Parallelizing Maximal Clique Enumeration on GPUs

Mohammad Almasri*†, Yen-Hsiang Chang*†, Izzat El Hajj‡, Rakesh Nagi§, Jinjun Xiong¶, Wen-mei Hwu†‖
*Both authors contributed equally to this research.
†ECE,§ISE, University of Illinois at Urbana-Champaign, Urbana, IL, USA
‡Department of Computer Science, American University of Beirut, Beirut, Lebanon
¶Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY, USA
‖Nvidia Corporation, Santa Clara, CA, USA
almasri3@illinois.edu, yhchang3@illinois.edu, izzat.elhajj@aub.edu.lb, nagi@illinois.edu,
jinjun@buffalo.edu, w-hwu@illinois.edu

***Abstract*—We present a GPU solution for exact maximal clique enumeration (MCE) that performs a search tree traversal following the Bron-Kerbosch algorithm. Prior works on parallelizing MCE on GPUs perform a breadth-first traversal of the tree, which has limited scalability because of the explosion in the number of tree nodes at deep levels. We propose to parallelize MCE on GPUs by performing depth-first traversal of independent subtrees in parallel. Since MCE suffers from high load imbalance and memory capacity requirements, we propose a worker list for dynamic load balancing, as well as partial induced subgraphs and a compact representation of excluded vertex sets to regulate memory consumption. Our evaluation shows that our GPU implementation on a single GPU outperforms the state-of-the-art parallel CPU implementation by a geometric mean of 4.9× (up to 16.7×), and scales efficiently to multiple GPUs. Our code has been open-sourced to enable further research on accelerating MCE.**

I. Introduction

A clique in a graph is a complete subgraph where every vertex in the subgraph is adjacent to every other vertex. A maximal clique is a clique that cannot be further expanded by including one more vertex. Maximal clique enumeration (MCE) aims to find all the maximal cliques in a graph, which has a wide variety of applications in numerous domains such as community detection [1]–[4], recommender systems [5], [6], graph compression and partitioning [7]–[10], prediction of protein functions in protein interaction networks [11]–[14], finding gene similarities in gene co-expression networks [15], [16], and identifying price fluctuations in finance networks [17].

One of the most widely used algorithms for solving MCE exactly is the Bron-Kerbosch algorithm [18]. The algorithm involves traversing a search tree that branches from parent nodes representing smaller cliques to child nodes representing larger cliques that contain them until maximal cliques are found. Prior works on parallelizing MCE on GPUs perform a breadth-first traversal of the search tree [19]–[22]. However, this approach does not scale well for large graphs because of the explosion in the number of search tree nodes that need to be tracked at deep levels of the tree. To overcome this limitation, we propose to parallelize MCE on GPUs by assigning independent subtrees to different thread blocks and having the threads within each block collaboratively perform a depth-first traversal of the block's subtree.

The approach of performing per-block depth-first traversal of independent subtrees has been applied in our prior work on k-clique counting [23]. However, MCE presents two key scalability challenges that are less of a concern in k-clique counting. The first challenge is that the MCE search tree is substantially more imbalanced, which means that assigning independent subtrees to different thread blocks suffers from high load imbalance. The second challenge is that MCE requires substantially more memory capacity to track the vertices excluded at each level of the traversal to test for maximality of a clique. These two challenges are particularly critical on GPUs in contrast with CPUs. GPUs are more sensitive to load imbalance than CPUs due to their massively parallel nature [24]. Moreover, GPUs typically have a smaller memory capacity than CPUs while putting more pressure on the memory capacity by traversing a larger number of subtrees in parallel.

In this paper, we propose a novel solution for accelerating MCE on GPUs that employs various techniques to address the load imbalance and memory capacity challenges that MCE imposes. We propose a worker list to enable thread blocks with large subtrees to offload branches of their subtrees to other thread blocks with low overhead. We propose using partial induced subgraphs to avoid the latency and memory capacity overhead of constructing full induced subgraphs. We propose a compact representation of the sets of excluded vertices that distinguishes between the part of each set that needs to be stored separately for each level, and the part that monotonically shrinks and can be reused across levels. We also retain several optimizations used in our prior work on k-clique counting [23], such as binary encoding of the induced subgraph and partitioning work at subwarp granularity.

Our evaluation shows that our parallel GPU implementation executing on a single server-grade GPU outperforms the state-of-the-art parallel CPU implementation [25] executing on a server-grade CPU by a geometric mean of 4.9× (up to 16.7×). We also show that our worker list approach is effective at achieving load balance with low overhead, and enables efficient scaling to multiple GPUs. Our code has been open-sourced for reproducibility and to enable further research on accelerating MCE.

2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT) | 979-8-3503-4254-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/PACT58117.2023.00022

979-8-3503-4254-3/23/$31.00 ©2023 IEEE
DOI 10.1109/PACT58117.2023.00022

II. Background

A. Maximal Clique Enumeration

Let $G = (V, E)$ be a simple undirected graph where V is the set of vertices in G and E is the set of edges in G. The *neighborhood* of a vertex $v \in V$ is the set of vertices adjacent to v, denoted by $N(v)$. A *clique* in G is a complete subgraph of G where every vertex in the subgraph is adjacent to every other vertex in the subgraph. A *maximal clique* in G is a clique that cannot be further expanded by including one more vertex. In other words, a maximal clique is a clique that is not contained in a larger clique. For example, the graph in Fig. 1(a) has two maximal cliques: ABCD and AEF. On the other hand, ABC, ABD, ACD, and BCD are not maximal cliques because they are all contained in ABCD.

MCE aims to find all the maximal cliques in a graph. We tackle MCE as an exact problem, which means that we enumerate all maximal cliques in the graph and do not apply any approximations or graph sampling. While approximate maximal cliques may be sufficient for some applications, other applications require exact maximal cliques. For example, in protein-protein interaction networks, a protein complex is necessarily a clique [26]. In addition, even for applications where approximate solutions can be used, there is added value in using an exact solution if finding it can be made sufficiently efficient.

One of the most widely used algorithms for exact MCE is the Bron-Kerbosch algorithm [18]. We describe different variants and optimizations of this algorithm in the rest of this section.

B. Bron-Kerbosch

The Bron-Kerbosch algorithm [18] is a backtracking algorithm that traverses a search tree to find maximal cliques. The search tree branches from parent nodes representing smaller cliques to child nodes representing larger cliques that contain them until maximal cliques are found. While searching, the algorithm maintains three disjoint sets for each tree node: result (R), possible (P), and exclude (X). R is the set of vertices in the clique currently being explored. P and X together contain the common neighbors of the vertices in R. P is the set of common neighbors that can still be added to the clique in R in the current branch of the tree. X is the set of common neighbors that have already been considered in another branch of the tree, so they are excluded from the maximal clique being searched for in the current branch.

Algorithm 1 shows the pseudocode for the Bron-Kerbosch algorithm, and Fig. 1(c) shows how this algorithm is applied to the example graph in Fig. 1(a). In the initial call to BronKerbosch, R is empty, P contains all the vertices in the graph, and X is empty. The recursive step (lines 5-8) iterates over all the vertices v in the set P. At the recursive call (line 6), v is added to the solution R, and P and X are intersected with $N(v)$ to remove non-neighbors of v. After returning from the call, all maximal cliques containing the vertices in $R \cup \{v\}$ have been found. To avoid finding the same cliques again, v is excluded from the search in later subtrees at the same level by removing v from P (line 7) and adding to X (line 8) before proceeding to the next loop iteration.

Algorithm 1 Bron-Kerbosch algorithm

```
1: procedure BronKerbosch(G, R, P, X)
2:     if P and X are both empty then
3:         R is a maximal clique
4:         return
5:     for v ∈ P do
6:         BronKerbosch(G, R ∪ {v}, P ∩ N(v), X ∩ N(v))
7:         P = P − {v}
8:         X = X ∪ {v}
```

The recursion stops when P is empty, which means that there are no more vertices that can be added to the clique in R. If P and X are both empty (line 2), then the vertices in R have no common neighbors, which means that R represents a maximal clique (line 3). If P is empty but X is not empty, then the vertices in R do have common neighbors (those in X) and R is not a maximal clique. However, the search stops because the common neighbors in X are excluded from the search on this tree branch, which means that any maximal clique containing R has already been found in other branches.

C. Bron-Kerbosch with Pivoting

It is clear from Fig. 1(c) that there can be many branches in the search tree that are not successful at finding a maximal clique because the clique is found by other branches. To avoid some of the unsuccessful branches, Bron and Kerbosch introduce *pivoting* [18]. Algorithm 2 shows the pseudocode for the Bron-Kerbosch algorithm with pivoting, and Fig. 1(d) shows how this algorithm is applied to the example graph in Fig. 1(a). The difference from Algorithm 1 is that in Algorithm 2, a *pivot vertex* is selected prior to branching (line 5) and the neighbors of the pivot vertex are excluded from the branching (line 6).

Algorithm 2 Bron-Kerbosch algorithm with pivoting

```
1: procedure BronKerboschPivot(G, R, P, X)
2:     if P and X are both empty then
3:         R is a maximal clique
4:         return
5:     v_pivot = choose a vertex from P ∪ X
6:     for v ∈ (P − N(v_pivot)) do
7:         BronKerboschPivot(G, R ∪ {v}, P ∩ N(v), X ∩
       N(v))
8:         P = P − {v}
9:         X = X ∪ {v}
```

The intuition behind pivoting is that any maximal clique that includes the pivot vertex and its neighbor will be found by the branch that adds the pivot vertex to R. On the other hand, any maximal clique that does not include the pivot vertex but includes its neighbor must include a non-neighbor of the pivot vertex, and will be found by the branches that add non-neighbors of the pivot vertex to R. Therefore, there is no need to explore the pivot vertex's neighbors on line 6.

