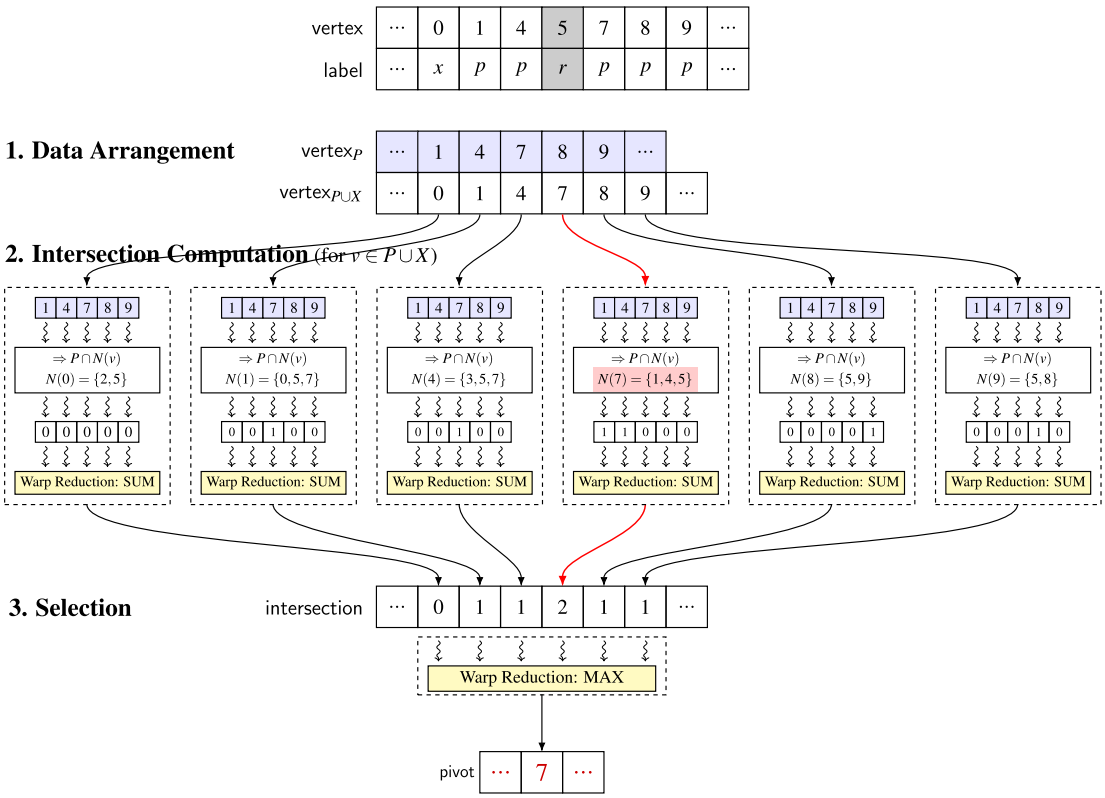

Fig. 9. Example for selecting a pivot for a subgraph on GPU.

the warp check whether any label is not equal to r. To lower memory usage and reduce memory latency, we use the *warp vote functions*, which allow the threads of a warp to perform a reduction-and-broadcast operation. The warp vote function evaluates a predicate for each thread and then broadcasts the result to all threads within a warp. After the statuses of all subgraphs involved are examined, prefix-sum operations are performed to compute the address for maximal cliques and count the number of maximal cliques found.

Algorithm 7. Kernel of Maximal Clique Examination

Require: Set C of subgraph.
Ensure: Report all maximal cliques in C.
1: **procedure** ExamineClique(C)
2: $i \leftarrow$ thread index
3: $n \leftarrow$ the size of the subgraph
4: label$_s \leftarrow$ the corresponding label array
5: **while** $i < n$ **parallel do**
6: **if** label$_s[i] \neq r$ **then**
7: all threads **return**
8: **end if**
9: $i \leftarrow i + 32$ // for the next wrap
10: **end while**
11: Mark the related subgraph as a maximal clique.
12: **end procedure**

4.2.2 Pivot Selection

The procedure SelectPivot(C) consists of three steps: data rearrangement, intersection computation, and selection. We first gather all vertices of the subgraphs marked as p to a new array P to make the global memory access coalesced. Then, we compute $P \cup X$ for each subgraph and store the results in a new array to avoid launching unnecessary thread blocks because vertices labeled r are not candidates for a pivot. Then, $P \cap N(v)$ for all vertices $v \in P \cup X$ are computed in parallel by a warp, where a binary search is exploited to check whether the vertices in both P and $N(v)$ to accelerate the intersection computations. After the set $P \cap N(v)$ has been obtained, the number of elements in the intersections is computed for each $v \in P \cup X$ by warp reduction using warp shuffle instructions. Then, a pivot is chosen from $P \cup X$ t that can maximize the size of the intersection of P and its neighbors, where warp reduction is also used for each subgraph.