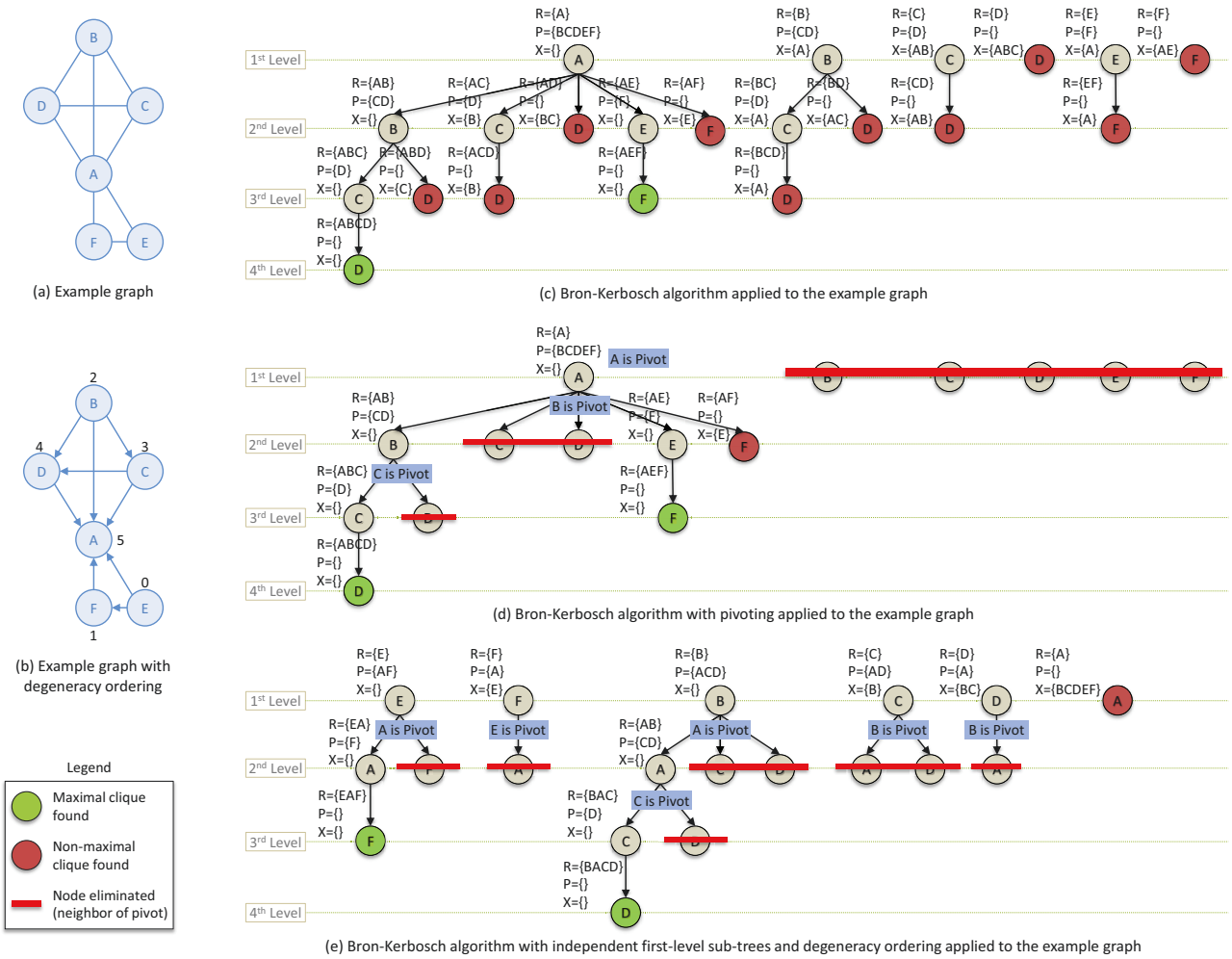

Fig. 1. Bron-Kerbosch algorithm variants applied to the example graph

The pivot vertex can be any vertex in $P \cup X$, but is typically selected to have the largest number of neighbors that are also in P in order to maximize the number of branches that are excluded from the search. The original Bron-Kerbosch algorithm with pivoting selects the pivot vertex from P, but Tomita et al. [27] improve the pivot selection by considering all vertices in $P \cup X$.

D. Bron-Kerbosch with Other Optimizations

Eppstien et al. [28], [29] further improve the Bron-Kerbosch algorithm with pivoting by introducing three key optimizations: independent first-level subtrees, degeneracy ordering, and induced subgraphs.

Independent First-level Subtrees. In Algorithms 1 and 2, the loop that iterates over the vertices in P has a loop-carried dependence for removing previously visited vertices from P and adding them to X. Eppstien et al. [28], [29] break this loop-carried dependence at the first level by having each iteration independently remove all previous vertices from P and add them to X. The pseudocode for doing so is shown in Algorithm 3. In each iteration, P for vertex v_i is initialized by intersecting $N(v_i)$ with the set of vertices that come after v_i (line 3), which removes the neighbors of v_i visited on prior iterations. On the other hand, X for vertex v_i is initialized by intersecting $N(v_i)$ with the set of vertices that come before v_i (line 4), which *keeps* the neighbors of v_i visited on prior iterations. The advantage of breaking this loop-carried dependence is that the loop iterations, which represent first-level subtrees, can be executed in parallel. For the second level onward, the algorithm simply calls the sequential BronKerboschPivot function (line 5).

Degeneracy Ordering. In Algorithm 3, the subtree for each vertex v_i only considers the vertices in P. Moreover, in Algorithm 2 (which is called by Algorithm 3 on line 5), several expensive set operations are performed with P such as $P - N(v_{pivot})$ (line 6) and $P \cap N(v)$ (line 7). Hence, the size

Algorithm 3 Bron-Kerbosch algorithm with independent first-level subtrees

```
1: procedure BronKerboschIndependentFirstLevel(G)
2:     for v_i ∈ V do
3:         P = N(v_i) ∩ {v_{i+1}, v_{i+2}, ..., v_{|V|-1}}
4:         X = N(v_i) ∩ {v_0, v_1, ..., v_{i-1}}
5:         BronKerboschPivot(G, {v_i}, P, X)
```

of P directly impacts the size of the subtree traversed and the cost of the set operations performed by the traversal. Recall that P represents the neighbors of v_i that are ordered after v_i, and X represents the neighbors of v_i that are ordered before v_i. For an arbitrary graph, the sizes of P and X are $O(\Delta)$ where Δ denotes the maximum degree of the graph and can be quite large for real graphs. To place a tighter bound on the size of P, Eppstien et al. [28] propose to reorder vertices based on *degeneracy ordering* which minimizes the maximum number of neighbors of any vertex that are ordered after that vertex. After degeneracy ordering, the maximum number of neighbors of any vertex that are ordered after that vertex is known as the degeneracy of the graph, and is denoted by d. The size of P thus becomes $O(d)$. For real graphs, d is typically much smaller than Δ (see Table I).

The advantage of degeneracy ordering is that by placing a smaller bound on the sizes of the P sets, it places a smaller bound on the sizes of the subtrees traversed and the cost of the set operations performed with P. However, the size of the X sets remains $O(\Delta)$, and the practical size of the maximum X set increases due to degeneracy ordering. Hence, the trade-off of degeneracy ordering is that it makes the operations on the X sets, such as $X \cap N(v)$ (line 7 in Algorithm 2), more expensive.

In Fig. 1(a), the first vertex A was also the vertex with the highest degree, which resulted in large subtrees being visited for vertex A in Fig. 1(c) and Fig. 1(d). The size of the P set for A was five which is the maximum degree of the graph. Fig. 1(b) shows how the graph in Fig. 1(a) can be reordered based on degeneracy ordering. In this figure, the graph is still intended to be undirected, but the edges are drawn with arrows from vertices earlier in the order to vertices later in the order. As shown in Fig. 1(b), A is now the last vertex in the order and has no vertices ordered after it. Fig. 1(e) shows how the example graph in Fig. 1(a) can be processed using independent first-level subtrees and degeneracy ordering. It is clear that compared to Fig. 1(c) and Fig. 1(d), Fig. 1(e) has more independent subtrees that are each smaller in size, its largest P set is smaller, and its largest X set is larger.

Induced Subgraphs. In Algorithm 3, the subtree for each vertex v_i only needs to access the neighbors of v_i and their edges. It does not need to access the entire graph. Based on this observation, Eppstien et al. [28] propose to construct an induced subgraph for each subtree that only includes the information needed by that subtree. In particular, we observe that Algorithm 2 performs three key operations that access the graph. The first operation is $P - N(v_{pivot})$ (line 6). Since $v_{pivot} \in P \cup X$, this operation needs to know the neighbors of any vertex in $P \cup X$ that are in P. The second operation is $P \cap N(v)$ (line 7). Since $v \in P$, this operation needs to know the neighbors of any vertex in P that are also in P. The third operation is $X \cap N(v)$ (line 7). Since $v \in P$, this operation needs to know the neighbors of any vertex in P that are in X. Overall, the algorithm needs the edges connecting any vertex in $P \cup X$ with any vertex in P. Eppstien et al. [28] induce a subgraph that contains only this information, denoted by $H_{P,X}$. The key advantage of using an induced subgraph is that it removes irrelevant edges from the adjacency lists, making set operations on the adjacency lists smaller.

Without degeneracy ordering, the size of $P \cup X$ is $O(\Delta)$ and the size of P is $O(\Delta)$. Hence, the size of $H_{P,X}$ is $O(\Delta^2)$ which is prohibitively expensive to store for large graphs. However, after degeneracy ordering, the size of P is reduced to $O(d)$, which reduces the size of $H_{P,X}$ to $O(\Delta \cdot d)$. Since typically $d \ll \Delta$, degeneracy ordering makes it more feasible to construct and store an induced subgraph.

III. Parallelizing MCE on GPUs

A. Challenges and Implementation Overview

We propose a parallel implementation of MCE on GPUs based on the Bron-Kerbosch algorithm with pivoting, independent first-level subtrees, degeneracy ordering, and induced subgraphs. One of the main challenges for parallelizing the Bron-Kerbosch algorithm on GPUs is extracting a sufficient amount of parallelism to fully-utilize the hardware resources. The majority of prior works [19]–[22] do so by performing a breadth-first traversal of the search tree. Breadth-first search is highly amenable to parallelization because tree nodes at each level of the search tree can be processed in parallel. However, it does not scale well for large graphs because of the explosion in the number of search tree nodes that need to be tracked as the level gets deeper. To avoid this explosion, one work [30] performs depth-first search on CPU while offloading primitive operations to GPU. However, this approach results in high communication overhead between CPU and GPU due to frequent kernel calls and data transfer operations.

To overcome these limitations, we propose to parallelize MCE on GPUs by assigning independent subtrees to different thread blocks and having each thread block perform a depth-first traversal of its subtree. Threads within the block collaborate to perform primitive operations such as set operations and finding pivots. This approach prevents the explosion in the number of search tree nodes that need to be tracked, and performs the entire traversal in a single kernel which eliminates CPU-GPU communication. There is no communication between CPU and GPU throughout the execution, except copying the graph to the GPU at the beginning and copying the result back at the end.

The parallelization approach of performing per-block depth-first traversals of independent subtrees has been applied in our prior work on k-clique counting [23]. That work also applies other optimizations such as binary encoding of the induced subgraph and partitioning work within a block at subwarp

granularity. In this work, we retain all these optimizations. To the best of our knowledge, this work is the first to use induced subgraphs, binary encoding, and subwarp partitioning for parallelizing MCE on GPUs.

Aside from applying these techniques to MCE, our main contribution in this work is addressing two key scalability challenges present in MCE that are less of a concern in k-clique counting. The first challenge is that MCE has substantially higher load imbalance. In k-clique counting, search trees have bounded depth (i.e., k). Hence, the sizes of subtrees that are assigned to different thread blocks are reasonably balanced. Moreover, our prior work on k-clique counting [23] shows that extracting independent subtrees at the second level instead of the first level is sufficient to balance the load completely. In contrast, in MCE, the subtrees may be arbitrarily deep depending on the size of the maximal clique they are exploring. Hence, MCE suffers from substantially higher load imbalance than k-clique counting and requires more sophisticated load balancing techniques.

The second challenge is that MCE has a substantially higher memory footprint than k-clique counting. Since MCE has potentially deeper subtrees, it needs to pre-allocate more stack space per thread block to support the depth-first traversal of these subtrees. Moreover, in k-clique counting, the traversal only needs to track the equivalent of the R and P sets at each level of the tree, which are $O(d)$ in size, and the induced subgraphs only need to store the edges between vertices in P and other vertices also in P, which requires $O(d^2)$ space. In contrast, in MCE, to test for maximality, the traversal also needs to track the X set for each level of the tree, which is $O(\Delta)$ in size, and the induced subgraphs also need to store the edges between vertices in X and vertices in P, which requires $O(\Delta \cdot d)$ space. Since Δ is much larger than d, MCE has a substantially higher memory footprint than k-clique counting and requires more sophisticated techniques for representing induced subgraphs and the X sets.

In the rest of this section, we describe our proposed approach for parallelizing MCE on GPUs, with a particular focus on unique aspects of our work, namely, how to mitigate load imbalance and how to efficiently represent induced subgraphs and the X sets at each level of the search tree.

B. Independent Second-level Subtrees

One common approach to improving load balance on GPUs is to extract many more parallel tasks than the number of tasks that can be executed simultaneously by the hardware. Our prior work on k-clique counting [23] advocates for extracting independent subtrees at the second level instead of the first, and shows that it is sufficient to balance load completely for that problem. We investigate the same technique in MCE.

Algorithm 4 shows the pseudocode for extracting independent second-level subtrees. Instead of iterating over vertices in V, we iterate over edges $\{v_i, v_j\}$ in E (line 2). For each edge, P is initialized to the common neighbors of v_i and v_j that are ordered after both vertices (line 3). On the other hand, X is initialized to the common neighbors of v_i and v_j that are ordered before the later of the two vertices (line 4).

Algorithm 4 Bron-Kerbosch algorithm with independent second-level subtrees

```
1: procedure BronKerboschIndependentSecondLevel(G)
2:     for {v_i, v_j} ∈ E do
3:         P = N(v_i) ∩ N(v_j) ∩ {v_{max(i,j)+1}, ..., v_{|V|-1}}
4:         X = N(v_i) ∩ N(v_j) ∩ {v_0, ..., v_{max(i,j)-1}}
5:         BronKerboschPivot(G, {v_i, v_j}, P, X)
```

The advantage of extracting subtrees at the second level instead of the first level is that it provides more parallel tasks to assist with load balancing. It also results in smaller induced subgraphs since the P sets at the second level are smaller than those at the first level. The disadvantage is that more induced subgraphs need to be constructed overall, and their construction cost is amortized across smaller subtree traversals.

We evaluate the trade-off between extracting subtrees at the first or second level throughout Section IV. We observe that although extracting second-level subtrees partially reduces load imbalance, the imbalance remains high for many graphs unlike in k-clique counting. This observation motivates us to propose another optimization for mitigating load imbalance in MCE, which is more effective and ultimately obviates the need for extracting second-level subtrees.

C. Dynamic Load Balancing with a Worker List

One approach to alleviate load imbalance on GPUs is using a *worklist*. Thread blocks with large tasks can add subtasks to the worklist, and thread blocks that complete their tasks can remove subtasks from the worklist. For example, Yamout et al. [31] use such an approach to achieve load balance while traversing the vertex cover search tree, leveraging the broker worker distributor [32] as their worklist data structure. However, in MCE, the data needed to represent a subtask is large, consisting of R, P, X, the current level, and a reference to the induced subgraph. The large size of the subtask data makes using a worklist inefficient for MCE for two reasons. The first reason is that a large amount of memory would be needed to store the worklist entries, which would place high pressure on the already constrained memory capacity. The second reason is that adding and removing subtasks from the worklist would incur high overhead, so there would be a high penalty when a block adds work to a worklist and there are no idle blocks actually needing any work.

To avoid the limitations of using a worklist, we instead propose to use a *worker list* for dynamic load balancing. The worker list holds IDs of thread blocks that are idle because they have completed their previous tasks. A thread block that completes its task adds its ID to the worker list to indicate that it can receive subtasks from other blocks. We call this block a *receiver* block. On the other hand, a thread block working on a large task periodically checks the worker list to see if there are any receiver blocks waiting. We call this block a *donor* block. If a donor block finds a receiver block in the worker

list, the donor block removes the receiver block and gives it a subtask to work on. The computation terminates when all blocks have added themselves to the worker list and there are no more executing donor blocks.

We incorporate our proposed worker list technique into our parallel MCE implementation as follows. We start by launching as many thread blocks as the maximum number that can run on the GPU simultaneously. These blocks execute in two phases. In the first phase, each block atomically increments a shared counter to reserve an independent first- or second-level subtree, and traverses that subtree. If the block completes the subtree, it atomically increments the counter again to obtain another subtree. This process continues until all the independent subtrees have been depleted, after which the second phase begins. Note that there is no global synchronization needed between the two phases. Donor blocks that are still executing their subtrees from the first phase know when the second phase has been reached by checking the shared counter every time they branch.

In the second phase, blocks that finish traversing their subtrees add themselves to the worker list and sleep by spinning on a flag with exponential back-off. The worker list is implemented as a multi-producer multi-consumer queue using a circular buffer. The buffer cannot overflow because the number of thread blocks is fixed. Donor blocks that have not finished traversing their subtrees check the worker list upon visiting a new branch. If a donor finds a receiver in the worker list, the donor atomically removes the receiver's ID from the worker list, offloads the new branch to the receiver by initializing the receiver's data structures, and wakes the receiver up by setting its flag. We use CUDA atomic objects from libcu++ and the release-acquire model to guarantee that data written by the donor is visible to the receiver.

In some cases, the benefit of a donor block offloading a branch to a receiver block is not worth the overhead. To avoid these unprofitable cases, we only have a donor block check the worker list and offload a branch if two conditions hold. The first condition is that the branch to be offloaded should not be small, otherwise the overhead of offloading the branch to the receiver may be higher than the cost of visiting the branch. To ensure that the branch is not small, we require that $|P| \geq 10$ for the root node of the branch, however we note that performance is not very sensitive to the choice of this threshold. The second condition is that the donor block should have a substantial amount of other work to do after offloading the branch, because it does not make sense for the donor to offload a branch, then finish traversing its subtree shortly after and start seeking work from other donors. To ensure that the donor has a substantial amount of other work, we only offload a branch if there are other branches at the same level *and* other branches in previous levels that have not yet been explored.

We evaluate the advantage of using a worker list in Sections IV-C and IV-D, including its importance when scaling to multiple GPUs.