Fig. 9 illustrates the pivot selection for subgraph $\{0, 1, 4, 5, 7, 8, 9\}$ by GBK. In the first step, the vertices labeled p and x are copied to the corresponding location for coalesced global memory accesses and reducing unnecessary blocks. We launch a warp for each subgraph to compute the location for each vertex using the prefix-sum operation, and vertices are copied to the corresponding locations in vertex$_P$ and vertex$_{P\cup X}$. Therefore, vertex$_P$ is $\{1, 4, 7, 8, 9\}$, and vertex$_{P\cup X}$ is $\{0, 1, 4, 7, 8, 9\}$. In the second step, we launch a warp for each vertex v in vertex$_{P\cup X}$ to compute the size of the intersection $P \cap N(v)$ and store it in the intersection array. In this example, six warps are launched and the threads of each warp use a binary search to determine whether each vertex v in vertex$_P$ is also in its neighborhood. Since threads of a warp are searching the same array space, the degree of warp divergence is very low. After all operations of the binary search are performed, the size of $P \cap N(v)$ is obtained by the prefix-sum operation

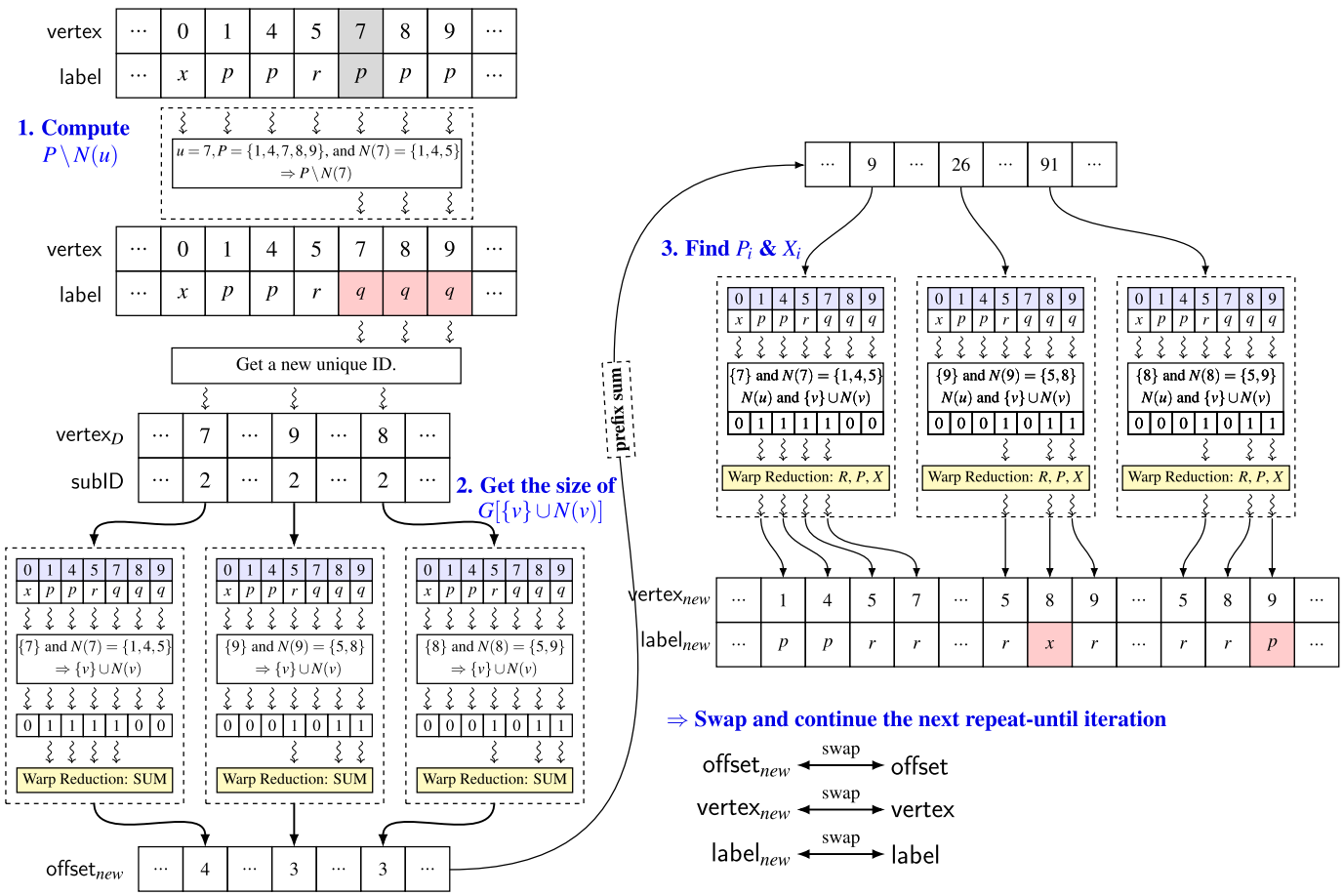

Fig. 10. Expanding maximal cliques on GPU.

using warp shuffle instructions, which greatly reduces the latency because these are processed at the register level. Then, each size of $P \cap N(v)$ is stored into the intersection array by a thread per warp. Finally, we launch a warp for each subgraph to select the vertex with the maximum size of intersections and store it to the pivot array. In this example, the maximum size of intersections is 2, and the corresponding vertex is 7. Thus, vertex 7 is selected as the pivot and stored in the pivot array for the next stage of clique expansions.

4.2.3 Clique Expansion

To expand the searching process for cliques on a GPU, the procedure ExpandClique(C, P) has three steps: computing subsets, computing sizes, and generating subgraphs. In the first step, we launch a warp for each subgraph to compute all vertices in $P \setminus N(u)$ and label them as q, where u is a pivot. These vertices are candidates for the following computation. Two arrays, vertex_D and subID, are used to store vertices in $P \setminus N(u)$ and the index of its subgraph. Since these two arrays store information for the candidates of all subgraphs in C, *atomicAdd*() is used to obtain the corresponding index for each candidate. In the second step, the size of the related subgraphs is computed by a binary search and warp reduction, similar to the pivot selection operations. Then, a prefix sum operation is activated to compute the new offsets for the new subgraphs. In the third step, the parameters used in the next iteration, such as $P_i \cap \{v_i\}$ and $X_i \cap \{v_i\}$, are computed by set operations based on a binary search and warp reduction. To mark a vertex having been processed, each vertex labeled q is labeled x if its vertex ID is smaller than that of the vertex added to the clique; otherwise, it is labeled p. Finally, we launch a warp for each new subgraph to write vertices and labels to the arrays vertex_new and label_new. This is implemented by swapping the corresponding pointers of the array instead of exchanging every pair of values in the corresponding array. Now the set C is updated and GBK starts the next iteration.

With the example used for pivot selection, we illustrate the expansion of subgraphs as shown in Fig. 10. First, vertices in $P \setminus N(u)$ are computed using a binary search and labeled as q. Then, each thread with a vertex in $P \setminus N(u)$ uses the atomic function to get a unique index in vertex_D and subID to store its vertex and corresponding subgraph ID. Thus, $P \setminus N(u) = \{7, 8, 9\}$ are obtained by a binary search and are labeled as q. Those three threads use the function $atomicAdd()$ to obtain their new IDs. Thus, the order of indices might not be the same as their thread IDs. In the second step, we launch a warp for each pair of vertex_D and subID to compute the size of newly induced subgraph and store it in the offset_new array. In the figure, we show only three pairs: (7,2), (9,2), and (8,2). The new subgraph is the subgraph induced by $\{v\} \cup N(v)$ for the current subgraph of v. Then, we use a binary search and warp shuffle instructions to obtain the size of the new subgraph. After

TABLE 1
Hardware Specifications

CPU	Intel Core i7-4770k 3.5 GHz
Main Memory	8 GB
GPU	NVIDIA GeForce GTX Titan X (Maxwell)
CUDA	7.5
Compute Capability	5.2
CUDA core	3072
Global Memory	12 GB DDR5X
Shared Memory	64 KB

computing the sizes of subgraphs, we perform a prefix-sum operation on the $\mathsf{offset}_{\mathsf{new}}$ array to obtain the offset of subgraphs in global memory. In the third step, we generate new subgraphs for further processing by applying a similar computation as in the second step. After the related parameters P_i, X_i are obtained, we store the information of the related vertices and labels to $\mathsf{vertex}_{\mathsf{new}}$ and $\mathsf{label}_{\mathsf{new}}$, respectively. The vertex labeled q is labeled p or x in the new subgraph. A vertex is labeled x if its vertex ID is smaller than that of the vertex added to the clique. Otherwise, it is labeled p. As seen in Fig. 10, the subgraphs formed by vertices 8 and 9 are the same, but the labels are different because of their vertex IDs. In the expansion of adding vertex 9, vertex 8 is labeled x. By contrast, vertex 9 is labeled p in the expansion of adding vertex 8. After gathering the unfinished subgraphs and updating the set C, we just need to switch the corresponding pointers of the arrays and go to the next repeat-until iteration.

5 Experimental Results

In this section, we compare the performance of GBK with that of the Bron-Kerbosch algorithm with degeneracy (BKD) [9] and a GPU-based algorithm GBKP. Because GPUs are suitable for massive data processing and the sequential BKD algorithm is very efficient for large-scale real-world graphs, we take the performance of BKD as a reference for comparison. On the other hand, GBKP is a GPU version of PBKP and PBKP is the parallel version of a sequential MCE solution with worst-case time complexity [23]. According to Algorithms 4 and 5, GBKP can be devised by the same idea of GBK and using the pivot selected in the first-level expansion. Moreover, to investigate the effect of degeneracy ordering, we discuss two different vertex orderings based on degree, the increasing and decreasing order, and study the impact of orderings on the performance of GBK.

In the experimental environment, we used C++ 11 and CUDA programming languages for code development in the operating system Debian 8. Table 1 lists the hardware specifications. For the implementation of BKD, we used the well-optimized source code provided in [9]. For GPU implementations, we used the same optimization techniques to fully utilize the GPU.

The experimental data consist of five publicly available datasets from real-world networks: SNAP [13], KONECT [1], Mark Newman, Pajek, and DIMACS [9]. We also tested a synthesized dataset and random graphs used in [9] to evaluate the performance on graphs with different sizes and densities. Tables 2, 3, 4, 5, and 6 show the experimental results. For each graph, we ran each algorithm 50 times and calculated the average execution time in seconds. We list the number of vertices $|V|$, number of edges $|E|$, degeneracy d, and number of maximal cliques c. To aid analysis, the graphs are listed in order by the number of maximal cliques. We further evaluated and compared the performance of three algorithms on social graphs and synthesis graphs. Moreover, we analyzed each stage of GBK and the efficiency of implementing GBK on a GPU.

5.1 Social Graphs

A social graph represents of a social network that depicts personal relations, and it is usually sparse. We evaluate the performance on social graphs, SNAP, KONECT, Mark Newman, and Pajek datasets, as shown in Tables 2, 3, and 4. GBK and GBKP take advantage of parallel computing on the GPU when the number of maximal cliques is large, while BKD outperforms when the number of maximal cliques is relatively small. In general, GBK and GBKP are faster than BKD for cases where the number of maximal cliques is greater than 500,000, except in some instances, such as web-NotreDame in Table 2 and Linux-reply in Table 3. GBK is even faster than GBKP for the first four large graphs in Tables 2 and 3, because GBK can divide the graph into smaller subgraphs by degeneracy ordering and explore them parallelly in the first level. On the other hand, as shown in Table 4, BKD was the fastest on all graphs in Mark Newman's dataset because the overhead for allocating memory space needed on the GPU takes about 200 milliseconds in these cases.