D. Partial Induced Subgraphs

Recall from Section II-D that one common optimization to reduce the size of adjacency lists and intersection operations is to construct, for each independent subtree, an induced subgraph with vertices and edges relevant to that subtree. Prior works that implement MCE on GPUs [19]–[22], [30] do not apply this optimization because they do not perform depth-first traversal of independent subtrees entirely on the GPU. To the best of our knowledge, our work is the first to use induced subgraphs for MCE on GPUs.

As mentioned in Section III-A, the induced subgraphs in MCE contain the edges between the vertices in P and the vertices in $P \cup X$, which makes the size of the induced subgraph $O(\Delta \cdot d)$. Since Δ can be large, these induced subgraphs are expensive to construct and store. To address this challenge, we propose to represent the induced subgraph using two alternatives: full or partial. For full induced subgraphs, we construct binary-encoded induced subgraphs containing all the edges between P and $P \cup X$. For partial induced subgraphs, we construct binary-encoded induced subgraphs with only the edges between vertices P and other vertices also in P, and use the original graph to look up edges between vertices in P and vertices in X. The original graph is stored using the Compressed Sparse Row (CSR) format [24] when first-level subtrees are used, and both the CSR and the Coordinate format (COO) [24] when second-level subtrees are used.

The advantage of using full induced subgraphs is that it makes set operations on X faster by using bitwise operations. The advantage of using partial induced subgraphs is that it avoids the high latency of constructing large induced subgraphs and the high memory capacity required for storing them. We evaluate these trade-offs and propose a heuristic for selecting the most suitable alternative in Section IV-E.

E. Compact Representation of the X Sets

Recall from Section III-A that MCE puts higher pressure on the memory capacity than other related problems because of the need to represent X at each level of a subtree to test for maximality. A subtree can have up to d levels, and the size of X is $O(\Delta)$. Therefore, a naive representation of the X sets would require $O(\Delta \cdot d)$ memory per subtree. Hence, the memory needed to represent the X sets can easily limit the number of subtrees that can be traversed in parallel.

To design an efficient representation of X, we first make the following observations. In Algorithm 2, the two operations that modify X as the tree is traversed are $X \cap N(v)$ (line 7) and $X \cup \{v\}$ (line 9), where $v \in P$. The first operation, $X \cap N(v)$, can only remove vertices from X. The second operation, $X \cup \{v\}$, adds vertices to X but these vertices can only come from P. Based on this observation, we divide the representation of X into two parts: X_P and X_X.

X_P represents the vertices in X that are part of the original P set at the root node of the subtree. These vertices may be added by the $X \cup \{v\}$ operation or removed by the $X \cap N(v)$ operation. Hence, X_P may grow or shrink as we descend to deeper levels of the subtree. However, X_P may not exceed

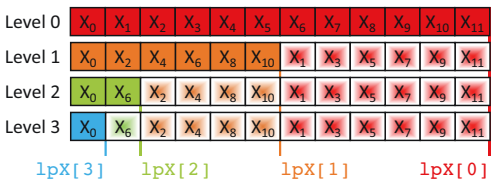

Fig. 2. Using a single array to represent X_X across levels

the size of P which is $O(d)$. For this reason, X_P is binary encoded for fast set operations on it, and a different copy of X_P is stored for each level of the tree.

On the other hand, X_X represents the vertices in X that were part of the original X set at the root node of the subtree. X_X may contain any vertex in the original X which makes its size $O(\Delta)$. However, since the vertices in the original X cannot be part of any P set in the subtree, the vertices in X_X may only be removed by the $X \cap N(v)$ operation as we descend to deeper levels of the subtree. Since X_X only shrinks as we descend down the subtree, we do not need to store a separate copy of X_X for each level. Instead, we store a single copy of X_X for all levels and an index for each level that points to where the X_X vertices end for that level.

Fig. 2 shows an example of how X_X is represented and updated as we descend down the tree. In this example, as we descend from Level 0 to Level 1, the vertices that remain in X in Level 1 are moved to the front of the array and the vertices that are removed are moved to the end of the array. To move the vertices, we implement an out-of-place partition operation where each thread moves one value after atomically incrementing a bin counter. We also tried the in-place partition operation in CUB [33] but it did not yield better performance. After moving the vertices, a level pointer array `lpX` is updated such that `lpX[1]` points to where the vertices in Level 1 end. The same process is repeated on the shrunk array as we descend to deeper levels. To go back to a previous level, nothing needs to be done since all the vertices for the previous level have stayed before the previous level's `lpX` pointer and only the order of vertices has changed.

By storing a single copy of X_X for all levels and different copies of X_P for each level, the memory needed for representing X for all levels becomes $O(\Delta + d^2)$ which is much smaller than $O(\Delta \cdot d)$. This compact representation is crucial for scalable acceleration of MCE on GPUs (and any other memory constrained system) and is used in all our implementations.

Finally, we note that if a partial induced subgraph is used instead of a full induced subgraph (see Section III-D), then the pivot is only selected from $X_P \cup P$. The reason is that finding a pivot vertex from X_X is expensive if the adjacency lists of the vertices in X_X are not binary encoded.

IV. Evaluation

A. Methodology

Evaluation Platforms. We evaluate our GPU implementations on two platforms. The first platform has four 32GB NVIDIA V100 GPUs attached to an Intel Xeon Gold 6230 CPU and is used for both single- and multi-GPU evaluation. On this platform, we compile our code with NVCC (CUDA 10.2) and GCC 8.3.1 with the -O3 flag. The CUDA driver version is 470.74. The second platform has a 40GB NVIDIA A100 GPU attached to an AMD EPYC 7702 CPU and is used for the single-GPU evaluation only. On this platform, we compile our code with NVCC (CUDA 11.4) and GCC 9.4.0 with the -O3 flag. The CUDA driver version is 470.103. We use 128 threads per block, which results in 1,280 blocks per GPU for V100 and 1,728 blocks per GPU for A100. We use CUB 1.8.0 [33] for the filter and exclusive scan operations during pre-processing. In the multi-GPU implementations, we use OpenMP 4.5 to create one CPU thread for each GPU.

CPU Baseline. To the best of our knowledge, the work of Blanuša et al. [25] is the state-of-the-art parallel CPU implementation of MCE, and also outperforms all prior GPU implementations. We compare the performance of our GPU implementation with the best execution times reported by Blanuša et al. which are obtained using a dual-socket Intel Xeon Skylake platform with 48 cores (96 threads) and 360 GB of main memory. For completeness, we also execute their publicly available code on our dual-socket Intel Xeon Gold 6230 Cascade Lake CPU with 40 cores (80 threads) and 512GB of main memory and report those results as well.

GPU Baseline. Prior GPU implementations do not have publicly available code. For this reason, we compare the performance of our implementation to the execution times reported in the most recent GPU work by Wei et al. [22]. However, this comparison is not fair because Wei et al. use an NVIDIA Titan X GPU which is weaker than the GPUs we use. We comment on this issue in Section IV-B.

Datasets We evaluate using the same graph datasets used by Blanuša et al. [25] which are shown in Table I.

Reporting of Measurements. The execution times reported by Blanuša et al. [25] include the time spent on counting maximal cliques and exclude the time spent on reading the graph from disk. For fair comparison, we follow the same strategy. We also include the time spent on pre-processing the graph to apply degeneracy ordering. Unless otherwise specified, we report the time achieved with the worker list enabled, and with the best combination of using independent first- or second-level subtrees and using partial or full induced subgraphs.

B. Performance

Performance comparison with prior CPU implementation. Table I compares the execution time of our single-GPU implementation with the state-of-the-art parallel CPU implementation [25]. We observe that our GPU implementation consistently and significantly outperforms the parallel

TABLE I
GRAPHS USED FOR EVALUATION AND COMPARISON OF EXECUTION TIME WITH THE STATE-OF-THE-ART PARALLEL CPU IMPLEMENTATION

Graph	$\|V\|$	$\|E\|$	Max degree (Δ)	Degeneracy (d)	# of maximal cliques	Avg maximal clique size	Max clique size	Parallel CPU time (s)		GPU time (s)		GPU speedup over Skylake with 96 threads	
								Cascade Lake with 80 threads	Skylake with 96 threads [25]	V100	A100	V100	A100
wiki-talk [34]	2,394,385	4,659,565	100,029	131	86,333,306	13.37	26	4.57	4	1.39	1.38	2.89	2.91
as-skitter [34]	1,696,415	11,095,298	35,455	111	37,322,355	19.91	67	3.74	3	0.81	0.79	3.70	3.82
socfb-B-anon [35]	2,937,613	20,959,854	4,356	63	27,593,398	5.24	24	2.38	2	0.56	0.45	3.60	4.42
soc-pokec [34]	1,632,804	22,301,964	14,854	47	19,376,873	3.67	29	1.45	1	0.38	0.36	2.65	2.78
wiki-topcats [34]	1,791,489	25,444,207	238,342	99	27,229,873	4.46	39	2.09	2	0.83	0.87	2.42	2.30
soc-livejournal [35]	4,033,138	27,933,062	2,651	213	38,413,665	29.97	214	5.45	5	0.81	0.76	6.21	6.57
soc-orkut [35]	3,072,442	117,185,083	33,313	253	2,269,631,973	20.24	51	110.61	93	25.23	17.82	3.69	5.22
soc-sinaweibo [35]	58,655,850	261,321,033	278,489	193	1,117,416,174	18.43	44	67.78	54	16.40	13.60	3.29	3.97
aff-orkut [35]	8,730,858	327,036,486	318,268	471	417,032,363	2.53	6	138.89	147	14.40	8.82	10.21	16.67
clueweb09-50m [35]	428,136,613	446,766,953	308,477	192	1,001,323,679	15.21	56	99.29	90	14.90	10.03	6.04	8.97
wiki-link [35]	27,154,799	543,183,611	4,271,341	1,120	568,730,123	4.51	428	112.14	109	34.82	32.23	3.13	3.38
soc-friendster [35]	65,608,367	1,806,067,135	5,214	304	3,364,773,700	6.88	129	406.33	380	64.50	39.59	5.89	9.60

CPU implementation for all graphs. The geometric mean speedup of our GPU implementation over the parallel CPU implementation is 4.1× (up to 10.2×) for the V100 GPU and 4.9× (up to 16.7×) for the A100 GPU. These results show the effectiveness of GPUs at accelerating MCE, despite the challenges of GPUs being more sensitive to load imbalance and having more constrained memory capacity. Note that while some of the optimizations introduced in this paper may be applied to CPU implementations, we do not expect them to be as effective because CPU implementations do not suffer as much from load imbalance and memory capacity constraint.