5.2 Synthesized Graphs

Compared to the class of social graphs, we synthetically generated two classes of graphs—dense graphs and random graphs—to study the algorithms' performance behaviors for these classes.

Dense Graphs.For this class of graphs, we evaluated the performance with two synthesis benchmarks: the DIMACS and Moon-Moser datasets. DIMACS is a collection of algorithmically generated graphs that are intended as difficult examples for MCE. Moon-Moser's dataset contains graphs generated according to Moon and Moser's theorem, which proved that any graph with n vertices has at most $3^{n/3}$ maximal cliques. Although BKD was proposed to improve the performance of the original Bron-Kerbosch algorithm on large sparse real-world graphs and GBK is a GPU implementation of BKD, the performance of these algorithms does not show significant improvements for dense graphs. However, GBK successfully accelerates BKD on dense graphs by parallelly searching for graphs with relatively large amounts of maximal cliques. As shown in Table 5, GBK reduces the total execution time of BKD by 50 to 75 percent when the number of maximal cliques is large. However, GBKP is even faster than GBK for these cases because most of the expanding tasks complete in the first few

TABLE 2
Comparison of Runtimes (in sec.) for Stanford Large Network Dataset Collection (SNAP)

Graph	$\|V\|$	$\|E\|$	d	c	BKD	GBKP	GBK
wiki-Talk	2,394,385	4,659,565	131	86,333,306	83.976	45.837	**41.092**
wiki-topcats	1,791,489	25,444,207	99	27,229,873	58.882	35.988	**26.577**
cit-Patents	3,774,768	16,518,947	64	14,787,032	30.109	8.1	**7.086**
sx-mathoverflow	24,759	187,986	78	3,867,588	3.727	2.801	**2.26**
web-Berkstan	685,231	6,649,470	201	3,405,813	7.688	17.949	**5.73**
roadNet-CA	1,965,206	2,766,607	3	2,537,996	1.74	**0.681**	1.698
soc-Epinions1	75,888	405,740	67	1,775,074	1.847	**1.231**	1.255
roadNet-TX	1,379,917	1,921,660	3	1,763,318	1.198	**0.578**	1.303
web-Google	875,713	4,322,051	44	1,417,580	4.097	2.249	**1.696**
roadNet-PA	1,088,092	1,541,898	3	1,413,391	0.973	**0.529**	1.069
sx-superuser	192,409	714,570	61	1,259,611	1.725	**1.104**	1.206
loc-Gowalla	196,591	950,327	51	1,212,679	1.704	1.1	**0.986**
web-Stanford	281,904	1,992,636	71	1,055,937	3.53	3.251	**1.671**
Amazon0505	410,236	2,439,437	10	1,034,133	2.928	1.826	**1.088**
Amazon0601	403,394	2,443,408	10	1,023,572	2.938	1.817	**1.084**
Amazon0312	400,727	2,349,869	10	1,007,757	2.856	1.792	**1.062**
Slashdot0902	82,168	504,230	55	890,041	0.978	1.049	**0.829**
Slashdot090221	82,144	500,481	54	854,413	0.953	**0.663**	0.789
Slashdot090216	81,871	497,672	54	847,241	0.943	**0.667**	0.77
Slashdot081106	77,357	468,554	54	823,845	0.888	**0.636**	0.738
Slashdot0811	77,360	469,180	54	823,415	0.876	**0.662**	0.741
com-Amazon	548,552	925,872	6	683,731	1.472	**0.721**	0.767
sx-askubuntu	157,222	455,691	48	547,924	0.745	**0.565**	0.686
web-NotreDame	325,730	1,090,108	155	495,948	**0.717**	1.114	1.021
Cit-HepTh	27,769	352,285	37	464,872	0.749	0.501	**0.472**
wiki-Vote	7,115	100,762	53	459,002	0.456	**0.402**	0.615
Cit-HepPh	34,546	420,877	30	412,493	0.669	0.502	**0.426**
Amazon0302	262,111	899,792	6	403,360	0.868	**0.536**	0.56
email-EuAll	265,214	364,481	37	377,956	**0.415**	0.594	0.455
loc-Brightkite	58,228	214,078	52	290,004	**0.353**	0.518	0.441
com-DBLP	317,081	1,049,866	113	257,552	0.969	1.084	**0.734**
email-Enrson	36,692	183,831	43	226,859	**0.344**	0.447	0.442
p2p-Gnutella31	62,586	147,892	6	144,482	**0.097**	0.282	0.326
p2p-Gnutella30	36,682	88,328	7	85,826	**0.048**	0.263	0.306
p2p-Gnutella24	26,518	65,369	5	63,727	**0.036**	0.26	0.298
p2p-Gnutella25	22,687	54,705	5	53,371	**0.028**	0.246	0.286
as-caida20071105	26,475	53,381	22	43,949	**0.036**	0.245	0.269
email-Eu-core	1,005	16,064	34	42,728	**0.047**	0.263	0.293
p2p-Gnutella04	10,879	39,994	7	38,500	**0.018**	0.252	0.285
CA-AstroPh	18,771	198,050	56	36,427	**0.235**	0.362	0.356
p2p-Gnutella05	8,846	31,839	9	30,581	**0.014**	0.25	0.285
p2p-Gnutella06	8,717	31,525	9	30,362	**0.014**	0.25	0.287
p2p-Gnutella09	8,114	26,013	10	24,480	**0.012**	0.259	0.288
p2p-Gnutella08	6,301	20,777	10	19,416	**0.009**	0.256	0.287
CA-CondMat	23,133	93,439	25	18,502	**0.072**	0.26	0.273
CA-HepPh	12,006	118,489	238	14,937	**0.141**	0.336	0.553
CA-HepTh	9,875	25,973	31	9,940	**0.015**	0.243	0.274
as20000102	6,474	12,572	12	9,220	**0.006**	0.234	0.255
CA-GrQc	5,241	14,484	43	3,905	**0.008**	0.242	0.271

subgraphs, and GBK takes extra time to reload and reschedule the subgraphs generated in each iteration.

*Random Graphs.*We also evaluated performance on random graphs with varying size and sparsity, as shown in Table 6. For these datasets, GBK and GBKP are faster than BKD when the number of maximal cliques is greater than 500,000, and GBKP is more efficient than GBK for most of these cases.

5.3 Percentage of Execution Time for Stages

We chose four small real-world graphs (Infectious, CA-GrQcv, Human-protein, and as20000102) and four large real-world graphs (Wiki-it, Wiki-nl, Wiki-Talk, and as-Skitter) to analyze the execution time of each stage in GBK, as shown in Table 7. Preparation, including allocating memory space and transferring vertices and edges of the graph, takes around 90 percent of the total execution time of GBK for the small graphs. However, for the large graphs, degeneracy sorting takes slightly more time than preparation because it cannot be executed parallelly and is therefore computed by the CPU. Both initialization and examination require less than 12 percent of the total execution time for these large graphs. Meanwhile, pivot selection and clique expansion are the most time-consuming parts of GBK, taking 85 to 90

TABLE 3
Comparison of Runtimes (in sec.) for Koblenz Network Collection (KONECT)

Graph	$\|V\|$	$\|E\|$	d	c	BKD	GBKP	GBK
wiki-it	1,204,010	20,016,908	271	148,294,130	304.886	89.902	**78.5**
wiki-nl	1,039,253	12,360,721	166	47,585,971	82.449	33.388	**30.217**
as-skitter	1,696,415	11,095,298	111	37,322,355	57.179	60.802	**40.879**
soc-Pokec	1,632,804	22,301,964	47	19,376,874	54.257	31.344	**12.854**
wiki-conflict	118,101	2,027,871	145	17,135,233	22.83	**9.867**	10.269
com-Youtube	3,223,590	9,375,374	88	15,624,590	25.573	11.225	**8.352**
Hudong	1,984,485	14,428,382	266	12,784,007	34.106	17.376	**13.438**
Flixster	2,523,387	7,918,801	68	10,729,184	15.099	**6.402**	6.72
Linux-reply	63,400	159,996	91	5,481,951	**6.229**	7.072	7.321
LiveMocha	104,104	2,193,083	92	3,711,570	5.97	**2.977**	4.678
prosper-loans	89,270	3,330,022	111	3,385,942	6.035	**1.701**	2.962
youtube-links	1,138,500	2,990,287	51	3,277,236	4.745	2.614	**2.198**
Youtube-friendship	1,134,891	2,987,624	51	3,265,957	4.708	**1.653**	2.192
TREC-WT10g	1,601,788	6,679,248	140	2,699,631	4.62	4.74	**2.765**
hyves	1,402,674	2,777,419	39	2,590,218	3.153	2.765	**1.234**
Facebook-friendship	63,732	817,035	52	1,539,039	2.427	1.837	**1.474**
wiki-signed	138,593	715,883	55	1,329,833	1.906	**1.123**	1.216
wiki-simple	100,313	824,968	90	1,273,462	1.882	**0.853**	0.854
citeseer	384,414	1,736,145	15	1,233,311	2.482	1.985	**0.817**
dblp-coauthor	1,282,469	5,179,996	118	1,219,321	7.057	4.958	**2.314**
twitter	465,018	833,540	30	805,171	0.611	**0.431**	0.522
slashdot-zoo	79,121	467,731	54	783,681	0.873	0.907	**0.661**
Internet-topology	34,762	107,720	63	581,514	0.725	**0.523**	0.541
enron	87,274	297,456	53	465,304	0.663	**0.533**	0.665
wiki-election	7,119	100,693	53	457,995	0.456	**0.35**	0.548
eat	23,133	297,094	34	291,504	0.377	0.432	**0.288**
wordnet-words	146,006	328,500	14	243,070	**0.317**	0.468	0.363
linux	30,838	213,217	23	187,126	**0.155**	0.323	0.274
Facebook-wallposts	46,953	183,412	16	132,206	**0.156**	0.284	0.258
slashdot-threads	51,084	116,573	14	106,105	**0.085**	0.257	0.267
JDK-dependency	6,435	53,658	65	33,646	**0.047**	0.231	0.288
javax-dependency	6,121	50,290	65	31,891	**0.045**	0.236	0.288
openflights	3,426	19,256	31	22,781	**0.029**	0.224	0.249
opsahl-openfights	2,940	15,677	28	16,169	**0.02**	0.219	0.244
Human-protein	2,240	6,432	10	5,808	**0.003**	0.205	0.224
Infectious	411	2,765	17	1,248	**0.002**	0.204	0.229