Performance comparison with prior GPU implementation. Table II compares the execution time of our single-GPU implementation with the most recent GPU implementation [22] for the common graphs reported in that implementation. The geometric mean speedup of our GPU implementation over the prior GPU implementation is 35.65× (up to 50.46×). As mentioned in Section IV-A, this comparison is not fair because Wei et al. use a Titan X GPU which is weaker than our V100 GPU. However, the V100 GPU has only 1.43x more cores and 1.88x higher memory bandwidth than the Titan X GPU so the achieved speedup cannot be attributed to the hardware difference alone. Unfortunately, we are unable to make a direct comparison on the same system because we do not have access to an NVIDIA Titan X GPU to evaluate our implementation on, nor do we have access to Wei et al.'s code to evaluate it on our system.

Performance relative to hardware peak capability. The efficiency of our implementation relative to the hardware peak capability depends on the graph being solved. For many graphs, the computation is compute-bound (computing set intersections) when the induced subgraph fits in the L1 cache. It shifts towards memory-bandwidth-boundedness when the induced subgraph is large and global memory needs to be accessed frequently. The SM utilization ranges between 25.26% and 70.71%, with a mean of 57.76%. The memory bandwidth utilization ranges between 19.63% and 59.50%, with a mean of 51.47%. Moreover, the SIMD utilization ranges between 66.4% and 91.8%, with a mean of 74.5%.

C. Load Balance

Fig. 3 compares the distribution of load across SMs for the A100 GPU when different combinations of optimizations are applied. The load of an SM is measured as the maximum number of tree nodes visited by any block on that SM (recall that we launch exactly the maximum number of concurrent blocks that can execute and reuse these blocks to process different subtrees). Based on these results, we make three key observations.

TABLE II
COMPARISON WITH THE GPU BASELINE

Graph	GPU baseline on Titan X (s) [22]	Our implementation on V100 (s)	Speedup over GPU baseline
wiki-talk	41.09	1.39	29.56
as-skitter	40.87	0.81	50.46
soc-pokec	12.85	0.38	33.82
wiki-topcats	26.57	0.83	32.01

The first observation is that when no worker list is used (No WL), using independent second-level subtrees (L2) instead of independent first-level subtrees (L1) substantially reduces load imbalance. This observation is consistent with our prior work on k-clique counting [23]. However, unlike our prior work, we note that in the case of MCE, even after L2 trees are used, the imbalance is still high for some graphs. The average across benchmarks of the maximum load across thread blocks is 2.28× the average load when using L1 trees and 1.63× the average load when using L2 trees, which is a 1.40× decrease in imbalance.

The second observation is that using a worker list (WL) substantially reduces load imbalance compared to not using a worker list. To further study the effectiveness of the worker list, Table III shows the number of donations performed for each graph. It is clear that the graphs with a large number of donations are also the ones with high imbalance in Fig. 3 that benefit from using the worker list. These results show the effectiveness of our proposed worker list approach at reducing the load imbalance of MCE on GPUs.

The third observation is that when a worker list is used, there is little difference in load imbalance between using L1 trees and L2 trees in most cases. The average across benchmarks of the maximum load across thread blocks is 1.17× the average load when using L1 trees and 1.11× the average load when using L2 trees, which is only a 1.05× decrease in imbalance. This observation shows that our proposed worker list approach obviates the need to use L2 trees for the purpose of load

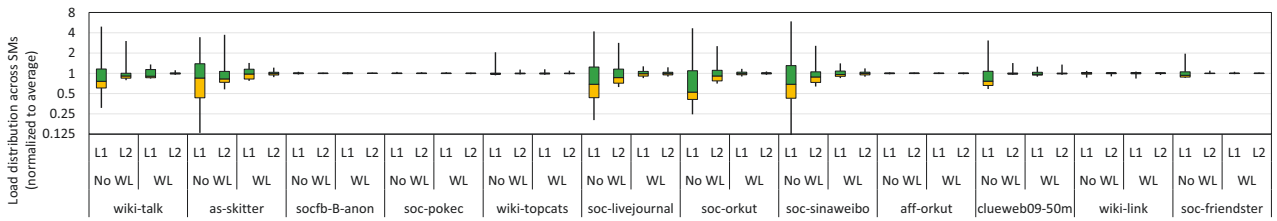

Fig. 3. Load distribution across streaming multiprocessors (SMs) for different combinations of optimizations

TABLE III
NUMBER OF DONATIONS WITH A WORKER LIST

Graph	L1	L2
wiki-talk	1,436,113	74,268
as-skitter	625,508	74,703
socfb-b-anon	170	0
soc-pokec	0	0
wiki-topcats	2,128	28
soc-livejournal	341,721	105,270

Graph	L1	L2
soc-orkut	9,728,983	2,438,166
soc-sinaweibo	11,119,926	1,164,829
aff-orkut	0	0
clueweb09-50m	890,122	49,799
wiki-link	2,997	99
soc-friendster	1,615,437	358,499

balancing in most cases. Still, using L2 trees may have other benefits such as smaller induced subgraphs and shorter set operations. We revisit this point in Section IV-E.

D. Scalability to Multiple GPUs

Fig. 4 shows the strong scaling of our GPU implementation across multiple V100 GPUs when different combinations of optimizations are applied. In the multi-GPU implementation, L1 or L2 trees are distributed across GPUs in a round-robin scheme and each GPU maintains its own private worker list. We also experimented with using an inter-GPU shared worker list, however its overhead was not worth its benefit. Currently, our implementation supports scaling to multiple GPUs within a single node, however, scaling to multiple GPU nodes is the subject of future work.

Based on the results in Fig. 4, we make three key observations. The first observation is that in most cases, on a single GPU, the implementations that use a worker list substantially outperform those that do not use a worker list. This observation shows the effectiveness of our proposed worker list approach at improving performance by reducing load imbalance.

The second observation is that in most cases, as we scale to multiple GPUs, the WL implementation scales well whereas the No WL implementation scales poorly. This observation shows that scaling to multiple GPUs exacerbates the load imbalance challenge of MCE, and that our proposed worker list approach is effective at overcoming this scalability challenge.

The third observation is that in most cases, using L2 trees instead of L1 trees has better performance and scalability when no worker list is used, but does not significantly improve performance and scalability when a worker list *is* used and may even degrade performance. This observation reiterates the observation in Section IV-C that our proposed worker list approach obviates the need to use L2 trees for the purpose of load balancing.

E. Choice of Optimizations

Fig. 5 shows the breakdown of execution time on the A100 GPU when different combinations of optimizations are applied. SM clocks are used to get the number of cycles spent by each thread block on each activity.

L1 trees vs. L2 trees. When comparing the use of L1 trees with L2 trees, we make two key observations. The first observation is that in most cases, the fraction of time spent on constructing the induced subgraph is larger for L2 trees. The reason is that using L2 trees extracts more subtrees that are each smaller in size, so more induced subgraphs are generated and the cost of generating them is amortized across fewer tree node traversals. The second observation is that in most cases, the fraction of time spent on set operations (such as intersections) is smaller for L2 trees. The reason is that using L2 trees results in smaller induced subgraphs, hence smaller sets to operate on. Nevertheless, the benefit of faster set operations does not overcome the increased overhead of constructing induced subgraphs, so we find that using L1 trees outperforms using L2 trees in the majority of cases. On average, using L1 trees is 1.2× (geometric mean) faster than using L2 trees. Note that in our prior work on k-clique counting [23], L2 trees were more effective in most cases because of their load balancing benefits. However, since these benefits are obviated by the worker list (see Sections IV-C and IV-D), the benefits of L1 trees become more pronounced.

Partial vs. full induced subgraphs. When comparing the use of partial induced subgraphs (IP, i.e., induced on P only) and full induced subgraphs (IPX, i.e., induced on P and X), we make three key observations. The first observation is that in most cases, the fraction of time spent on constructing induced subgraphs is larger for IPX. The reason is that the induced subgraphs in IPX are larger than the induced subgraphs in IP, thereby taking longer to construct. The second observation is that in most cases, the fraction of time spent on pivoting is larger for IPX. The reason is that in IPX, we consider pivots from $X \cup P$, whereas in IP, we only consider pivots in $X_P \cup P$ and do not consider pivots from X_X. As a result, we spend less time on pivoting in IP, however, we may not find the best possible pivot. The third observation is that in most cases, the fraction of time spent on set operations is smaller for IPX. The reason is that including the edges between P and X vertices in the induced subgraphs makes set operations on the X sets less costly. This trade-off between the time

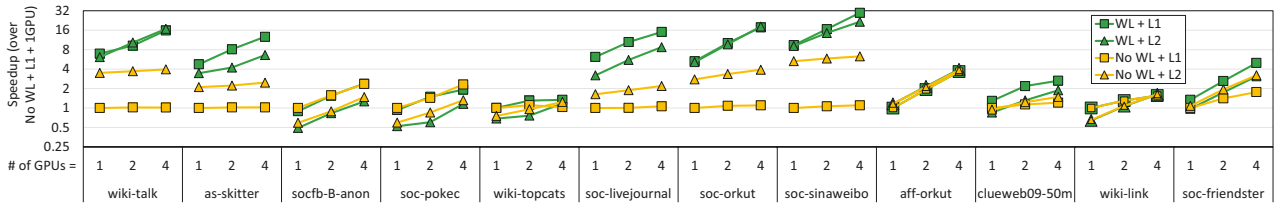

Fig. 4. Strong scaling with respect to the number of GPUs for different combinations of optimizations

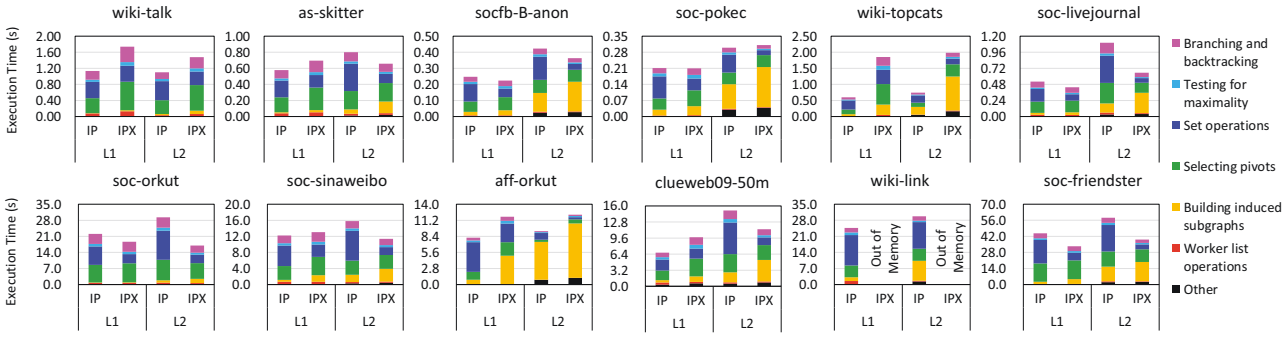

Fig. 5. Breakdown and comparison of execution time for different combinations of optimizations

spent on constructing induced subgraphs, the time spent on pivoting, and the time spent on performing set operations makes each approach perform better on different datasets. Overall, IP performs better in six cases whereas IPX performs better in six cases.

Heuristics for selecting optimizations. To select the best combination of optimizations, we recommend the following heuristic. First, L1 trees should always be selected instead of L2 trees. Second, IP should be selected when $\Delta/d > 200$, and IPX should be selected otherwise. The intuition is that IPX requires $O(\Delta \cdot d)$ space for each induced subgraph and IP requires only $O(d^2)$ space, so Δ/d represents how much more space IPX requires compared to IP. If this value is too high (IPX requires too much more space), it is better to select IP, otherwise it is better to select IPX. Table IV shows that this heuristic selects the best combination in the majority of cases, with a geometric mean slowdown of 1.02× (up to 1.10×) from selecting incorrectly. One possible additional optimization is to use different strategies for storing different induced subgraphs within the same graph based on the local ratio of degree to out-degree. This optimization is the subject of future work.

Finally, we make one additional observation from Fig. 5 that the fraction of time spent adding to and removing from the worker list is small. This observation shows that the substantial load balancing benefits that the worker list provides come with a low performance overhead. Furthermore, the fraction of time spent performing worker list operations in Fig. 5 tends to be larger for graphs where a large number of donations is performed according to Table III.

TABLE IV
COMPARING HEURISTIC (UNDERLINED) AND OPTIMAL (BOLD) SELECTION OF OPTIMIZATION COMBINATIONS

		Execution time (s)				Heuristic
Graph	Δ/d	L1 + IP	L1 + IPX	L2 + IP	L2 + IPX	slowdown
wiki-talk	763.58	1.14	1.74	**1.10**	1.48	1.03
as-skitter	319.41	**0.58**	0.70	0.80	0.66	1.00
socfb-B-anon	69.14	0.25	**0.22**	0.42	0.36	1.00
soc-pokec	316.04	0.21	**0.21**	0.30	0.31	1.00
wiki-topcats	2,407.49	**0.60**	1.85	0.74	1.99	1.00
soc-livejournal	12.45	0.52	**0.44**	1.10	0.65	1.00
soc-orkut	131.67	22.10	18.68	29.29	**16.98**	1.10
soc-sinaweibo	1,442.95	12.20	13.05	15.80	**11.39**	1.07
aff-orkut	675.73	**8.16**	11.82	9.31	12.18	1.00
clueweb09-50m	1,606.65	**6.70**	9.77	15.10	11.35	1.00
wiki-link	3,813.70	**24.67**	-	29.76	-	1.00
soc-friendster	17.15	44.64	**33.41**	58.16	39.17	1.00
					Geomean	1.02

V. Related Work

MCE has been extensively studied on CPUs [18], [27], [28], [36], [37] and many attempts to parallelize it on the CPU have been made [25], [38]–[42]. To the best of our knowledge, the work of Blanuša et al. [25] is the state-of-the-art parallel CPU implementation of MCE, and its reported performance is the highest among all prior CPU (and GPU) implementations. Our work targets accelerating MCE on GPUs. We compare the performance of our work to that of Blanuša et al. [25] in Section IV. MCE has also been parallelized on distributed CPU systems [43]–[46]. Our work focuses on parallelizing MCE on single-node single- and multi-GPU systems, however, parallelizing MCE on distributed GPU systems is an interesting future work. Many works have parallelized MCE on GPUs [19]–[22], [30]. We compare our approach to these

works in depth in Section III-A.

k-clique enumeration has been studied on CPUs [47]–[53] and GPUs [23]. Triangle counting, which is a special case of k-clique counting, has also been studied on CPUs [54]–[58] and GPUs [59]–[69]. Our MCE work uses similar techniques to those used in our prior GPU work on k-clique counting [23], namely per-block depth-first traversal, binary encoding of induced subgraphs, and subwarp partitioning. However, as discussed in Section III-A, MCE imposes unique challenges that we overcome with additional techniques such as the worker list, partial induced subgraphs, and the compact representation of excluded vertex sets.

Generalized graph pattern matching has also been studied on CPUs [70]–[74] and GPUs [75]–[83]. Cliques are special cases of patterns that graph pattern matching works aim to find. While graph pattern matching algorithms perform similar tree traversals to those in MCE, their general nature makes them more difficult to scale. For example, using induced subgraphs in graph pattern matching would require $O(\Delta^2)$ space, which causes most of these works to avoid such an optimization.

k-truss decomposition has also been studied on CPUs [84]–[87] and GPUs [88]–[93]. A truss is a relaxation of a clique, and finding trusses uses significantly different techniques that are not based on search trees. In our work, we aim to find exact maximal cliques.

VI. Conclusion

We present a GPU solution for accelerating maximal clique enumeration that assigns independent subtrees to different thread blocks and has each thread block perform a depth-first traversal of its subtree. We propose a worker list for dynamic load balancing to mitigate the high imbalance in the MCE search tree. We propose partial induced subgraphs and a compact representation of excluded vertex sets to regulate memory consumption. We also apply various optimizations used in prior works such as binary encoding of induced subgraph and partitioning work at subwarp granularity. Our evaluation shows that our GPU implementation substantially outperforms the state-of-the-art parallel CPU implementation, which outperforms prior GPU implementations.

Acknowledgment

We thank Zaid Qureshi and Amir Nassereldine for their insights and technical assistance. This work is supported in part by the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR). Izzat El Hajj acknowledges the support of the University Research Board of the American University of Beirut (AUB-URB-104391-26749). Jinjun Xiong acknowledges the support of NSF (FuSe-TG 2235364) and the joint support of NSF and IES through AI4ExceptionalEd (2229873). We are also grateful to NVIDIA's Applied Research Accelerator Program for donating A100 GPUs that were helpful in the final testing stages of this work.

Appendix
Artifact Appendix

A. Abstract

The artifact contains a pre-compiled binary for our application, a Dockerfile preparing software dependencies, and scripts for downloading datasets, running experiments, and reproducing figures and tables in Section IV. To reproduce results inside the docker image built from the Dockerfile, a CPU with 4 cores in x86_64 architecture, 128 GB of RAM, 256 GB of disk space and 4 NVIDIA GPUs of compute capability 7.0 or higher (i.e., Volta architecture or later) with 32 GB of GPU memory each are the minimum hardware requirements. We also require CUDA driver version of at least 450.80.02 with built-in CUB library, or CUDA driver version of at least 440.33 with CUB library from source in Linux OS, and Docker version above 19.03 with NVIDIA Container Toolkit as the run-time environment. Our source code is also provided in case there is a version mismatch in the environment, which may require recompilation.