TABLE 4
Runtimes for Mark Newman and Pajek Datsets

Graph	$\|V\|$	$\|E\|$	d	c	BKD	GBKP	GBK
polblogs	1,490	16,715	36	49,884	**0.049**	0.294	0.361
internet	22,963	48,436	25	39,288	**0.032**	0.279	0.338
cond-mat	40,421	175,693	29	34,274	**0.139**	0.347	0.377
astro-ph	16,706	121,251	56	15,794	**0.09**	0.338	0.374
power	4,941	6,594	5	5,687	**0.003**	0.278	0.35
celegansneural	297	2,148	10	1,386	**0.001**	0.269	0.33
netscience	1,589	2,742	19	741	**0.001**	0.276	0.344
adjnoun	112	425	6	303	**0.0**	0.26	0.319
football	115	613	8	281	**0.0**	0.264	0.326
polbooks	105	441	6	199	**0.0**	0.27	0.336
dolphins	62	159	4	84	**0.0**	0.265	0.326
lesmis	77	254	9	59	**0.0**	0.269	0.333
karate	34	78	4	36	**0.0**	0.257	0.315
daysall	13,308	148,035	73	2,173,772	2.0	**1.376**	1.671
NDwww	325,729	1,090,108	155	495,947	**0.721**	0.776	0.979
patents	240,547	560,943	24	482,538	0.639	**0.543**	0.611
hep-th	27,240	341,923	37	446,823	0.717	**0.532**	0.545
EAT-RS	23,219	304,937	34	298,164	0.431	**0.427**	**0.427**
foldoc	13,356	91,471	12	39,590	**0.055**	0.303	0.363

TABLE 5
Runtimes for DIMACS and Moon-Moser Datasets

Graph	$\|V\|$	$\|E\|$	d	c	BKD	GBKP	GBK
keller4	171	9,435	102	10,284,321	4.138	**1.347**	1.992
johnson16-2-4	120	5,460	91	2,027,025	4.295	**1.685**	1.918
hamming6-2	64	1,824	57	1,281,402	0.817	**0.78**	0.966
MANN-a9	45	918	40	590,887	**0.152**	0.449	0.552
brock200-2	200	9,876	84	431,586	**0.493**	0.517	0.511
johnson8-4-4	70	1,855	53	114,690	**0.093**	0.334	0.413
p-hat300-1	300	10,933	49	58,176	**0.055**	0.331	0.439
hamming6-4	64	704	22	464	**0.0**	0.309	0.382
c-fat500-10	500	46,627	185	8	**0.048**	0.391	0.722
c-fat200-5	200	8,473	83	7	**0.005**	0.326	0.423
m-m-48	48	1,080	45	43,723,813	5.905	**3.104**	3.217
m-m-45	45	945	42	14,348,907	1.96	**1.386**	1.611
m-m-30	30	405	27	59,049	**0.009**	0.317	0.395

TABLE 6
Runtimes for Random Graphs

Graph	$\|V\|$	$\|E\|$	d	c	BKD	GBKP	GBK
1000-0.3	1,000	149,998	266	15,671,489	28.141	4.727	**4.528**
100-0.8	100	3,978	70	5,776,276	4.505	**2.24**	3.583
300-0.5	300	22,328	130	4,151,668	5.518	**1.528**	1.585
10000-0.03	10,000	1,499,357	262	3,733,699	8.484	**1.15**	2.959
700-0.3	700	73,563	184	3,107,208	4.809	**1.07**	1.173
3000-0.1	3,000	448,782	263	2,886,628	5.728	**1.262**	1.465
1000-0.2	1,000	99,850	172	1,190,899	1.785	**0.551**	0.622
2000-0.1	2,000	199,817	170	750,991	1.307	**0.67**	0.825
500-0.3	500	37,331	127	701,292	0.946	**0.504**	0.607
300-0.4	300	18,043	101	555,724	0.693	**0.439**	0.544
100-0.7	100	3,484	59	439,928	**0.386**	0.407	0.489
10000-0.01	10,000	499,910	80	349,244	0.7	**0.663**	0.781
700-0.2	700	48,704	117	321,245	0.4	**0.374**	0.461
10000-0.005	10,000	249,982	38	215,477	**0.224**	0.627	0.416
10000-0.003	10,000	150,336	21	141,865	**0.103**	0.383	0.375
1000-0.1	1,000	49,963	82	99,561	**0.119**	0.38	0.405
500-0.2	500	24,844	81	98,875	**0.109**	0.326	0.463
300-0.3	300	13,371	74	86,179	**0.095**	0.318	0.428
100-0.6	100	2,977	51	59,898	**0.058**	0.301	0.377
10000-0.001	10,000	50,077	7	49,716	**0.024**	0.288	0.362
700-0.1	700	24,642	56	38,139	**0.037**	0.325	0.449
300-0.2	300	9,073	47	18,911	**0.018**	0.311	0.394
500-0.1	500	12,568	39	15,311	**0.013**	0.314	0.383
300-0.1	300	4,396	21	3,663	**0.003**	0.297	0.371

TABLE 7
Percentage of Runtime by Stages

Graph	Preparation	Sort	Initialize	Examine	Select	Expand
Infectious	94.92%	1.16%	0.07%	0.88%	1.84%	1.15%
CA-GrQcv	86.26%	2.35%	0.34%	2.19%	5.49%	3.36%
Human-protein	96.86%	1.55%	0.09%	0.35%	0.69%	0.46%
as20000102	93.90%	2.46%	0.14%	0.63%	1.62%	1.25%
wiki-it	0.61%	4.62%	3.76%	4.66%	56.71%	29.64%
wiki-nl	1.14%	6.89%	5.66%	6.15%	51.62%	28.10%
wiki-Talk	0.35%	7.33%	0.39%	2.22%	70.36%	24.25%
as-Skitter	0.55%	4.46%	2.08%	2.99%	68.08%	21.83%

TABLE 8
Percentage of Runtime of Selection and Expansion

Graph	SELECT-1	SELECT-2	SELECT-3	EXPAND-1	EXPAND-2	EXPAND-3
wiki-it	7.35%	47.00%	2.37%	6.05%	11.65%	11.93%
wiki-nl	8.29%	40.80%	2.53%	5.99%	10.96%	11.15%
wiki-Talk	4.52%	63.59%	2.24%	6.45%	8.88%	8.92%
as-Skitter	5.46%	60.80%	1.83%	5.38%	8.17%	8.29%

TABLE 9
The Efficiency of Multiprocessors of Each Kernel on Infectious, Wiki-Talk, and as-Skitter

Stages	Infectious		Wiki-Talk		as-Skitter	
	Max.	Avg.	Max.	Avg.	Max.	Avg.
INITIALIZE	40.59%	40.59%	**99.96%**	8.16%	**99.93%**	18.08%
EXAMINE	42.97%	24.17%	**99.95%**	65.75%	**99.96%**	56.12%
SELECT-1	44.38%	26.06%	**99.94%**	63.84%	**99.95%**	57.60%
SELECT-2	**91.72%**	59.60%	**99.99%**	**91.59%**	**100.00%**	**89.71%**
SELECT-3	47.45%	30.48%	**99.94%**	67.07%	**99.95%**	61.14%
EXPAND-1	67.77%	39.66%	**99.97%**	79.74%	**99.97%**	76.32%
EXPAND-2	43.81%	25.61%	**99.98%**	79.71%	**99.99%**	78.25%
EXPAND-3	65.26%	39.82%	**99.98%**	80.32%	**99.98%**	78.55%

percent of the total execution time. For further analysis, we divided the pivot selection and clique expansion into three steps, as shown in Table 8, where each step corresponds to the steps in Figs. 9 and 10. The most time-consuming step is SELECT-2 because we launched a block for each vertex labeled P or X in all subgraphs in C. EXPAND-2 and EXPAND-3 take about the same time because the main computation is a binary search performed by a block for each subgraph. Note that SELECT-1 takes a relatively short time to rearrange vertex_P and $\mathsf{vertex}_{P\cup X}$, making the global memory accesses coalesced for the binary search in the next step.

5.4 Efficiency

To investigate the usage of computing resources on the GPU for GBK, we exploited the profiler provided by NVIDIA to investigate the efficiency of multiprocessors, as measured by the percentage of time at least one warp is active on a multiprocessor. Table 9 shows the maximum and average efficiency of multiprocessors of each kernel in GBK. For the Infectious graph, although the maximum efficiency of multiprocessors of SELECT-2 in GBK was 91.72 percent, the average efficiency of multiprocessors of kernels in GBK is lower than 60 percent because Infectious is a small graph. Thus, we also measured the efficiency of multiprocessors of each kernel in GBK on two large graphs: wiki-Talk and as-Skitter. As shown in Table 9, the maximum efficiency of the multiprocessors of all kernels is close to 100 percent, which means the GPU was fully utilized. The average efficiency of the multiprocessors for the most time-consuming step, SELECT-2, is 90 percent in both large graphs. For other kernels, the average efficiency of multiprocessors is between 60 and 80 percent, except for INITIALIZE. In the large graphs, the degeneracy of most vertices is high, meaning that more maximal cliques are generated. Hence, a relatively small number of vertices is added to the set of subgraphs in each iteration. As a result, the average efficiency of multiprocessors of INITIALIZE is quite low.

5.5 Expansion Order in the First Level

In Table 7, GBK takes about 5 percent of the execution time to arrange vertices in degeneracy order for large graphs. The procedure of the degeneracy order is calculated sequentially since the subgraphs involved are dependent and not easy to be parallelized. Thus, we consider exploring graphs in the first level in the increasing order of degree because GBK tends to process vertices of smaller degree first and the increasing order of degree can be calculated in parallel on GPU. Table 10 shows the runtimes of four GPU-based implements of MCE. GBK$^+$ and GBK* are two adaptions from GBK with the first-level expansion based on the increasing and decreasing degree order, respectively. Since GBKP tends to process vertices of larger degree first, we take GBK* as a reference. The symbol NA indicates that the algorithm was unable to run on that problem instance due to GPU's memory limitations. For these large graphs, GBK$^+$ outperforms the other algorithms for most cases by further reducing the sorting overhead.

TABLE 10
Runtimes of GPU-Based Algorithms for SNAP Dataset

Graph of SNAP	GBK	GBK$^+$	GBK*	GBKP
wiki-Talk	**41.092**	47.375	NA	45.837
wiki-topcats	26.577	**16.712**	43.329	35.988
cit-Patents	7.086	**2.364**	6.821	8.1
sx-mathoverflow	**2.26**	2.935	2.805	2.801
web-Berkstan	5.73	**4.423**	19.354	17.949
roadNet-CA	1.698	**0.646**	0.648	0.681
soc-Epinions1	1.255	**1.199**	1.47	1.231
roadNet-TX	1.303	**0.546**	0.549	0.578
web-Google	1.696	**0.941**	2.203	2.249
roadNet-PA	1.069	**0.496**	**0.496**	0.529

6 Conclusion

The objectives of this study were to accelerate MCE computations aided by a GPU and to achieve the most balanced GPU workloads. Because optimization of MCE computations through sequential algorithms is approaching the ceiling of this kind of problem, researchers have recently begun to look at parallel methods. In addition, recent advances in GPU architectures have allowed researchers to design various algorithms to achieve the goal of faster computations. We proposed GBK, a faster MCE algorithm that effectively utilizes GPU architectures. Using the Moon-Moser bound, we searched and expanded the cliques on a GPU to as many as possible in each iteration to reduce the number of kernels launched and the number of data transfers between host and device. Further, we used binary searching to efficiently conduct set manipulations in parallel. By rearranging vertices, we minimized the random access of the memory deriving from binary searches. Furthermore, warp reduction reduces memory latency in most GPU kernels. The experimental results showed that the proposed algorithm successfully accelerates MCE in both sparse and dense graphs. We further analyzed the percentage of time and the efficiency of streaming multiprocessors in each phase, demonstrating that GBK is efficient.