B. Artifact check-list (meta-information)

- **Algorithm:** Bron-Kerbosch Algorithm for maximal clique enumeration on GPUs
- **Binary:** A pre-compiled binary built from the Makefile is provided, with software dependencies prepared in the Dockerfile.
- **Dataset:** The SNAP Datasets[1] and the Network Repository[2]
- **Run-time environment:** CUDA driver version of at least 450.80.02 with built-in CUB library, or CUDA driver version of at least 440.33 with CUB library from source[3] in Linux OS, and Docker version above 19.03 with NVIDIA Container Toolkit[4]
- **Hardware:** At least a CPU with 4 cores in x86_64 architecture, 128 GB of RAM, 256 GB of disk space and 4 NVIDIA GPUs of compute capability 7.0 or higher (i.e., Volta architecture or later) with 32 GB of GPU memory each
- **Execution:** Approximately 6 hours, might fluctuate based on downloading datasets in the first hour due to internet bandwidth
- **Output:** Figures and tables in Section IV
- **Experiments:** Building docker image, downloading datasets, running experiments, and visualizing results as figures and tables
- **Disk space required:** 200 GB
- **Publicly available:** https://github.com/yen-hsiang-chang/mce-gpu
- **Code license:** University of Illinois/NCSA Open Source License
- **Archived:** https://zenodo.org/record/8270171

C. Description

1) How to access: The artifact can be accessed from GitHub at https://github.com/yen-hsiang-chang/mce-gpu.

2) Hardware dependencies: Our experiments require at least a CPU with 4 cores in x86_64 architecture to have a unique CPU thread for each GPUs, 128 GB of RAM to load and unzip graphs, 256 GB of disk space to store datasets, and 4 NVIDIA GPUs of compute capability 7.0 or higher (i.e., Volta architecture or later) with 32 GB of GPU memory each to execute our kernels on multiple GPUs. Our multi-GPU experiments are done on a platform with four 32GB NVIDIA V100 GPUs attached to an Intel Xeon Gold 6230

[1] https://snap.stanford.edu/
[2] https://networkrepository.com/
[3] https://github.com/NVIDIA/cub
[4] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/

CPU. Other platform satisfying the requirements will achieve similar results, while the performance might fluctuate due to different computing power and memory bandwidth.

3) Software dependencies: We require CUDA driver version of at least 450.80.02 with built-in CUB library, or CUDA driver version of at least 440.33 with CUB library from source to pre-process the graphs. We also require Docker 19.03 or higher with NVIDIA Container Toolkit to make GPUs ready to be used with Docker. Other software dependencies prepared in the docker image include OpenMP 4.5 to have a unique CPU thread for each GPUs, and Python 3.6 with numpy, matplotlib and tabulate to run the pre-compiled binary, plotting figures and formatting tables.

4) Datasets: We use wiki-talk, as-skitter, soc-pokec and wiki-topcats from the SNAP datasets and socfb-B-anon, soc-livejournal, soc-orkut, soc-sinaweibo, aff-orkut, clueweb09-50m, wiki-link and soc-friendster from the Network Repository as our datasets for evaluation. A script is provided in the artifact to help download and extract datasets.

D. Installation and Experiment Instructions

1) Install Docker with NVIDIA Container Toolkit and install CUDA driver with CUB library
2) Get the artifact from GitHub
3) Build and launch the docker image:
   `$ ./docker.sh /path/to/data/ \`
   `/path/to/results/`
   Notice that `/path/to/data/` is the path on host that will store datasets and it needs to have at least 200 GB disk space, and `/path/to/results/` is the path on host that will store evaluation results
4) Inside the docker image, reproduce all experiments with the script provided:
   `$ ./all_experiments.sh`
   The whole experiments take about six hours to finish, and there might be some fluctuations since downloading datasets depends on the internet bandwidth. The `all_experiments.sh` script includes downloading datasets using `download.py`, running experiments and evaluations in load balance, time breakdown and donation using `run.py`, and plotting figures and formatting tables using `plot.py`
5) After the experiments are done, exit the docker image and inspect the results in `/path/to/results/` on host

E. Evaluation and expected results

The `/path/to/results/` directory on host contains figures and tables reported in Section IV, where figures are stored in the `plot/` sub-directory and tables are stored in the `table/` sub-directory. The descriptions are as follows and please refer to the paper for more details:

1) `load-balance.png`: Visualize distribution of loads across streaming multiprocessors (SMs) for different combinations of optimizations as in Fig. 3
2) `multigpu.png`: Visualize strong scaling experiments for different combinations of optimizations as in Fig. 4
3) `breakdown.png`: Visualize time breakdown of execution time for different combinations of optimizations as in Fig. 5
4) `time.txt`: Output the GPU time as in Table I. Note that the time includes both the degeneracy ordering time and the maximal clique counting time
5) `donation.txt`: Output the number of donations as in Table III
6) `heuristics.txt`: Output the GPU time for different combinations of optimizations as in Table IV

We expect the results to be similar as our evaluation in Section IV. However, we do expect some minor differences for the GPU time reported in Fig. 5 and Table IV in the paper, as optimization combinations are sensitive to memory bandwidth and computing power on GPUs, and different GPUs have different characteristics.

F. Notes

Our code has been open-sourced on GitHub to enable further research on accelerating maximal clique enumeration on GPUs. The repository contains a README file with instructions on running experiments without Docker and usages of the pre-compiled binary and each scripts if running individually.

REFERENCES

[1] E. Gregori, L. Lenzini, and S. Mainardi, "Parallel k-clique community detection on large-scale networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1651–1660, 2012.

[2] I. Derényi, G. Palla, and T. Vicsek, "Clique percolation in random networks," *Physical review letters*, vol. 94, no. 16, p. 160202, 2005.

[3] W. Gao, K.-F. Wong, Y. Xia, and R. Xu, "Clique percolation method for finding naturally cohesive and overlapping document clusters," in *International Conference on Computer Processing of Oriental Languages*. Springer, 2006, pp. 97–108.

[4] Z. Lu, J. Wahlström, and A. Nehorai, "Community detection in complex networks via clique conductance," *Scientific reports*, vol. 8, no. 1, pp. 1–16, 2018.

[5] P. Vilakone, D.-S. Park, K. Xinchang, and F. Hao, "An efficient movie recommendation algorithm based on improved k-clique," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, pp. 1–15, 2018.

[6] S. Manoharan, "Patient diet recommendation system using k clique and deep learning classifiers," *Journal of Artificial Intelligence*, vol. 2, no. 02, pp. 121–130, 2020.

[7] F. Glaria, C. Hernández, S. Ladra, G. Navarro, and L. Salinas, "Compact structure for sparse undirected graphs based on a clique graph partition," *Information Sciences*, vol. 544, pp. 485–499, 2021.

[8] R. A. Rossi and R. Zhou, "Graphzip: a clique-based sparse graph compression method," *Journal of Big Data*, vol. 5, no. 1, pp. 1–14, 2018.

[9] ——, "System and method for compressing graphs via cliques," Feb. 26 2019, uS Patent 10,217,241.

[10] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 217–228, 2016.

[11] H. N. Chua, K. Ning, W.-K. Sung, H. W. Leong, and L. Wong, "Using indirect protein-protein interactions for protein complex prediction," in *Computational Systems Bioinformatics: (Volume 6)*. World Scientific, 2007, pp. 97–109.

[12] M. Pellegrini, M. Baglioni, and F. Geraci, "Protein complex prediction for large protein protein interaction networks with the core&peel method," *BMC bioinformatics*, vol. 17, no. 12, pp. 37–58, 2016.

[13] L. Yang, X. Zhao, and X. Tang, "Predicting disease-related proteins based on clique backbone in protein-protein interaction network," *International journal of biological sciences*, vol. 10, no. 7, p. 677, 2014.

[14] H. Yu, A. Paccanaro, V. Trifonov, and M. Gerstein, "Predicting interactions in protein networks by completing defective cliques," *Bioinformatics*, vol. 22, no. 7, pp. 823–829, 2006.

[15] Z. Shi, C. K. Derow, and B. Zhang, "Co-expression module analysis reveals biological processes, genomic gain, and regulatory mechanisms associated with breast cancer progression," *BMC systems biology*, vol. 4, no. 1, pp. 1–14, 2010.

[16] A. Emamjomeh, E. Saboori Robat, J. Zahiri, M. Solouki, and P. Khosravi, "Gene co-expression network reconstruction: a review on computational methods for inferring functional information from plant-based expression data," *Plant biotechnology reports*, vol. 11, no. 2, pp. 71–86, 2017.

[17] V. Boginski, S. Butenko, and P. M. Pardalos, "Statistical analysis of financial networks," *Computational statistics & data analysis*, vol. 48, no. 2, pp. 431–443, 2005.

[18] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[19] T. Alusaifeer, S. Ramanna, C. J. Henry, and J. Peters, "Gpu implementation of mce approach to finding near neighbourhoods," in *International Conference on Rough Sets and Knowledge Technology*. Springer, 2013, pp. 251–262.

[20] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, "Maximal clique enumeration with data-parallel primitives," in *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2017, pp. 16–25.

[21] P. Jayaraj, K. Rahamathulla, and G. Gopakumar, "A gpu based maximum common subgraph algorithm for drug discovery applications," in *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 2016, pp. 580–588.

[22] Y.-W. Wei, W.-M. Chen, and H.-H. Tsai, "Accelerating the bron-kerbosch algorithm for maximal clique enumeration using gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2352–2366, 2021.

[23] M. Almasri, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu, "Parallel k-clique counting on gpus," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–14.

[24] W.-M. W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.

[25] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Manycore clique enumeration with fast set intersections," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2676–2690, 2020.

[26] J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari, "Modular decomposition of protein-protein interaction networks," *Genome biology*, vol. 5, pp. 1–12, 2004.