Acknowledgments

This work was supported in part by the Ministry of Science Technology under the Grant MOST 107-2221-E-011-015-MY2.

References

[1] The Koblenz network collection – KONECT, 2016. [Online]. Available: http://konect.uni-koblenz.de/networks
[2] E. A. Akkoyunlu, "The enumeration of maximal cliques of large graphs," *SIAM J. Comput.*, vol. 2, no. 1, pp. 1–6, 1973.
[3] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.
[4] F. Cazals and C. Karande, "A note on the problem of reporting maximal cliques," *Theor. Comput. Sci.*, vol. 407, no. 1–3, pp. 564–568, 2008.
[5] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.
[6] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura, "Shared-memory parallel maximal clique enumeration from static and dynamic graphs," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, pp. 1–28, 2020.
[7] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin, "Parallel algorithm for enumerating maximal cliques in complex network," in *Mining Complex Data*. Berlin, Germany: Springer, 2009, pp. 207–221.
[8] D. Eppstein, "All maximal independent sets and dynamic dominance for sparse graphs," *ACM Trans. Algorithms*, vol. 5, no. 4, 2009, Art. no. 38.
[9] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," *J. Exp. Algorithmics*, vol. 18, 2013, Art. no. 3.1.
[10] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the GPU," in *Proc. Eur. Conf. Parallel Process.*, 2011, pp. 425–437.
[11] H. Johnston, "Cliques of a graph-variations on the Bron-Kerbosch algorithm," *Int. J. Parallel Program.*, vol. 5, no. 3, pp. 209–238, 1976.
[12] E. L. Lawler, J. K. Lenstra, and A. Rinnooy Kan, "Generating all maximal independent sets: NP-hardness and polynomial-time algorithms," *SIAM J. Comput.*, vol. 9, no. 3, pp. 558–565, 1980.
[13] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," 2014. [Online]. Available: http://snap.stanford.edu/data
[14] H. Li, K. Li, J. An, and K. Li, "MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1530–1544, Jul. 2018.
[15] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.
[16] E. Loukakis, "A new backtracking algorithm for generating the family of maximal independent sets of a graph," *Comput. Math. Appl.*, vol. 9, no. 4, pp. 583–589, 1983.
[17] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *Proc. Scand. Workshop Algorithm Theory*, 2004, pp. 260–272.
[18] J. W. Moon and L. Moser, "On cliques in graphs," *Israel J. Math.*, vol. 3, no. 1, pp. 23–28, 1965.
[19] K. A. Naudé, "Refined pivot selection for maximal clique enumeration in graphs," *Theor. Comput. Sci.*, vol. 613, pp. 28–37, 2016.
[20] P. San Segundo, J. Artieda, and D. Strash, "Efficiently enumerating all maximal cliques with bit-parallelism," *Comput. Operations Res.*, vol. 92, pp. 37–46, 2018.
[21] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park, "A scalable, parallel algorithm for maximal clique enumeration," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 417–428, 2009.
[22] V. Stix, "Finding all maximal cliques in dynamic graphs," *Comput. Optim. Appl.*, vol. 27, no. 2, pp. 173–186, 2004.
[23] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comput. Sci.*, vol. 363, no. 1, pp. 28–42, 2006.
[24] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM J. Comput.*, vol. 6, no. 3, pp. 505–517, 1977.
[25] B. Wu, S. Yang, H. Zhao, and B. Wang, "A distributed algorithm to enumerate all maximal cliques in MapReduce," in *Proc. 4th Int. Conf. Front. Comput. Sci. Technol.*, 2009, pp. 45–51.
[26] Y. Xu, J. Cheng, and A. W.-C. Fu, "Distributed maximal clique computation and management," *IEEE Trans. Services Comput.*, vol. 9, no. 1, pp. 110–122, Jan./Feb. 2016.
[27] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima, "Relational joins on GPUs: A closer look," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2663–2673, Sep. 2017.
[28] T. Yu and M. Liu, "A memory efficient maximal clique enumeration method for sparse graphs with a parallel implementation," *Parallel Comput.*, vol. 87, pp. 46–59, 2019.

Yi-Wen Wei received the BS and PhD degrees in computer and electronic engineering, both from the National Taiwan University of Science and Technology, Taipei, Taiwan, in 2013 and 2018, respectively. His research interests includes multi-dimensional data structures, algorithms, and general-purpose computing on graphic processing.

Wei-Mei Chen received the PhD degree in computer science and information engineering from the National Taiwan University, Taipei, Taiwan, in 2000. She is currently a professor of electronic and computer engineering with the National Taiwan University of Science and Technology, where she initially joined in August 2005. Her research interests include algorithm design and analysis, data mining, and parallel computing.

Hsin-Hung Tsai received the BS degree from the National Taipei University of Technology, Taipei, Taiwan, in 2019. He is currently working toward the master's degree at the National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include parallel computing and data mining.