[27] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical computer science*, vol. 363, no. 1, pp. 28–42, 2006.

[28] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *International Symposium on Algorithms and Computation*. Springer, 2010, pp. 403–414.

[29] ——, "Listing all maximal cliques in large sparse real-world graphs," *Journal of Experimental Algorithmics (JEA)*, vol. 18, pp. 3–1, 2013.

[30] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the gpu," in *European Conference on Parallel Processing*. Springer, 2011, pp. 425–437.

[31] P. Yamout, K. Barada, A. Jaljuli, A. E. Mouawad, and I. El Hajj, "Parallel vertex cover algorithms on gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 201–211.

[32] B. Kerbl, M. Kenzel, J. H. Mueller, D. Schmalstieg, and M. Steinberger, "The broker queue: A fast, linearizable fifo queue for fine-granular work distribution on the gpu," in *Proceedings of the 2018 International Conference on Supercomputing*, 2018, pp. 76–85.

[33] D. Merrill, "Cub," *NVIDIA Research*, 2015.

[34] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[35] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[36] T. Yu and M. Liu, "A linear time algorithm for maximal clique enumeration in large sparse graphs," *Information Processing Letters*, vol. 125, pp. 35–40, 2017.

[37] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks," *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 4, pp. 1–34, 2011.

[38] N. S. Dasari, R. Desh, and Z. M, "pbitmce: A bit-based approach for maximal clique enumeration on multicore processors," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 478–485.

[39] T. Yu and M. Liu, "A memory efficient maximal clique enumeration method for sparse graphs with a parallel implementation," *Parallel Computing*, vol. 87, pp. 46–59, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819118301297

[40] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park, "A scalable, parallel algorithm for maximal clique enumeration," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 417–428, 2009.

[41] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura, "Shared-memory parallel maximal clique enumeration from static and dynamic graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–28, 2020.

[42] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2012, pp. 1240–1248.

[43] Y. Xu, J. Cheng, A. W.-C. Fu, and Y. Bu, "Distributed maximal clique computation," in *2014 IEEE International Congress on Big Data*. IEEE, 2014, pp. 160–167.

[44] Q. Chen, C. Fang, Z. Wang, B. Suo, Z. Li, and Z. G. Ives, "Parallelizing maximal clique enumeration over graph data," in *International Conference on Database Systems for Advanced Applications*. Springer, 2016, pp. 249–264.

[45] B. Hou, Z. Wang, Q. Chen, B. Suo, C. Fang, Z. Li, and Z. G. Ives, "Efficient maximal clique enumeration over graph data," *Data Science and Engineering*, vol. 1, no. 4, pp. 219–230, 2016.

[46] M. Svendsen, A. P. Mukherjee, and S. Tirthapura, "Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes," *Journal of Parallel and distributed computing*, vol. 79, pp. 104–114, 2015.

[47] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on computing*, vol. 14, no. 1, pp. 210–223, 1985.

[48] I. Finocchi, M. Finocchi, and E. G. Fusco, "Clique counting in mapreduce: Algorithms and experiments," *Journal of Experimental Algorithmics (JEA)*, vol. 20, pp. 1–20, 2015.

[49] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 589–598.

[50] J. Shi, L. Dhulipala, and J. Shun, "Parallel clique counting and peeling algorithms," *arXiv preprint arXiv:2002.10047*, 2020.

[51] R.-H. Li, S. Gao, L. Qin, G. Wang, W. Yang, and J. X. Yu, "Ordering heuristics for k-clique listing," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2536–2548, 2020.

[52] L. Gianinazzi, M. Besta, Y. Schaffner, and T. Hoefler, "Parallel algorithms for finding large cliques in sparse graphs," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 243–253.

[53] A. Lonkar and S. Beamer, "Accelerating clique counting in sparse real-world graphs via communication-reducing optimizations," *arXiv preprint arXiv:2112.10913*, 2021.

[54] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Information Processing Letters*, vol. 112, no. 7, pp. 277–281, 2012.

[55] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

[56] M. Halappanavar and S. Ghosh, "Tric: Distributed-memory triangle counting by exploiting the graph structure," Pacific Northwest National Lab.(PNNL), Richland, WA (United States), Tech. Rep., 2020.

[57] M. N. Kolountzakis, G. L. Miller, R. Peng, and C. E. Tsourakakis, "Efficient triangle counting in large graphs via degree-based vertex partitioning," *Internet Mathematics*, vol. 8, no. 1-2, pp. 161–185, 2012.

[58] T. Steil, T. Reza, K. Iwabuchi, B. W. Priest, G. Sanders, and R. Pearce, "Tripoll: computing surveys of triangles in massive-scale temporal graphs with metadata," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–12.

[59] M. Almasri, N. Vasudeva, R. Nagi, J. Xiong, and W.-M. Hwu, "Hykernel: A hybrid selection of one/two-phase kernels for triangle counting on gpus," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[60] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on triangle counting on gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[61] L. Wang and J. D. Owens, "Fast bfs-based triangle counting on gpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–6.

[62] S. Pandey, X. S. Li, A. Buluc, J. Xu, and H. Liu, "H-index: Hash-indexing for parallel triangle counting on gpus," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[63] Y. Hu, H. Liu, and H. H. Huang, "High-performance triangle counting on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–5.

[64] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (cpu+ gpu) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[65] M. Bisson and M. Fatica, "Update on static graph challenge on gpu," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–8.

[66] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, 2014, pp. 1–8.

[67] O. Green, P. Yalamanchili, and L.-M. Munguía, "Fast triangle counting on the gpu," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA¡sup¿3¡/sup¿ '14. IEEE Press, 2014, p. 1–8.

[68] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the gpu," in *Proceedings of the ACM Workshop on High Performance Graph Processing*, 2016, pp. 1–8.

[69] M. G. Olabi, J. G. Luna, O. Mutlu, W.-m. Hwu, and I. El Hajj, "A compiler framework for optimizing dynamic parallelism on gpus," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 1–13.

[70] A. Pinar, C. Seshadhri, and V. Vishal, "Escape: Efficiently counting all 5-vertex subgraphs," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 1431–1440.

[71] N. K. Ahmed, J. Neville, R. A. Rossi, N. G. Duffield, and T. L. Willke, "Graphlet decomposition: Framework, algorithms, and applications," *Knowledge and Information Systems*, vol. 50, no. 3, pp. 689–722, 2017.

[72] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis, "Distributed estimation of graph 4-profiles," in *Proceedings of the 25th International Conference on World Wide Web*, 2016, pp. 483–493.

[73] R. A. Rossi, R. Zhou, and N. K. Ahmed, "Estimation of graphlet counts in massive networks," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 1, pp. 44–57, 2018.

[74] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. Lui, D. Towsley, J. Tao, and X. Guan, "Moss-5: A fast method of approximating counts of 5-node graphlets in large graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 1, pp. 73–86, 2017.

[75] X. Chen, R. Dathathri, G. Gill, and K. Pingali, "Pangolin: an efficient and flexible graph mining system on cpu and gpu," *Proceedings of the VLDB Endowment*, vol. 13, no. 10, pp. 1190–1205, 2020.

[76] W. Guo, Y. Li, and K.-L. Tan, "Exploiting reuse for gpu subgraph enumeration," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[77] L. Wang, Y. Wang, and J. D. Owens, "Fast parallel subgraph matching on the gpu," in *HPDC*, 2016.

[78] W. Lin, X. Xiao, X. Xie, and X.-L. Li, "Network motif discovery: A gpu approach," *IEEE transactions on knowledge and data engineering*, vol. 29, no. 3, pp. 513–528, 2016.

[79] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *International Conference on Database Systems for Advanced Applications*. Springer, 2015, pp. 299–315.

[80] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1249–1260.

[81] L. Wang and J. D. Owens, "Fast gunrock subgraph matching (gsm) on gpus," *arXiv preprint arXiv:2003.01527*, 2020.

[82] X. Chen and A. Satyanarayan, "Efficient and scalable graph pattern mining on gpus," *arXiv preprint arXiv:2112.09761*, 2021.

[83] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "GPU-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1067–1082.

[84] J. Wang and J. Cheng, "Truss decomposition in massive networks," *arXiv preprint arXiv:1205.6693*, 2012.

[85] R. Pearce and G. Sanders, "K-truss decomposition for scale-free graphs at scale in distributed memory," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[86] T. M. Low, D. G. Spampinato, A. Kutuluru, U. Sridhar, D. T. Popovici, F. Franchetti, and S. McMillan, "Linear algebraic formulation of edge-centric k-truss algorithms with adjacency matrices," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[87] A. Conte, D. De Sensi, R. Grossi, A. Marino, and L. Versari, "Discovering *k*-trusses in large-scale networks," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–6.

[88] M. Almasri, O. Anjum, C. Pearson, Z. Qureshi, V. S. Mailthody, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on k-truss decomposition on gpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[89] S. Diab, M. G. Olabi, and I. El Hajj, "Ktrussexplorer: Exploring the design space of k-truss decomposition optimizations on gpus," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–8.

[90] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo, "Accelerating truss decomposition on heterogeneous processors," *Proceedings of the VLDB Endowment*, vol. 13, no. 10, pp. 1751–1764, 2020.

[91] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, "Quickly finding a truss in a haystack," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, 2017, pp. 1–7.

[92] M. Bisson and M. Fatica, "Update on static graph challenge on gpu," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–8.

[93] M. Blanco, T. M. Low, and K. Kim, "Exploration of fine-grained parallelism for load balancing eager k-truss on gpu and cpu," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.